0. Pandas introduction

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from plotnine import ggplot, aes
import plotnine as pn
```

Why should we use Pandas instad of just opening our table in Excel? While Excel is practical to browse through data, it is very cumbersome to use it to combine, re-arrange and thoroughly analyze data: code is hidden and difficult to share, there's no version control, it's difficult to automate tasks, the manual clicking around leads to mistakes etc.

In this course you will learn how to handle tabular data with Pandas, a Python package widely used in the scientific and data science areas. You will learn how to create and import tables, how to combine them, modify them, do statistical analysis on them and finally how to use them to easily create complex visualisations.

So that you see where this leads, we start with a short example of how Pandas can be used in a project. We look here at tables provided by the Federal Office of Statistics about all Swiss towns every year (population, number of vacant houses, unemployement etc.)

0.1 From import to plot in 3 lines

- Import: In a first step we import the spread sheet:

In [2]:
```
towns = pd.read_excel('Datasets/2013.xls', skiprows=list(range(5))+list
(range(6,9)), skipfooter=34, index_col='Commune',na_values=['*'])
```

- Then we can look at it:

In [3]:
```
towns.head()
```

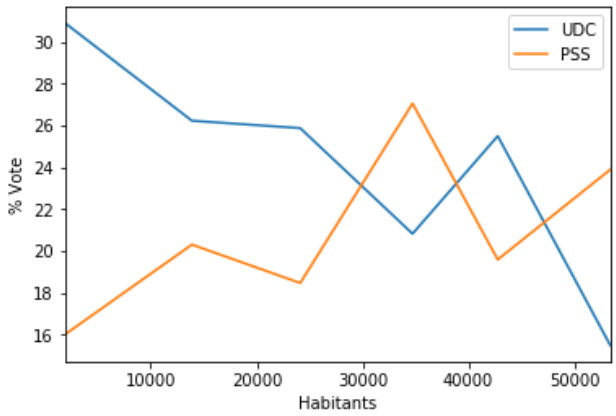
Out[3]:

	Code commune	Habitants	Densité de la population par km²	Variation en %	Par mouvement migratoire	Par excédent naturel	Etrangers en %	0-19 ans
Commune								
Aeugst am Albis	1	1910	241	4.7	4.4	0.1	13.6	21.6
Affoltern am Albis	2	11160	1052	0.6	0.0	0.4	26.0	21.4
Bonstetten	3	5173	696	0.3	-1.0	0.9	12.7	23.9
Hausen am Albis	4	3356	246	-0.1	-0.4	0.3	12.4	23.8
Hedingen	5	3469	531	1.7	0.9	0.6	14.0	24.0

5 rows × 37 columns

- And in one line do a rather complex plot where cities are split into size groups and their average vote calcualted:

In [4]:
```
ax = towns.groupby(pd.cut(towns['Habitants'], np.arange(0, 100000, 1000
0)),as_index=False).mean().dropna().plot(x = 'Habitants', y = ['UDC','PS
S'])
ax.set_ylabel('% Vote')
plt.show()
```



0.2 Combining data and assemble complex plot

Above we imported a single table. However often you need to combine information from multiple sources or multiple experiments. This can be extremely tedious to do in Excel. Here it is done in a few lines:

- import all tables in a loop

In [5]:
```
towns = []
for x in range(2013, 2015):
    temp_town = pd.read_excel('Datasets/'+str(x)+'.xls', skiprows=list(r
ange(5))+list(range(6,9)), skipfooter=34, index_col='Commune',na_values=
['*','+'])
    temp_town['year'] = x
    towns.append(temp_town)
```

- assemble the tables (concatenation)

In [6]:
```
towns_concat = pd.concat(towns,sort=False)
```

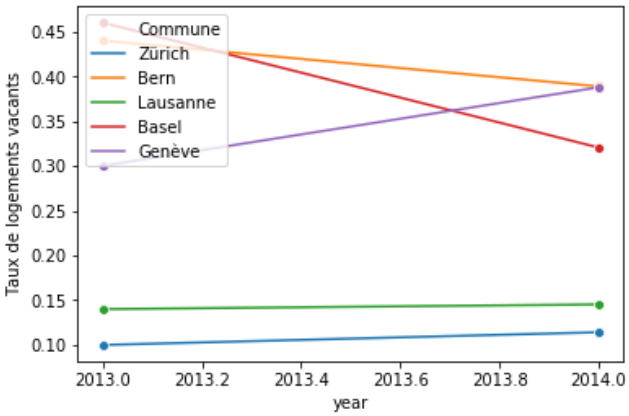
- Make a list of cities you are interested in and select only those:

In [7]:
```
cities = ['Zürich', 'Bern','Lausanne','Basel','Genève']
```

Plot for each city a parameter you are interested in like the fraction of vacant appartments:

In [8]:
```
import seaborn as sns
```

In [9]:
```
g = sns.lineplot(data = towns_concat.loc[cities].reset_index(), x = 'yea
r', y = 'Taux de logements vacants',
             hue = 'Commune', marker = "o", dashes=False);
plt.legend(loc='upper left')
plt.show()
```



We can also exploit the plotting capabilities of advanced packages such as plotnine, a Python version of ggplot, to create complex plots with little effort. For example here, we show how the voting depends on how much a town depends on agriculture. We separate the data by year as well as py party.

First we just import two years of data (two different parliaments):

In [14]:
```
towns = []
for x in [2014, 2018]:
    temp_town = pd.read_excel('Datasets/'+str(x)+'.xls', skiprows=list(r
ange(5))+list(range(6,9)), skipfooter=34, index_col='Commune',na_values=
['*','+'])
    temp_town['year'] = x
    towns.append(temp_town)
towns_concat = pd.concat(towns,sort=False)
```

We recover the necessary information and do some data reshaping (tidying) to be able to easily realize the plot:

In [15]:
```
towns_parties = towns_concat.reset_index()[['year','UDC','PS','Commune
','Surface agricole en %']]

#wide to long
towns_parties = pd.melt(towns_parties, id_vars=['Commune','Surface agric
ole en %','year'],
                        value_vars=["UDC", "PS"], value_name='Vote fract
ion', var_name='Party')
```

In [16]:
```
towns_parties.head()
```

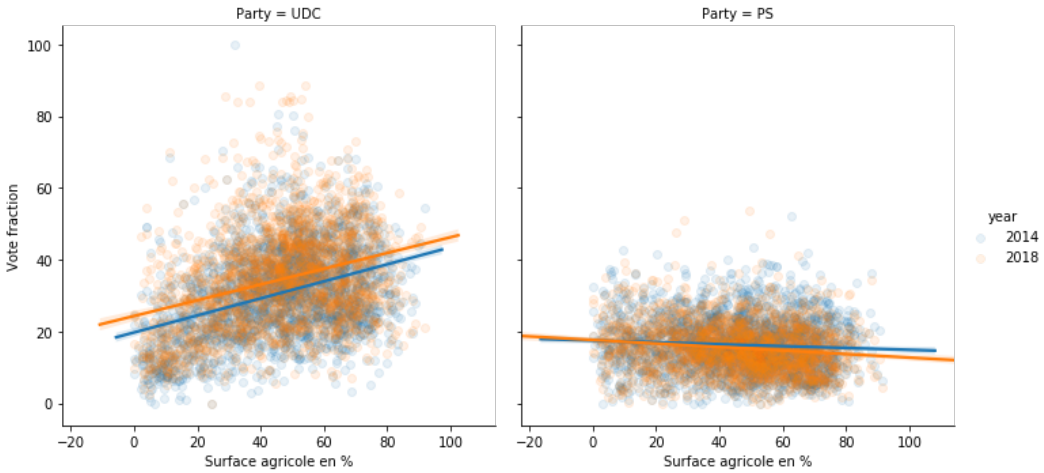
Out[16]:

	Commune	Surface agricole en %	year	Party	Vote fraction
0	Aeugst am Albis	51.334180	2014	UDC	28.652058
1	Affoltern am Albis	40.094340	2014	UDC	31.894371
2	Bonstetten	55.436242	2014	UDC	27.095457
3	Hausen am Albis	55.774854	2014	UDC	33.535200
4	Hedingen	46.248086	2014	UDC	31.438683

And finally we can plot our data:

In [17]:
```
sns.lmplot(data = towns_parties.dropna(), x = 'Surface agricole en %',
           y = 'Vote fraction', hue = 'year',col = 'Party',
            scatter_kws={'alpha' :0.1})
```

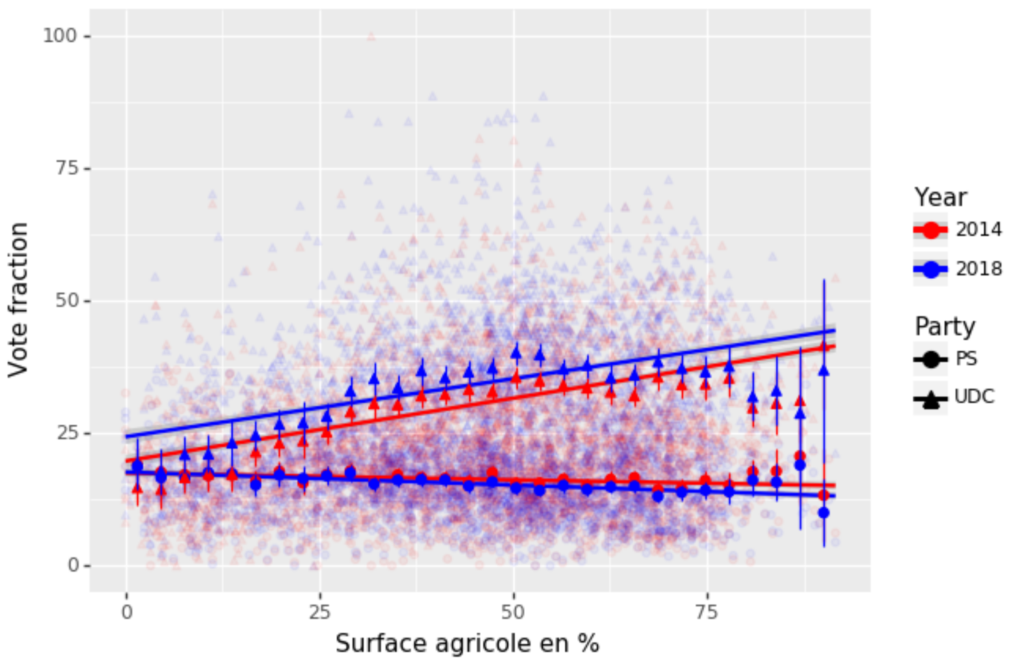
Out[17]: <seaborn.axisgrid.FacetGrid at 0x7f3a68867e10>



In [18]:
```
(ggplot(towns_parties.dropna(),aes(x = 'Surface agricole en %', y = 'Vote fraction', color = 'factor(year)', shape = 'Party'))
 + pn.geom_point(alpha = 0.05)
 + pn.geom_smooth(method='lm')
 + pn.stats.stat_summary_bin(fun_data = 'mean_cl_normal', bins = 30)#fun_y = np.mean, fun_ymin=np.var, fun_ymax=np.var, bins = 30)
 + pn.labs(color = 'Year')
 + pn.scale_color_manual(['red','blue'])
).draw();
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/computation/check.py:19: UserWarning: The installed version of numexpr 2.4.3 is not supported in pandas and will be not be used
The minimum supported version is 2.6.1

  ver=ver, min_ver=_MIN_NUMEXPR_VERSION), UserWarning)
/usr/local/lib/python3.5/dist-packages/numpy/core/fromnumeric.py:2223: FutureWarning: Method .ptp is deprecated and will be removed in a future version. Use numpy.ptp instead.
  return ptp(axis=axis, out=out, **kwargs)
/usr/local/lib/python3.5/dist-packages/numpy/core/fromnumeric.py:2223: FutureWarning: Method .ptp is deprecated and will be removed in a future version. Use numpy.ptp instead.
  return ptp(axis=axis, out=out, **kwargs)
/usr/local/lib/python3.5/dist-packages/numpy/core/fromnumeric.py:2223: FutureWarning: Method .ptp is deprecated and will be removed in a future version. Use numpy.ptp instead.
  return ptp(axis=axis, out=out, **kwargs)
/usr/local/lib/python3.5/dist-packages/numpy/core/fromnumeric.py:2223: FutureWarning: Method .ptp is deprecated and will be removed in a future version. Use numpy.ptp instead.
  return ptp(axis=axis, out=out, **kwargs)
```



1. Pandas objects

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

If you have already used Python, you know about its standard data structures (list, dicts etc). If you use Python for science, you also probably know Numpy arrays which underlying almost all other specialized scientific packages.

None of these structures offers a simple way to handle database style data, nor to easily do standard database operations. This is why Pandas exists: it offers a complete ecosystem of structures and functions dedicated to handle large tables with inhomogeneous contents.

In this first chapter, we are going to learn about the two main structures of Pandsa: Series and Dataframes.

1.1 Series

1.1.1 Simple series

Series are a the Pandas version of 1-D Numpy arrays. To understand their specificities, let's create one. Usually Pandas structures (Series and Dataframes) are creates from other simpler structures like Numpy arrays or dictionaries:

(MZ: Pandas builds on numpy; Series are equivalent of a list)

In [2]:
```
numpy_array = np.array([4,8,38,1,6])
```

In [3]:
```
# MZ:
numpy_array**2
```

Out[3]:
```
array([  16,   64, 1444,    1,   36])
```

The function `pd.Series()` allows us to convert objects into Series:

In [4]:
```
pd_series = pd.Series(numpy_array)
pd_series
# on the left => the indices; can be anything, e.g. names of towns
```

Out[4]:
```
0     4
1     8
2    38
3     1
4     6
dtype: int64
```

The underlying structure can be recovered with the `.values` attribute:

In [5]:
```
pd_series.values
# MZ: output is numpy array
```

Out[5]:
```
array([ 4,  8, 38,  1,  6])
```

Otherwise, indexing works as for regular arrays:

In [6]:
```
pd_series[1]
```

Out[6]:
```
8
```

1.1.2 Indexing

On top of accessing values in a series by regular indexing, one can create custom indices for each element in the series:

In [7]:
```
pd_series2 = pd.Series(numpy_array, index=['a', 'b', 'c', 'd','e'])
# MZ: force the index to be what we give
# MZ: to retrieve the indexes (keys)
pd_series2.keys()
```

Out[7]:
```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

In [8]:
```
pd_series2
```

Out[8]:
```
a     4
b     8
c    38
d     1
e     6
dtype: int64
```

Now a given element can be accessed either by using its regular index:

In [9]:
```
pd_series2[1]
```

Out[9]:
```
8
```

or its chosen index:

In [10]:
```
pd_series2['b']
```

Out[10]:
```
8
```

A more direct way to create specific indexes is to transform as dictionary into a Series:

In [11]:
```
# MZ: use dict to build the Series
composer_birth = {'Mahler': 1860, 'Beethoven': 1770, 'Puccini': 1858, 'S
hostakovich': 1906}
```

In [12]:
```
pd_composer_birth = pd.Series(composer_birth)
pd_composer_birth
```

Out[12]:
```
Beethoven       1770
Mahler          1860
Puccini         1858
Shostakovich    1906
dtype: int64
```

In [13]:
```
pd_composer_birth['Puccini']
```

Out[13]:
```
1858
```

1.2 Dataframes

In most cases, one has to deal with more than just one variable, e.g. one has the birth year and the death year of a list of composers. Also one might have different types of information, e.g. in addition to numerical variables (year) one might have string variables like the city of birth. The Pandas structure that allow one to deal with such complex data is called a Dataframe, which can somehow be seen as an aggregation of Series with a common index.

1.2.1 Creating a Dataframe

To see how to construct such a Dataframe, let's create some more information about composers:

In [14]:
```
composer_death = pd.Series({'Mahler': 1911, 'Beethoven': 1827, 'Puccini
': 1924, 'Shostakovich': 1975})
composer_city_birth = pd.Series({'Mahler': 'Kaliste', 'Beethoven': 'Bonn
', 'Puccini': 'Lucques', 'Shostakovich': 'Saint-Petersburg'})
```

Now we can combine multiple series into a Dataframe by precising a variable name for each series. Note that all our series need to have the same indices (here the composers' name):

In [15]:
```
# MZ: put Series together into a Dataframe
composers_df = pd.DataFrame({'birth': pd_composer_birth, 'death': compos
er_death, 'city': composer_city_birth})
composers_df
```

Out[15]:

	birth	city	death
Beethoven	1770	Bonn	1827
Mahler	1860	Kaliste	1911
Puccini	1858	Lucques	1924
Shostakovich	1906	Saint-Petersburg	1975

A more common way of creating a Dataframe is to construct it directly from a dictionary of lists:

In [16]:
```
dict_of_list = {'birth': [1860, 1770, 1858, 1906], 'death':[1911, 1827,
1924, 1975],
 'city':['Kaliste', 'Bonn', 'Lucques', 'Saint-Petersburg']}
```

In [17]:
```
pd.DataFrame(dict_of_list)
# MZ: default indexes from 0 to 3
```

Out[17]:

	birth	city	death
0	1860	Kaliste	1911
1	1770	Bonn	1827
2	1858	Lucques	1924
3	1906	Saint-Petersburg	1975

However we now lost the composers name. We can enforce it by providing, as we did before for the Series, a list of indices:

In [18]:
```
pd.DataFrame(dict_of_list, index=['Mahler', 'Beethoven', 'Puccini', 'Sho
stakovich'])
# MZ: you can explicitly pass the index
```

Out[18]:

	birth	city	death
Mahler	1860	Kaliste	1911
Beethoven	1770	Bonn	1827
Puccini	1858	Lucques	1924
Shostakovich	1906	Saint-Petersburg	1975

1.2.2 Accessing values

There are multiple ways of accessing values or series of values in a Dataframe. Unlike in Series, a simple bracket gives access to a column and not an index, for example:

In [19]:
```
composers_df['city']
```

Out[19]:
```
Beethoven                  Bonn
Mahler                  Kaliste
Puccini                 Lucques
Shostakovich    Saint-Petersburg
Name: city, dtype: object
```

returns a Series. Alternatively one can also use the *attributes* synthax and access columns by using:

In [20]:
```
composers_df.city
# rather recommended to use the brackets
```

Out[20]:
```
Beethoven                  Bonn
Mahler                  Kaliste
Puccini                 Lucques
Shostakovich    Saint-Petersburg
Name: city, dtype: object
```

The attributes synthax has some limitations, so in case something does not work as expected, revert to the brackets notation.

When specifiying multiple columns, a DataFrame is returned:

In [21]:
```
composers_df[['city', 'birth']]
```

Out[21]:

	city	birth
Beethoven	Bonn	1770
Mahler	Kaliste	1860
Puccini	Lucques	1858
Shostakovich	Saint-Petersburg	1906

One of the important differences with a regular Numpy array is that here, regular indexing doesn't work:

In [22]:
```
#composers_df[0,0]
```

Instead one has to use either the `.iloc[]` or the `.loc[]` method. `.ìloc[]` can be used to recover the regular indexing:

In [23]:
```
# MZ: recover by positions
composers_df.iloc[0,1]
```

Out[23]:
```
'Bonn'
```

While `.loc[]` allows one to recover elements by using the **explicit** index, on our case the composers name:

In [24]:
```
# MZ: recover by indices
composers_df.loc['Mahler','death']
```

Out[24]:
```
1911
```

Remember that `loc` and ``ìloc``` use brackets [] and not parenthesis ().

Numpy style indexing works here too

In [25]:
```
composers_df.iloc[1:3,:]
```

Out[25]:

	birth	city	death
Mahler	1860	Kaliste	1911
Puccini	1858	Lucques	1924

If you are working with a large table, it might be useful to sometimes have a list of all the columns. This is given by the `.keys()` attribute:

In [26]:
```
composers_df.keys()
```

Out[26]:
```
Index(['birth', 'city', 'death'], dtype='object')
```

1.2.3 Adding columns

It is very simple to add a column to a Dataframe. One can e.g. just create a column a give it a default value that we can change later:

In [27]:
```
# MZ: if pass a single value, add the same value everywhere
composers_df['country'] = 'default'
```

In [28]:
```
composers_df
```

Out[28]:

	birth	city	death	country
Beethoven	1770	Bonn	1827	default
Mahler	1860	Kaliste	1911	default
Puccini	1858	Lucques	1924	default
Shostakovich	1906	Saint-Petersburg	1975	default

Or one can use an existing list:

In [29]:
```
country = ['Austria','Germany','Italy','Russia']
```

In [30]:
```
composers_df['country2'] = country
```

In [31]:
```
composers_df
```

Out[31]:

	birth	city	death	country	country2
Beethoven	1770	Bonn	1827	default	Austria
Mahler	1860	Kaliste	1911	default	Germany
Puccini	1858	Lucques	1924	default	Italy
Shostakovich	1906	Saint-Petersburg	1975	default	Russia

2. Importing excel files

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import seaborn as sns
```

We have seen in the previous chapter what structures are offered by Pandas and how to create them. Another very common way of "creating" a Pandas Dataframe is by importing a table from another format like CSV or Excel.

2.1 Simple import

An Excel table containing the same information as we had in Chapter 1 (01-Pandas_structures.ipynb) is provided in composers.xlsx (composers.xlsx) and can be read with the `read_excel` function. There are many more readers for other types of data (csv, json, html etc.) but we focus here on Excel.

In [2]:
```
pd.read_excel('Datasets/composers.xlsx')
```

Out[2]:

	composer	birth	death	city
0	Mahler	1860	1911	Kaliste
1	Beethoven	1770	1827	Bonn
2	Puccini	1858	1924	Lucques
3	Shostakovich	1906	1975	Saint-Petersburg

The reader automatically recognized the heaers of the file. However it created a new index. If needed we can specify which column to use as header:

In [3]:
```
pd.read_excel('Datasets/composers.xlsx', index_col = 'composer')
# MZ: specify which column you want to be the index
```

Out[3]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

If we open the file in Excel, we see that it is composed of more than one sheet. Clearly, when not specifying anything, the reader only reads the first sheet. However we can specify a sheet:

In [4]:
```
pd.read_excel('Datasets/composers.xlsx', index_col = 'composer',sheet_name='Sheet2')
```

Out[4]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg
Sibelius	unknown	unknown	unknown
Haydn	NaN	NaN	Rohrau

For each reader, there is a long list of options to specify how the file should be read. We can see all these options using the help (see below). Imagine that our tables contains a title and unnecessary rows: we can use the skiprows argument. Imagine you have dates in your table: you can use the date_parser argument to specify how to format them etc.

In [5]:
```
#use shift+tab within the parenthesis to see optional arguemnts
#pd.read_excel()
```

2.2 Handling unknown values

As you can see above, some information is missing. Some missing values are marked as "unknown" while other are NaN. NaN is the standard symbol for unknown/missing values and is understood by Pandas while "unknown" is just seen as text. This is impractical as now we have e.g. columns with a mix of numbers and text which will make later computations difficult. What we would like to do is to replace all "irrelevant" values with the standard NaN symbol that says "no information".

For this we can use the na_values argument to specify what should be a NaN. Let's compare the two imports:

In [6]:
```
import1 = pd.read_excel('Datasets/composers.xlsx', index_col = 'composer',sheet_name='Sheet2')
import1
```

Out[6]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg
Sibelius	unknown	unknown	unknown
Haydn	NaN	NaN	Rohrau

In [7]:
```
# MZ: specify which values should be set to NA
import2 = pd.read_excel('Datasets/composers.xlsx', index_col = 'composer
',sheet_name='Sheet2', na_values=['unknown'])
import2
```

Out[7]:

	birth	death	city
composer			
Mahler	1860.0	1911.0	Kaliste
Beethoven	1770.0	1827.0	Bonn
Puccini	1858.0	1924.0	Lucques
Shostakovich	1906.0	1975.0	Saint-Petersburg
Sibelius	NaN	NaN	NaN
Haydn	NaN	NaN	Rohrau

If we look now at one column, we can see that columns have been imported in different ways. One column is an object, i.e. mixed types, the other contains floats.

In [8]:
```
import1.birth
```

Out[8]:
```
composer
Mahler             1860
Beethoven          1770
Puccini            1858
Shostakovich       1906
Sibelius        unknown
Haydn               NaN
Name: birth, dtype: object
```

In [9]:
```
import2.birth
```

Out[9]:
```
composer
Mahler          1860.0
Beethoven       1770.0
Puccini         1858.0
Shostakovich    1906.0
Sibelius           NaN
Haydn              NaN
Name: birth, dtype: float64
```

Handling bad or missing values is a very important part of data science. Taking care of the most common occurrences at import is a good solution.

2.3 Column types

We see above that the birth column has been "classified" as a float. However we know that this is not the case, it's just an integer. Here again, we can specify the column type already at import time using the dtype option and a dictionary:

In [10]:
```
# MZ: explicitly specify what types should be
# (e.g. to have faster computations using integers instead of floats)
import2 = pd.read_excel('Datasets/composers.xlsx', index_col = 'composer
',sheet_name='Sheet1', na_values=['unknown'],
                        dtype={'composer':np.str,'birth':np.int32,'death
':np.int32,'city':np.str})
```

In [11]:
```
import2.birth
```

Out[11]:
```
composer
Mahler          1860
Beethoven       1770
Puccini         1858
Shostakovich    1906
Name: birth, dtype: int32
```

2.4 Modifications after import

Of course we don't have to do all these adjustement at import time. We can also do a default import and check what has to be corrected afterward.

2.4.1 Create NaNs

If we missed some bad values at import we can just replace all those directly in the dataframe. We can achieve that by using the `replace()` method and specifying what should be replaced:

In [12]:
```
import1
```

Out[12]:

	birth	death	city
composer			
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg
Sibelius	unknown	unknown	unknown
Haydn	NaN	NaN	Rohrau

In [13]:
```
import_nans = import1.replace('unknown', np.nan) # MZ: what replace, wit
h what to replace as arguments
import_nans.birth
```

Out[13]:
```
composer
Mahler          1860.0
Beethoven       1770.0
Puccini         1858.0
Shostakovich    1906.0
Sibelius           NaN
Haydn              NaN
Name: birth, dtype: float64
```

Note that when we fix "bad" values, e.g. here the "unknown" text value with NaNs, Pandas automatically adjust the type of the column, allowing us for exampel to later do mathemtical operations.

In [14]:
```
import1.birth.dtype
```

Out[14]:
```
dtype('O')
```

In [15]:
```
import_nans.birth.dtype
# MZ: was before mix type -> after fixing the 'unknown' -> automatically
update the type
```

Out[15]:
```
dtype('float64')
```

2.4.2 Changing the type

We can also change the type of a column on an existing Dataframe:

In [16]:
```
import2.birth
```

Out[16]:
```
composer
Mahler          1860
Beethoven       1770
Puccini         1858
Shostakovich    1906
Name: birth, dtype: int32
```

In [17]:
```
import2.birth.astype('float')
```

Out[17]:
```
composer
Mahler          1860.0
Beethoven       1770.0
Puccini         1858.0
Shostakovich    1906.0
Name: birth, dtype: float64
```

If we look again at import2:

In [18]:
```
import2.birth
```

Out[18]:
```
composer
Mahler          1860
Beethoven       1770
Puccini         1858
Shostakovich    1906
Name: birth, dtype: int32
```

we see that we didn't actually change the type. Changes on a Dataframe are only effective if we reassign the column:

In [19]:
```
import2.birth = import2.birth.astype('float')
```

In [20]:
```
import2.birth
```

Out[20]:
```
composer
Mahler          1860.0
Beethoven       1770.0
Puccini         1858.0
Shostakovich    1906.0
Name: birth, dtype: float64
```

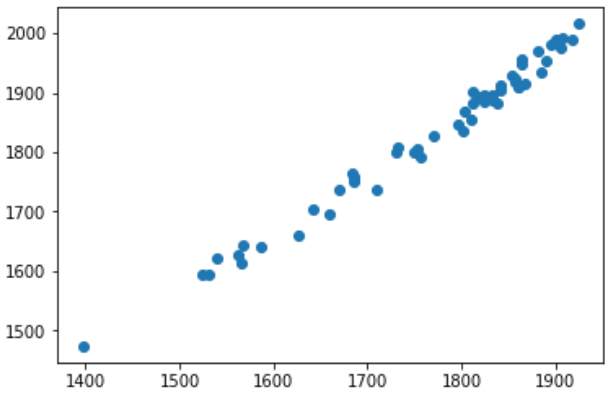
2.5 Plotting

We will learn more about plotting later, but let's see here some possibilities offered by Pandas. Pandas builds on top of Matplotlib but exploits the knowledge included in Dataframes to improve the default output. Let's see with a simple dataset.

In [21]:
```
composers = pd.read_excel('Datasets/composers.xlsx', sheet_name='Sheet5
')
```

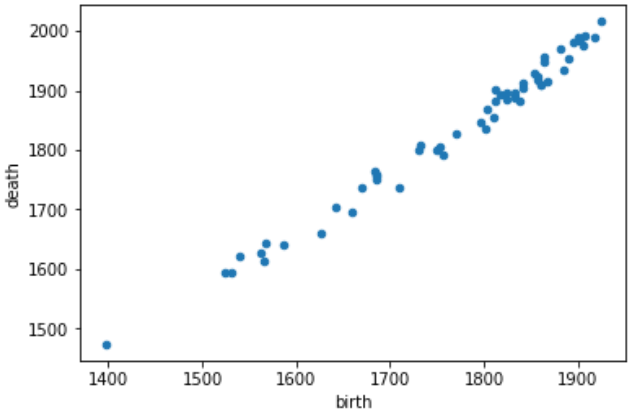
We can pass Series to Matplotlib which manages to understand them. Here's a default scatter plot:

In [22]:
```
plt.plot(composers.birth, composers.death, 'o')
plt.show()
```



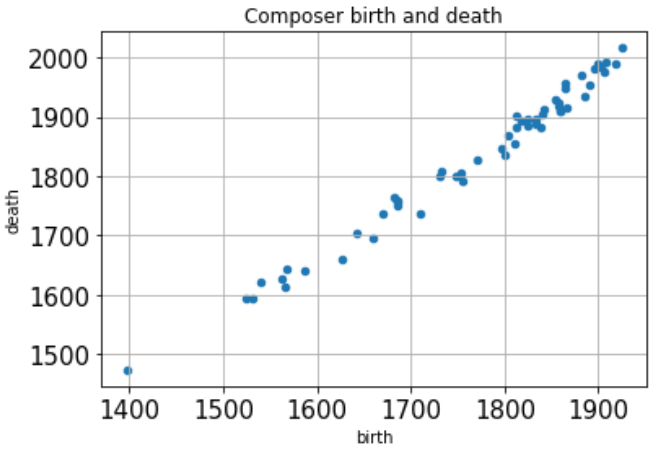
Now we look at the default Pandas output. Different types of plots are accessible when using the `data_frame.plot` function via the `kind` option. The variables to plot are column names passed as keywords instead of whole series like in Matplotlib:

In [23]:
```
composers.plot(x = 'birth', y = 'death', kind = 'scatter')
# MZ: in this way you directly have the labels
plt.show()
```



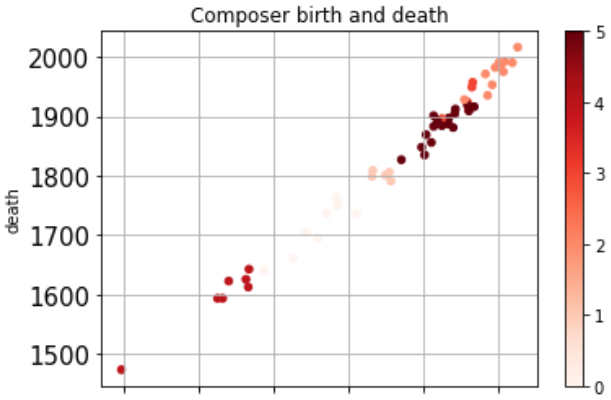
We see that the plot automatically gets axis labels. Another gain is that some obvious options like setting a title are directly accesible when creating the plot:

In [24]:
```
composers.plot(x = 'birth', y = 'death', kind = 'scatter',
                    title = 'Composer birth and death', grid = True, fontsize
= 15)
plt.show()
```



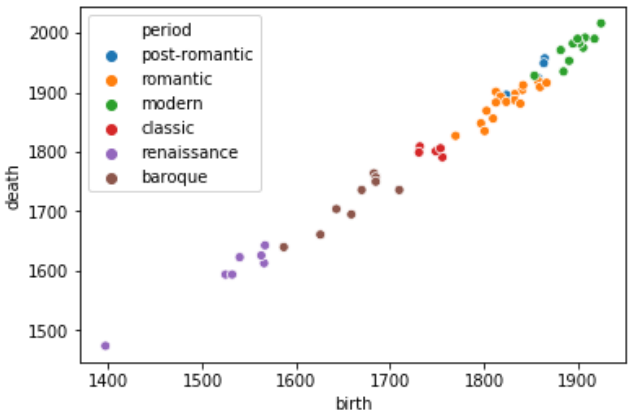
One can add even more information on the plot by using more arguments used in a similar way as a grammar of graphics. For example we can color the scatter plot by periods:

In [25]:
```
composers.plot(x = 'birth', y = 'death',kind = 'scatter',
               c = composers.period.astype('category').cat.codes, colorm
ap = 'Reds', title = 'Composer birth and death', grid = True, fontsize =
15)
plt.show()
```



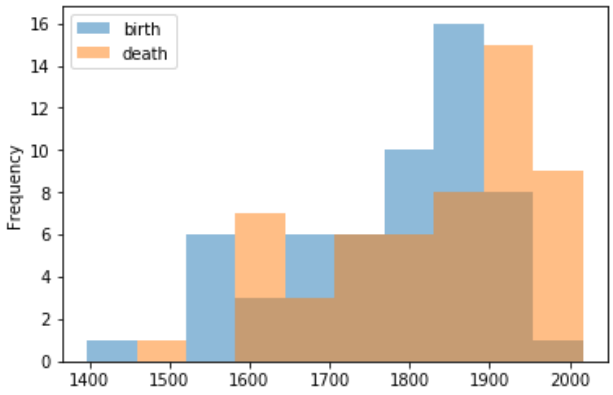
Here you see already a limitation of the plotting library. To color dots by the peiod category, we had to turn the latter into a series of numbers. We could then rename those to improve the plot, but it's better to use more specialized packages such as Seaborn which allow to realize this kind of plot easily:

In [26]:
```
sns.scatterplot(data = composers, x = 'birth', y = 'death', hue = 'perio
d') # MZ: 'hue' to get the colors with the dots
plt.show()
```



Some additional plotting options are available in the `plot()` module. For example histograms:

In [27]:
```
composers.plot.hist(alpha = 0.5)
plt.show()
```



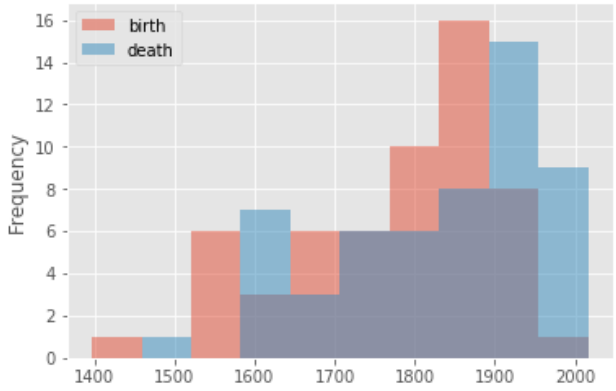
Here you see again the gain from using Pandas: without specifying anything, Pandas made a histogram of the two columns containing numbers, labelled the axis and even added a legend to the plot.

All these features are very nice and very helpful when exploring a dataset. When anaylzing data in depth and creating complex plots, Pandas's plotting might however be limiting and other options such as Seaborn or Plotnine can be used.

Finally, all plots can be "styled" down to the smallest detail, either by using Matplotlib options or by directly applying a style e.g.:

In [28]:
```
plt.style.use('ggplot')
```

In [29]:
```
composers.plot.hist(alpha = 0.5)
plt.show()
```



3. Operations with Pandas objects

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

One of the great advantages of using Pandas to handle tabular data is how simple it is to extract valuable information from them. Here we are going to see various types of operations that are available for this.

3.1 Matrix types of operations

The strength of Numpy is its natural way of handling matrix operations, and Pandas reuses a lot of these features. For example one can use simple mathematical operations to opereate at the cell level:

In [2]:
```
compo_pd = pd.read_excel('Datasets/composers.xlsx')
compo_pd
```

Out[2]:

	composer	birth	death	city
0	Mahler	1860	1911	Kaliste
1	Beethoven	1770	1827	Bonn
2	Puccini	1858	1924	Lucques
3	Shostakovich	1906	1975	Saint-Petersburg

In [3]:
```
compo_pd['birth']*2
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/computation/check.py:1
9: UserWarning: The installed version of numexpr 2.4.3 is not supported i
n pandas and will be not be used
The minimum supported version is 2.6.1

  ver=ver, min_ver=_MIN_NUMEXPR_VERSION), UserWarning)
```

Out[3]:
```
0    3720
1    3540
2    3716
3    3812
Name: birth, dtype: int64
```

In [4]:
```
np.log(compo_pd['birth'])
```

Out[4]:
```
0    7.528332
1    7.478735
2    7.527256
3    7.552762
Name: birth, dtype: float64
```

Here we applied functions only to series. Indeed, since our Dataframe contains e.g. strings, no operation can be done on it:

In [5]:
```
#compo_pd+1
```

If however we have a homogenous Dataframe, this is possible:

```
In [6]: compo_pd[['birth','death']]*2
```

Out[6]:

	birth	death
0	3720	3822
1	3540	3654
2	3716	3848
3	3812	3950

3.2 Column operations

There are other types of functions whose purpose is to summarize the data. For example the mean or standard deviation. Pandas by default applies such functions column-wise and returns a series containing e.g. the mean of each column:

```
In [7]: np.mean(compo_pd)
```

```
Out[7]: birth    1848.50
        death    1909.25
        dtype: float64
```

Note that columns for which a mean does not make sense, like the city are discarded. A series of commun functions like mean or standard deviation are directly implemented as methods and can be accessed in the alternative form:

```
In [8]: compo_pd.mean()
```

```
Out[8]: birth    1848.50
        death    1909.25
        dtype: float64
```

If you need the mean of only a single column you can of course chains operations:

```
In [9]: compo_pd.birth.mean()
```

```
Out[9]: 1848.5
```

3.3 Operations between Series

We can also do computations with multiple series as we would do with Numpy arrays:

```
In [10]: compo_pd['death']-compo_pd['birth']
         # operations between columns
```

```
Out[10]: 0    51
         1    57
         2    66
         3    69
         dtype: int64
```

We can even use the result of this computation to create a new column in our Dataframe:

In [11]:
```
compo_pd['age'] = compo_pd['death']-compo_pd['birth']
```

In [12]:
```
compo_pd
```

Out[12]:

	composer	birth	death	city	age
0	Mahler	1860	1911	Kaliste	51
1	Beethoven	1770	1827	Bonn	57
2	Puccini	1858	1924	Lucques	66
3	Shostakovich	1906	1975	Saint-Petersburg	69

3.4 Other functions

Sometimes one needs to apply to a column a very specific function that is not provided by default. In that case we can use one of the different `apply` methods of Pandas.

The simplest case is to apply a function to a column, or Series of a DataFrame. Let's say for example that we want to define the the age >60 as 'old' and <60 as 'young'. We can define the following general function:

In [13]:
```
def define_age(x):
    if x>60:
        return 'old'
    else:
        return 'young'
```

In [14]:
```
define_age(30)
```

Out[14]:
```
'young'
```

In [15]:
```
define_age(70)
```

Out[15]:
```
'old'
```

We can now apply this function on an entire Series:

In [16]:
```
compo_pd.age.apply(define_age)
# MZ: apply take variable inputs and return variable outputs
# to apply, you can pass Series, DataFrame; can return DataFrame or sing
le numbers or list of numbers, etc.
```

Out[16]:
```
0    young
1    young
2      old
3      old
Name: age, dtype: object
```

In [17]:
```
# MZ: or using lambda function
compo_pd.age.apply(lambda x: x**2)
```

Out[17]:
```
0    2601
1    3249
2    4356
3    4761
Name: age, dtype: int64
```

And again, if we want, we can directly use this output to create a new column:

In [18]:
```
compo_pd['age_def'] = compo_pd.age.apply(define_age)
compo_pd
# MZ: NB: you can also apply functions to rows of the dataframe
# can be useful to create categorical variables
```

Out[18]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

We can also apply a function to an entire DataFrame. For example we can ask how many composers have birth and death dates within the XIXth century:

In [19]:
```
def nineteen_century_count(x):
    return np.sum((x>=1800)&(x<1900))
#def nineteen_century_count2(x):
#    return np.sum((x>=1800)and(x<1900)) # does not work !!
```

In [20]:
```
5 < 10 and 5 < 6
```

Out[20]:
```
True
```

In [21]:
```
compo_pd[['birth','death']].apply(nineteen_century_count)
#compo_pd[['birth','death']].apply(nineteen_century_count2)
```

Out[21]:
```
birth    2
death    1
dtype: int64
```

The function is applied column-wise and returns a single number for each in the form of a series.

In [22]:
```
def nineteen_century_true(x):
    return (x>=1800)&(x<1900)
```

In [23]:
```
compo_pd[['birth','death']].apply(nineteen_century_true)
```

Out[23]:

	birth	death
0	True	False
1	False	True
2	True	False
3	False	False

Here the operation is again applied column-wise but the output is a Series.

There are more combinations of what can be the in- and output of the apply function and in what order (column- or row-wise) they are applied that cannot be covered here.

3.5 Logical indexing

Just like with Numpy, it is possible to subselect parts of a Dataframe using logical indexing. Let's have a look again at an example:

In [24]:
```
compo_pd
```

Out[24]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

If we use a logical comprison on a series, this yields a **logical Series**:

In [25]:
```
compo_pd['birth'] > 1859
```

Out[25]:
```
0     True
1    False
2    False
3     True
Name: birth, dtype: bool
```

In [26]:
```
log_indexer = compo_pd['birth'] > 1859
log_indexer
compo_pd[log_indexer]
# MZ: select the rows based on logical indexing
# MZ: to negate the logicals
compo_pd[~log_indexer]
# ! again here not is not working !
# compo_pd[not log_indexer] # ERROR !
```

Out[26]:

	composer	birth	death	city	age	age_def
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old

Just like in Numpy we can use this logical Series as an index to select elements in the Dataframe:

In [27]:
```
compo_pd[compo_pd['birth'] > 1859]
```

Out[27]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

We can also create more complex logical indexings:

In [28]:
```
compo_pd[(compo_pd['birth'] > 1859)&(compo_pd['age']>60)]
```

Out[28]:

	composer	birth	death	city	age	age_def
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

And we can create new arrays containing only these subselections:

In [29]:
```
compos_sub = compo_pd[compo_pd['birth'] > 1859]
# MZ how the 2 are connected ??? tricky to know if Pandas create a copy
or not
# best to explicitly create a copy !
```

We can then modify the new array:

In [30]:
```
compos_sub.loc[0,'birth'] = 3000
# warning to tell that something might go wrong (because not used copy)
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/indexing.py:543: Setti
ngWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-doc
s/stable/indexing.html#indexing-view-versus-copy
  self.obj[item] = s
```

Note that we get this SettingWithCopyWarning warning. This is a very common problem hand has to do with how new arrays are created when making subselections. Simply stated, did we create an entirely new array or a "view" of the old one? This will be very case-dependent and to avoid this, if we want to create a new array we can just enforce it using the `copy()` method (for more information on the topic see for example this explanation (https://www.dataquest.io/blog/settingwithcopywarning/):

In [31]:
```
# MZ: better to explicitly create a copy
compos_sub2 = compo_pd[compo_pd['birth'] > 1859].copy()
compos_sub2.loc[0,'birth'] = 3000
```

04-Combining information in Pandas

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Often information is comming from different sources and it is necessary to combine it into one object. We are going to see the different ways in which information contained within separate Dataframes can be combined in a meaningful way.

4.1 Concatenation

The simplest way we can combine two Dataframes is simply to "paste" them together:

In [2]:
```
composers1 = pd.read_excel('Datasets/composers.xlsx', index_col='compose
r',sheet_name='Sheet1')
composers1
```

Out[2]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

In [3]:
```
composers2 = pd.read_excel('Datasets/composers.xlsx', index_col='compose
r',sheet_name='Sheet3')
composers2
```

Out[3]:

composer	birth	death	city
Verdi	1813	1901	Roncole
Dvorak	1841	1904	Nelahozeves
Schumann	1810	1856	Zwickau
Stravinsky	1882	1971	Oranienbaum
Mahler	1860	1911	Kaliste

To be concatenated, Dataframes need to be provided as a list:

In [4]:
```
all_composers = pd.concat([composers1,composers2])
# columns have the same -> ok
```

In [5]:
```
all_composers
```

Out[5]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg
Verdi	1813	1901	Roncole
Dvorak	1841	1904	Nelahozeves
Schumann	1810	1856	Zwickau
Stravinsky	1882	1971	Oranienbaum
Mahler	1860	1911	Kaliste

One potential problem is that two tables contain duplicated information:

In [6]:
```
# be careful if same index !! can have twice the same index !
# ! ensure to not have the same index
all_composers.loc['Mahler']
```

Out[6]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Mahler	1860	1911	Kaliste

It is very easy to get rid of it using:

In [7]:
```
all_composers.drop_duplicates()
# suppress duplicates from the table
```

Out[7]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg
Verdi	1813	1901	Roncole
Dvorak	1841	1904	Nelahozeves
Schumann	1810	1856	Zwickau
Stravinsky	1882	1971	Oranienbaum

4.2 Joining two tables

An other classical case is that of two list with similar index but containing different information, e.g.

In [8]:
```
composers1 = pd.read_excel('Datasets/composers.xlsx', index_col='compose
r',sheet_name='Sheet1')
composers1
```

Out[8]:

composer	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

In [9]:
```
composers2 = pd.read_excel('Datasets/composers.xlsx', index_col='compose
r',sheet_name='Sheet4')
composers2
```

Out[9]:

composer	first name
Mahler	Gustav
Beethoven	Ludwig van
Puccini	Giacomo
Brahms	Johannes

If we we use again simple concatenation, this doesn't help us much. We just end up with a large matrix with lots of NaN's:

In [10]:
```
pd.concat([composers1, composers2])
```

```
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:1: FutureWar
ning: Sorting because non-concatenation axis is not aligned. A future ver
sion
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.

  """Entry point for launching an IPython kernel.
```

Out[10]:

	birth	city	death	first name
composer				
Mahler	1860.0	Kaliste	1911.0	NaN
Beethoven	1770.0	Bonn	1827.0	NaN
Puccini	1858.0	Lucques	1924.0	NaN
Shostakovich	1906.0	Saint-Petersburg	1975.0	NaN
Mahler	NaN	NaN	NaN	Gustav
Beethoven	NaN	NaN	NaN	Ludwig van
Puccini	NaN	NaN	NaN	Giacomo
Brahms	NaN	NaN	NaN	Johannes

The better way of doing this is to **join** the tables. This is a classical database concept avaialble in Pandas.

`join()` operates on two tables: the first one is the "left" table which uses `join()` as a method. The other table is the "right" one.

Let's try the default join settings:

In [11]:
```
composers1.join(composers2)
# get all elements of the 1st table, merged with the 2nd table
# everything based on the left table (what from the 2nd table and is not
in the 1st table is dropped)
# by default is left based
```

Out[11]:

	birth	death	city	first name
composer				
Mahler	1860	1911	Kaliste	Gustav
Beethoven	1770	1827	Bonn	Ludwig van
Puccini	1858	1924	Lucques	Giacomo
Shostakovich	1906	1975	Saint-Petersburg	NaN

We see that Pandas was smart enough to notice that the two tables had a index name and used it to combine the tables. We also see that one element from the second table (Brahms) is missing. The reason for this is the way indices not present in both tables are handled. There are four ways of doing this with two tables called here the "left" and "right" table.

4.2.1. Join left

Here "left" and "right" just represent two Dataframes that should be merged. They have a common index, but not necessarily the same items. For example here Shostakovich is missing in the second table, while Brahms is missing in the first one. When using the "right" join, we use the first Dataframe as basis and only use the indices that appear there.

In [12]:
```
composers1.join(composers2, how = 'left') # this is the default
```

Out[12]:

	birth	death	city	first name
composer				
Mahler	1860	1911	Kaliste	Gustav
Beethoven	1770	1827	Bonn	Ludwig van
Puccini	1858	1924	Lucques	Giacomo
Shostakovich	1906	1975	Saint-Petersburg	NaN

Hence Brahms is left out.

4.2.2. Join right

We can do the the opposite and use the indices of the second Dataframe as basis:

In [13]:
```
composers1.join(composers2, how = 'right')
```

Out[13]:

	birth	death	city	first name
composer				
Mahler	1860.0	1911.0	Kaliste	Gustav
Beethoven	1770.0	1827.0	Bonn	Ludwig van
Puccini	1858.0	1924.0	Lucques	Giacomo
Brahms	NaN	NaN	NaN	Johannes

Here we have Brahms but not Shostakovich.

4.2.3. Inner, outer

Finally, we can just say that we want to recover eihter only the items that appaer in both Dataframes (inner, like in a Venn diagram) or all the items (outer).

In [14]:
```
composers1.join(composers2, how = 'inner')
# => take all indices of both tables
```

Out[14]:

	birth	death	city	first name
composer				
Mahler	1860	1911	Kaliste	Gustav
Beethoven	1770	1827	Bonn	Ludwig van
Puccini	1858	1924	Lucques	Giacomo

In [15]:
```
composers1.join(composers2, how = 'outer')
```

Out[15]:

	birth	death	city	first name
composer				
Beethoven	1770.0	1827.0	Bonn	Ludwig van
Brahms	NaN	NaN	NaN	Johannes
Mahler	1860.0	1911.0	Kaliste	Gustav
Puccini	1858.0	1924.0	Lucques	Giacomo
Shostakovich	1906.0	1975.0	Saint-Petersburg	NaN

4.3.4 Joining on columns : merge

Above we have used `join` to join based on indices. However sometimes tables don't have the same indices but similar contents that we want to merge. For example let's imagine whe have the two Dataframes below:

In [16]:
```
composers1 = pd.read_excel('Datasets/composers.xlsx', sheet_name='Sheet1
')
composers2 = pd.read_excel('Datasets/composers.xlsx', sheet_name='Sheet6
')
```

In [17]:
```
composers1
```

Out[17]:

	composer	birth	death	city
0	Mahler	1860	1911	Kaliste
1	Beethoven	1770	1827	Bonn
2	Puccini	1858	1924	Lucques
3	Shostakovich	1906	1975	Saint-Petersburg

In [18]:
```
composers2
```

Out[18]:

	last name	first name
0	Puccini	Giacomo
1	Beethoven	Ludwig van
2	Brahms	Johannes
3	Mahler	Gustav

The indices don't match and are not the composer name. In addition the columns containing the composer names have different labels. Here we can use `merge()` and specify which columns we want to use for merging, and what type of merging we need (inner, left etc.)

In [19]:
```
# take left and right tables and can specify which column from each to p
erform the merge
pd.merge(composers1, composers2, left_on='composer', right_on='last name
')
```

Out[19]:

	composer	birth	death	city	last name	first name
0	Mahler	1860	1911	Kaliste	Mahler	Gustav
1	Beethoven	1770	1827	Bonn	Beethoven	Ludwig van
2	Puccini	1858	1924	Lucques	Puccini	Giacomo

Again we can use another variety of join than the default inner:

In [20]:
```
pd.merge(composers1, composers2, left_on='composer', right_on='last name
',how = 'outer')
```

Out[20]:

	composer	birth	death	city	last name	first name
0	Mahler	1860.0	1911.0	Kaliste	Mahler	Gustav
1	Beethoven	1770.0	1827.0	Bonn	Beethoven	Ludwig van
2	Puccini	1858.0	1924.0	Lucques	Puccini	Giacomo
3	Shostakovich	1906.0	1975.0	Saint-Petersburg	NaN	NaN
4	NaN	NaN	NaN	NaN	Brahms	Johannes

In [21]:
```
pd.merge(composers1, composers2, left_on='composer', right_on='last name
',how = 'right')
```

Out[21]:

	composer	birth	death	city	last name	first name
0	Mahler	1860.0	1911.0	Kaliste	Mahler	Gustav
1	Beethoven	1770.0	1827.0	Bonn	Beethoven	Ludwig van
2	Puccini	1858.0	1924.0	Lucques	Puccini	Giacomo
3	NaN	NaN	NaN	NaN	Brahms	Johannes

In [22]:
```
# MZ: to remove the column that now contains duplicated information:
dt1 = pd.merge(composers1, composers2, left_on='composer', right_on='las
t name')
dt1
# default is to drop rows, to drop columns set 1st axis
dt1.drop('last name', axis=1)
dt1
dt1.drop('last name', axis=1, inplace=True)
dt1
```

Out[22]:

	composer	birth	death	city	first name
0	Mahler	1860	1911	Kaliste	Gustav
1	Beethoven	1770	1827	Bonn	Ludwig van
2	Puccini	1858	1924	Lucques	Giacomo

5.Splitting data

Often one has tables that mix regular variables (e.g. the size of cells in microscopy images) with categorical variables (e.g. the type of cell to which they belong). In that case, it is quite usual to split the data using the category to do computations. Pandas allows to do this very easily.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

5.1 Grouping

Let's import some data and have a look at them

In [2]:
```
composers = pd.read_excel('Datasets/composers.xlsx', sheet_name='Sheet5
')
```

In [3]:
```
composers.head()
```

Out[3]:

	composer	birth	death	period	country
0	Mahler	1860	1911.0	post-romantic	Austria
1	Beethoven	1770	1827.0	romantic	Germany
2	Puccini	1858	1924.0	post-romantic	Italy
3	Shostakovich	1906	1975.0	modern	Russia
4	Verdi	1813	1901.0	romantic	Italy

In [4]:
```
# MZ
# you don't have to explicitly go through the table and groupe elements
# simply use the 'groupby' function
```

5.1.1 Single level

What if we want now to count how many composers we have in each category? In classical computing we would maybe do a for loop to count occurrences. Pandas simplifies this with the `groupby()` function, which actually groups elements by a certain criteria, e.g. a categorical variable like the period:

In [5]:
```
composer_grouped = composers.groupby('period')
composer_grouped
# MZ: create new type of object from Pandas
```

Out[5]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f5ff3cfbf60>
```

The output is a bit cryptic. What we actually have is a new object called a group which has a lot of handy properties. First let's see what the groups actually are. As for the Dataframe, let's look at a summary of the object:

In [6]:
```
composer_grouped.describe()
# MZ: get all the statistics by the groups created

# MZ: for example to see the different levels
composers.country.unique()
```

Out[6]:
```
array(['Austria', 'Germany', 'Italy', 'Russia', 'Czechia', 'Finland',
       'France', 'RUssia', 'England', 'Belgium', 'Spain', 'USA'],
      dtype=object)
```

So we have a dataframe with a statistical summary of the the contents. The "names" of the groups are here the indices of the Dataframe. These names are simply all the different categories that were present in the column we used for grouping. Now we can recover a single group:

In [7]:
```
composer_grouped.get_group('baroque')
```

Out[7]:

	composer	birth	death	period	country
14	Haendel	1685	1759.0	baroque	Germany
16	Purcell	1659	1695.0	baroque	England
17	Charpentier	1643	1704.0	baroque	France
20	Couperin	1626	1661.0	baroque	France
21	Rameau	1683	1764.0	baroque	France
28	Caldara	1670	1736.0	baroque	Italy
29	Pergolesi	1710	1736.0	baroque	Italy
30	Scarlatti	1685	1757.0	baroque	Italy
31	Caccini	1587	1640.0	baroque	Italy
47	Bach	1685	1750.0	baroque	Germany

In [8]:
```
composer_grouped.get_group('post-romantic')
```

Out[8]:

	composer	birth	death	period	country
0	Mahler	1860	1911.0	post-romantic	Austria
2	Puccini	1858	1924.0	post-romantic	Italy
8	Sibelius	1865	1957.0	post-romantic	Finland
18	Bruckner	1824	1896.0	post-romantic	Austria
49	Strauss	1864	1949.0	post-romantic	Germany

5.2.2 Multi-level

If one has multiple categorical variables, one can also do a grouping on several levels. For example here we want to classify composers both by period and country. For this we just give two column names to the `groupby()` function:

In [9]:
```
# MZ: groupping can be done on multiple columns
composer_grouped = composers.groupby(['period','country'])
composer_grouped.describe()
```

Out[9]:

		birth							
		count	mean	std	min	25%	50%	75%	max
period	country								
baroque	England	1.0	1659.000000	NaN	1659.0	1659.00	1659.0	1659.00	1659.0
	France	3.0	1650.666667	29.263174	1626.0	1634.50	1643.0	1663.00	1683.0
	Germany	2.0	1685.000000	0.000000	1685.0	1685.00	1685.0	1685.00	1685.0
	Italy	4.0	1663.000000	53.285395	1587.0	1649.25	1677.5	1691.25	1710.0
classic	Austria	2.0	1744.000000	16.970563	1732.0	1738.00	1744.0	1750.00	1756.0
	Czechia	1.0	1731.000000	NaN	1731.0	1731.00	1731.0	1731.00	1731.0
	Italy	1.0	1749.000000	NaN	1749.0	1749.00	1749.0	1749.00	1749.0
	Spain	1.0	1754.000000	NaN	1754.0	1754.00	1754.0	1754.00	1754.0
modern	Austria	1.0	1885.000000	NaN	1885.0	1885.00	1885.0	1885.00	1885.0
	Czechia	1.0	1854.000000	NaN	1854.0	1854.00	1854.0	1854.00	1854.0
	England	2.0	1936.500000	48.790368	1902.0	1919.25	1936.5	1953.75	1971.0
	France	2.0	1916.500000	12.020815	1908.0	1912.25	1916.5	1920.75	1925.0
	Germany	1.0	1895.000000	NaN	1895.0	1895.00	1895.0	1895.00	1895.0
	RUssia	1.0	1891.000000	NaN	1891.0	1891.00	1891.0	1891.00	1891.0
	Russia	2.0	1894.000000	16.970563	1882.0	1888.00	1894.0	1900.00	1906.0
	USA	3.0	1918.333333	18.502252	1900.0	1909.00	1918.0	1927.50	1937.0
post-romantic	Austria	2.0	1842.000000	25.455844	1824.0	1833.00	1842.0	1851.00	1860.0
	Finland	1.0	1865.000000	NaN	1865.0	1865.00	1865.0	1865.00	1865.0
	Germany	1.0	1864.000000	NaN	1864.0	1864.00	1864.0	1864.00	1864.0
	Italy	1.0	1858.000000	NaN	1858.0	1858.00	1858.0	1858.00	1858.0
renaissance	Belgium	2.0	1464.500000	95.459415	1397.0	1430.75	1464.5	1498.25	1532.0
	England	2.0	1551.500000	16.263456	1540.0	1545.75	1551.5	1557.25	1563.0
	Italy	3.0	1552.666667	23.965253	1525.0	1545.50	1566.0	1566.50	1567.0
romantic	Czechia	2.0	1832.500000	12.020815	1824.0	1828.25	1832.5	1836.75	1841.0
	France	3.0	1821.000000	19.672316	1803.0	1810.50	1818.0	1830.00	1842.0
	Germany	4.0	1806.500000	26.388129	1770.0	1800.00	1811.5	1818.00	1833.0
	Italy	4.0	1817.250000	28.004464	1797.0	1800.00	1807.0	1824.25	1858.0
	Russia	2.0	1836.000000	4.242641	1833.0	1834.50	1836.0	1837.50	1839.0
	Spain	2.0	1863.500000	4.949747	1860.0	1861.75	1863.5	1865.25	1867.0

In [10]:
```
composer_grouped.get_group(('baroque','Germany'))
```

Out[10]:

	composer	birth	death	period	country
14	Haendel	1685	1759.0	baroque	Germany
47	Bach	1685	1750.0	baroque	Germany

5.2 Operations on groups

The main advantage of this Group object is that it allows us to do very quickly both computations and plotting without having to loop through different categories. Indeed Pandas makes all the work for us: it applies functions on each group and then reassembles the results into a Dataframe (or Series depending on output).

For example we can apply most functions we used for Dataframes (mean, sum etc.) on groups as well and Pandas seamlessly does the work for us:

In [11]:
```
composer_grouped.mean()
# MZ: often you can directly apply the functions on the Pandas object
```

Out[11]:

		birth	death
period	country		
baroque	England	1659.000000	1695.000000
	France	1650.666667	1709.666667
	Germany	1685.000000	1754.500000
	Italy	1663.000000	1717.250000
classic	Austria	1744.000000	1800.000000
	Czechia	1731.000000	1799.000000
	Italy	1749.000000	1801.000000
	Spain	1754.000000	1806.000000
modern	Austria	1885.000000	1935.000000
	Czechia	1854.000000	1928.000000
	England	1936.500000	1983.000000
	France	1916.500000	2004.000000
	Germany	1895.000000	1982.000000
	RUssia	1891.000000	1953.000000
	Russia	1894.000000	1973.000000
	USA	1918.333333	1990.000000
post-romantic	Austria	1842.000000	1903.500000
	Finland	1865.000000	1957.000000
	Germany	1864.000000	1949.000000
	Italy	1858.000000	1924.000000
renaissance	Belgium	1464.500000	1534.000000
	England	1551.500000	1624.500000
	Italy	1552.666667	1616.666667
romantic	Czechia	1832.500000	1894.000000
	France	1821.000000	1891.333333
	Germany	1806.500000	1865.750000
	Italy	1817.250000	1875.750000
	Russia	1836.000000	1884.000000
	Spain	1863.500000	1912.500000

In [12]:
```
composer_grouped.count()
```

Out[12]:

		composer	birth	death
period	**country**			
baroque	**England**	1	1	1
	France	3	3	3
	Germany	2	2	2
	Italy	4	4	4
classic	**Austria**	2	2	2
	Czechia	1	1	1
	Italy	1	1	1
	Spain	1	1	1
modern	**Austria**	1	1	1
	Czechia	1	1	1
	England	2	2	1
	France	2	2	2
	Germany	1	1	1
	RUssia	1	1	1
	Russia	2	2	2
	USA	3	3	2
post-romantic	**Austria**	2	2	2
	Finland	1	1	1
	Germany	1	1	1
	Italy	1	1	1
renaissance	**Belgium**	2	2	2
	England	2	2	2
	Italy	3	3	3
romantic	**Czechia**	2	2	2
	France	3	3	3
	Germany	4	4	4
	Italy	4	4	4
	Russia	2	2	2
	Spain	2	2	2

We can also design specific functions (again, like in the case of Dataframes) and apply them on groups:

In [13]:
```
def mult(ser):
    return ser.max() * 3
```

In [14]:
```
composer_grouped.apply(mult)
# MZ: most functions can be applied irrespectively of the object (DataFr
ame, group, Series, etc.)
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/computation/check.py:1
9: UserWarning: The installed version of numexpr 2.4.3 is not supported i
n pandas and will be not be used
The minimum supported version is 2.6.1

  ver=ver, min_ver=_MIN_NUMEXPR_VERSION), UserWarning)
```

Out[14]:

		composer	birth	death	period
period	**country**				
baroque	**England**	PurcellPurcellPurcell	4977	5085.0	baroquebaroquebaroq
	France	RameauRameauRameau	5049	5292.0	baroquebaroquebaroq
	Germany	HaendelHaendelHaendel	5055	5277.0	baroquebaroquebaroq
	Italy	ScarlattiScarlattiScarlatti	5130	5271.0	baroquebaroquebaroq
classic	**Austria**	MozartMozartMozart	5268	5427.0	classicclassicclassic
	Czechia	DusekDusekDusek	5193	5397.0	classicclassicclassic
	Italy	CimarosaCimarosaCimarosa	5247	5403.0	classicclassicclassic
	Spain	SolerSolerSoler	5262	5418.0	classicclassicclassic
modern	**Austria**	BergBergBerg	5655	5805.0	modernmodernmoderr
	Czechia	JanacekJanacekJanacek	5562	5784.0	modernmodernmoderr
	England	WaltonWaltonWalton	5913	5949.0	modernmodernmoderr
	France	MessiaenMessiaenMessiaen	5775	6048.0	modernmodernmoderr
	Germany	OrffOrffOrff	5685	5946.0	modernmodernmoderr
	RUssia	ProkofievProkofievProkofiev	5673	5859.0	modernmodernmoderr
	Russia	StravinskyStravinskyStravinsky	5718	5925.0	modernmodernmoderr
	USA	GlassGlassGlass	5811	5970.0	modernmodernmoderr
post-romantic	**Austria**	MahlerMahlerMahler	5580	5733.0	post-romanticpost-rom romantic
	Finland	SibeliusSibeliusSibelius	5595	5871.0	post-romanticpost-rom romantic
	Germany	StraussStraussStrauss	5592	5847.0	post-romanticpost-rom romantic
	Italy	PucciniPucciniPuccini	5574	5772.0	post-romanticpost-rom romantic
renaissance	**Belgium**	LassusLassusLassus	4596	4782.0	renaissancerenaissanc
	England	DowlandDowlandDowland	4689	4878.0	renaissancerenaissanc
	Italy	PalestrinaPalestrinaPalestrina	4701	4929.0	renaissancerenaissanc
romantic	**Czechia**	SmetanaSmetanaSmetana	5523	5712.0	romanticromanticroma
	France	MassenetMassenetMassenet	5526	5736.0	romanticromanticroma
	Germany	WagnerWagnerWagner	5499	5691.0	romanticromanticroma
	Italy	VerdiVerdiVerdi	5574	5757.0	romanticromanticroma
	Russia	MussorsgskyMussorsgskyMussorsgsky	5517	5661.0	romanticromanticroma
	Spain	GranadosGranadosGranados	5601	5748.0	romanticromanticroma

5.3 Unstacking

Let's have a look again at one of our grouped Dataframe on which we applied some summary function like a mean on the age column:

In [15]:
```
composers['age']= composers['death']-composers['birth']
```

In [16]:
```
composers.groupby(['country','period']).age.mean()
```

Out[16]:
```
country  period
Austria  classic          56.000000
         modern           50.000000
         post-romantic    61.500000
Belgium  renaissance      69.500000
Czechia  classic          68.000000
         modern           74.000000
         romantic         61.500000
England  baroque          36.000000
         modern           81.000000
         renaissance      73.000000
Finland  post-romantic    92.000000
France   baroque          59.000000
         modern           87.500000
         romantic         70.333333
Germany  baroque          69.500000
         modern           87.000000
         post-romantic    85.000000
         romantic         59.250000
Italy    baroque          54.250000
         classic          52.000000
         post-romantic    66.000000
         renaissance      64.000000
         romantic         58.500000
RUssia   modern           62.000000
Russia   modern           79.000000
         romantic         48.000000
Spain    classic          52.000000
         romantic         49.000000
USA      modern           81.000000
Name: age, dtype: float64
```

Here we have two level of indices, with the main one being the country which contains all periods. Often for plotting we however need to have the information in another format. In particular we would like each of these values to be one observation in a regular table. For example we could have a country vs period table where all elements are the mean age. To do that we need to **unstack** our multi-level Dataframe:

In [17]:
```
# MZ: to obtain regular 2dim object
composer_unstacked = composers.groupby(['country','period']).age.mean().
unstack()
```

In [18]:
```
composer_unstacked
```

Out[18]:

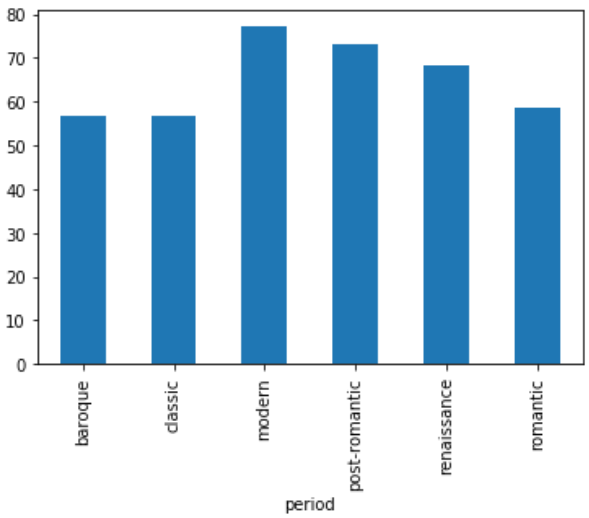
period	baroque	classic	modern	post-romantic	renaissance	romantic
country						
Austria	NaN	56.0	50.0	61.5	NaN	NaN
Belgium	NaN	NaN	NaN	NaN	69.5	NaN
Czechia	NaN	68.0	74.0	NaN	NaN	61.500000
England	36.00	NaN	81.0	NaN	73.0	NaN
Finland	NaN	NaN	NaN	92.0	NaN	NaN
France	59.00	NaN	87.5	NaN	NaN	70.333333
Germany	69.50	NaN	87.0	85.0	NaN	59.250000
Italy	54.25	52.0	NaN	66.0	64.0	58.500000
RUssia	NaN	NaN	62.0	NaN	NaN	NaN
Russia	NaN	NaN	79.0	NaN	NaN	48.000000
Spain	NaN	52.0	NaN	NaN	NaN	49.000000
USA	NaN	NaN	81.0	NaN	NaN	NaN

5.4 Plotting groups

The possibility to create groups gives us also the opportunity to easily create interesting plots without writing too much code. For example we can caluclate the average age of composers in each period and plot it as a bar plot:

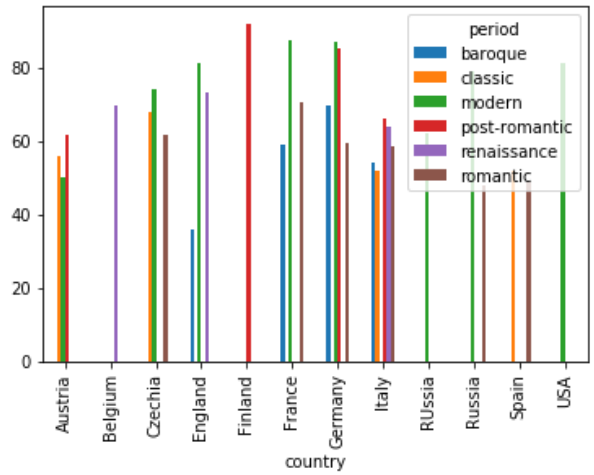
In [19]:
```
composers.groupby('period')['age'].mean().plot(kind = 'bar')
# MZ: group by period and plot the mean of the ages
```

Out[19]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6024431278>
```



We can also use our unstacked table of country vs. period to automatically plot all average ages split by country and period:

In [20]:
```
composer_unstacked.plot(kind = 'bar');
# average age for each country and each period
```



There are much more powerful ways of using grouping-like features for plotting using the ggplot type grammar of graphics where objects can be grouped within an "aeasthetic". In the example above the "colour aesthetic" would e.g. be assigned to the period variable. Such an approach removes the need to do explicit groupings as done here.

6. Advanced plotting

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import seaborn as sns
```

We have seen already two options to plot data: we can use the "raw" Matplotlib which in principle allows one to create any possible plot, however with lots of code, and we saw the simpler internal Pandas solution. While the latter solution is very practical to quickly look through data, it is rather cumbersome to realise more complex plots.

Here we look at another type of plotting resting on the concepts of the grammar of graphics. This approach allows to create complex plots where data can be simply split in a plot into color, shapes etc. without having to do a grouping operation in beforehand. We will mainly look at Seaborn, and finish with an example with Plotnine, the port to Python of ggplot.

Importing data

We come back here to the dataset of swiss towns. To make the dataset more interestig we add to it some categorical data. First we attempt to add the main language for each town. It is a good example of the type of data wranglig one ofen has to do by combining information from different sources.

In [2]:
```
#load table indicating to which canton each town belongs
cantons = pd.read_excel('Datasets/be-b-00.04-osv-01.xls',sheet_name=
1)[['KTKZ','ORTNAME']]
```

In [3]:
```
#load general table with infos on towns
towns = pd.read_excel('Datasets/2018.xls', skiprows=list(range(5))+list
(range(6,9)),
                      skipfooter=34, index_col='Commune',na_values=['*
','X'])
towns = towns.reset_index()
```

In [4]:
```
#merge tables using the town name. This adds the canton abbreviation to
the main table
towns_canton = pd.merge(towns, cantons, left_on='Commune', right_on='ORT
NAME',how = 'inner')
```

In [5]:
```
#load data indicating languages of each canton
language = pd.read_excel('Datasets/je-f-01.08.01.02.xlsx',skiprows=[0,2,
3,4],skipfooter=11)
languages = language[['Allemand (ou suisse allemand)','Français (ou pato
is romand)',
         'Italien (ou dialecte tessinois/italien des grisons)']]
languages = languages.apply(pd.to_numeric, errors='coerce')
#check which language has majority in each canton
languages['language'] = np.argmax(languages.values.astype(float),axis=1)
code={0:'German', 1:'French', 2:'Italian'}
languages['Language'] = languages.language.apply(lambda x: code[x])
languages['canton'] = language['Unnamed: 0']
languages = languages[['canton','Language']]

#load table matching canton name to abbreviation
cantons_abbrev = pd.read_excel('Datasets/cantons_abbrev.xlsx')
#add full canton name to table by merging on abbreviation
canton_language = pd.merge(languages, cantons_abbrev,on='canton')
```

In [6]:
```
#add language by merging on canton abbreviation
towns_language = pd.merge(towns_canton, canton_language, left_on='KTKZ',
right_on='abbrev')
```

In [7]:
```
towns_language['town_type'] = towns_language['Surface agricole en %'].ap
ply(lambda x: 'Land' if x<50 else 'City')
```

In [8]:
```
#Create a new party column and a new party score column
parties = pd.melt(towns_language,id_vars=['Commune'], value_vars=['UDC
','PS','PDC'],
                  var_name= 'Party', value_name='Party score')
towns_language = pd.merge(parties, towns_language, on='Commune')

towns_language
```

Out[8]:

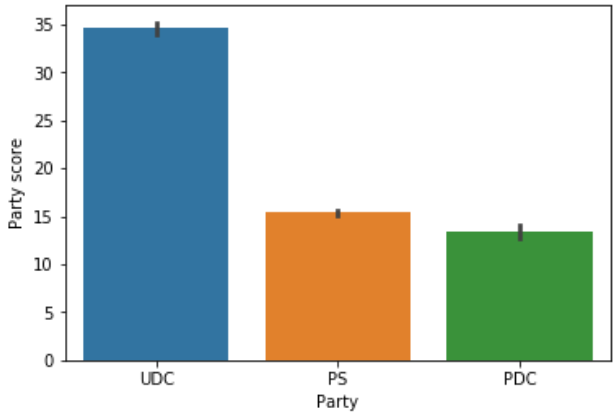
	Commune	Party	Party score	Code commune	Habitants	Variation en %	Densité de la population par km²	Etrangers en %
0	Aeugst am Albis	UDC	30.929249	1	1977	8.388158	249.936789	13.100658
1	Aeugst am Albis	PS	18.645940	1	1977	8.388158	249.936789	13.100658
2	Aeugst am Albis	PDC	2.076428	1	1977	8.388158	249.936789	13.100658
3	Affoltern am Albis	UDC	33.785785	2	11900	7.294203	1123.701605	27.848740
4	Affoltern am Albis	PS	19.080314	2	11900	7.294203	1123.701605	27.848740
5	Affoltern am Albis	PDC	4.585387	2	11900	7.294203	1123.701605	27.848740
6	Bonstetten	UDC	29.100156	3	5435	5.349874	731.493943	14.149034
7	Bonstetten	PS	20.403265	3	5435	5.349874	731.493943	14.149034
8	Bonstetten	PDC	3.378541	3	5435	5.349874	731.493943	14.149034
9	Hausen am Albis	UDC	34.937369	4	3571	6.279762	262.573529	14.533744
10	Hausen am Albis	PS	19.393305	4	3571	6.279762	262.573529	14.533744
11	Hausen am Albis	PDC	2.881915	4	3571	6.279762	262.573529	14.533744
12	Hedingen	UDC	30.114599	5	3687	8.123167	564.624809	14.971522
13	Hedingen	PS	22.478008	5	3687	8.123167	564.624809	14.971522
14	Hedingen	PDC	3.918166	5	3687	8.123167	564.624809	14.971522
15	Kappel am Albis	UDC	48.615099	6	1110	20.915033	140.151515	18.018018
16	Kappel am Albis	PS	10.285425	6	1110	20.915033	140.151515	18.018018
17	Kappel am Albis	PDC	2.744469	6	1110	20.915033	140.151515	18.018018
18	Knonau	UDC	32.876136	7	2168	20.444444	335.085008	17.158672
19	Knonau	PS	18.436553	7	2168	20.444444	335.085008	17.158672
20	Knonau	PDC	3.126052	7	2168	20.444444	335.085008	17.158672
21	Maschwanden	UDC	43.383446	8	626	1.623377	133.475480	12.140575
22	Maschwanden	PS	22.732529	8	626	1.623377	133.475480	12.140575
23	Maschwanden	PDC	3.502396	8	626	1.623377	133.475480	12.140575
24	Mettmenstetten	UDC	35.671015	9	4861	14.565166	373.062164	14.873483
25	Mettmenstetten	PS	18.800282	9	4861	14.565166	373.062164	14.873483
26	Mettmenstetten	PDC	3.649155	9	4861	14.565166	373.062164	14.873483
27	Obfelden	UDC	36.174029	10	5131	9.496372	680.503979	20.015591

Basic plotting

We finally have a table with mostly numerical information but also two categorical data: language and town type (land or city). With Seaborn we can now easily make all sorts of plots. For example what are the average scores of the different parties:

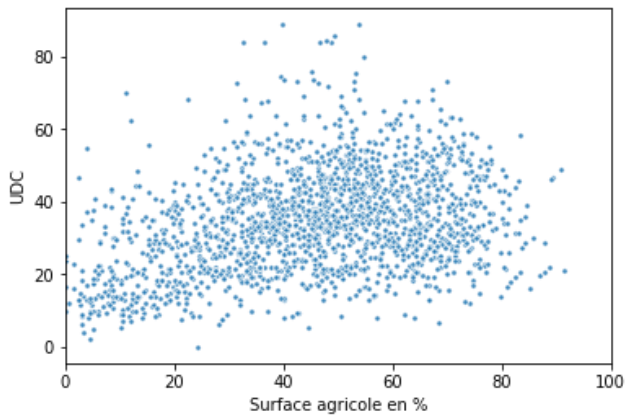
In [9]:
```
sns.barplot(data = towns_language, y='Party score', x = 'Party');
```

```
/usr/local/lib/python3.5/dist-packages/scipy/stats/stats.py:1713: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is depre
cated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this wi
ll be interpreted as an array index, `arr[np.array(seq)]`, which will res
ult either in an error or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



Do land towns vote more for the right-wing party ?

In [10]:
```
g = sns.scatterplot(data = towns_language, y='UDC', x = 'Surface agricol
e en %', s = 10, alpha = 0.5);
g.set_xlim([0,100]);
```

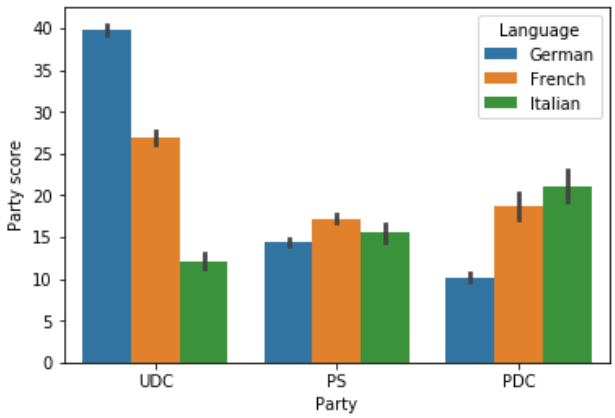


Using categories as "aesthetics"

The greate advantage of using these packages is that they allow to include categories as "aesthetics" of the plot. For example we looked before at average party scores. But are they different between language regions ? We can just specify that the hue (color) should be mapped to the town language:

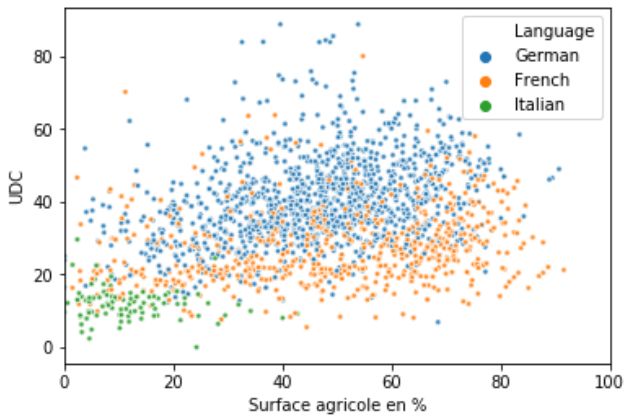
In [11]:
```
sns.barplot(data = towns_language, y='Party score', x = 'Party', hue = '
Language');
```

```
/usr/local/lib/python3.5/dist-packages/scipy/stats/stats.py:1713: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is depre
cated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this wi
ll be interpreted as an array index, `arr[np.array(seq)]`, which will res
ult either in an error or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



Similarly with scatter plots. Is the relation between land and voting on the right language dependent ?

In [12]:
```
g = sns.scatterplot(data = towns_language, y='UDC', x = 'Surface agricol
e en %', hue = 'Language',
                    s = 10, alpha = 0.5);
g.set_xlim([0,100]);
```

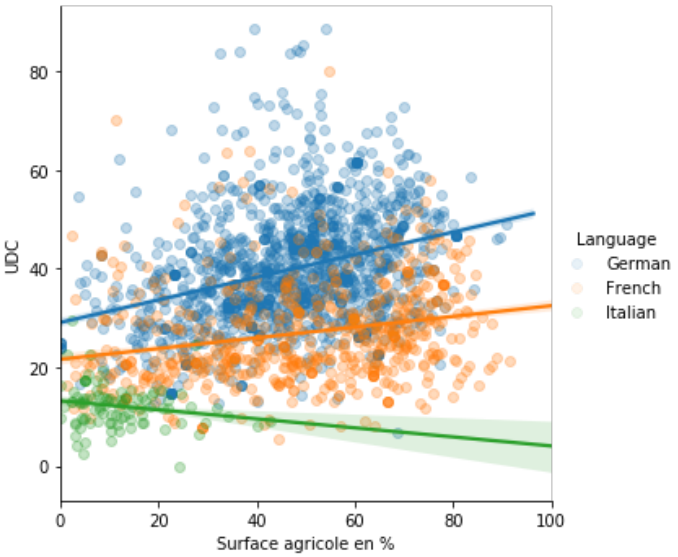


Statistics

We see difference in the last plot, but it is still to clearly see the relation. Luckiliy these packages allow us to either create summary statistics or to fit the data:

In [13]:
```
g = sns.lmplot(data = towns_language, x = 'Surface agricole en %', y='UDC', hue = 'Language', scatter=True,
               scatter_kws={'alpha': 0.1});
g.ax.set_xlim([0,100]);
```

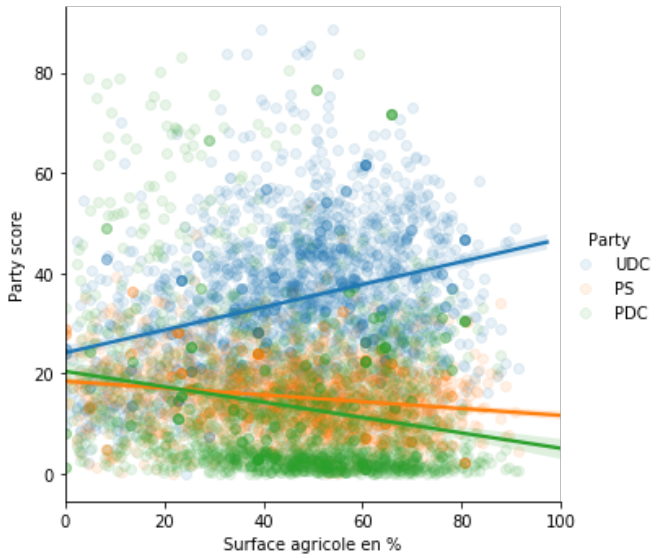
```
/usr/local/lib/python3.5/dist-packages/scipy/stats/stats.py:1713: FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an array index, `arr[np.array(seq)]`, which will result either in an error or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



Now we can also do the same exercise for all parties. Does the relation hold?

In [14]:
```
g = sns.lmplot(data = towns_language, x = 'Surface agricole en %', y='Party score',
               hue = 'Party', scatter=True,
              scatter_kws={'alpha': 0.1});
g.ax.set_xlim([0,100]);
```

```
/usr/local/lib/python3.5/dist-packages/scipy/stats/stats.py:1713: FutureWarning: Using a non-tuple sequence for multidimensional indexing is deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this will be interpreted as an array index, `arr[np.array(seq)]`, which will result either in an error or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



Adding eve more information

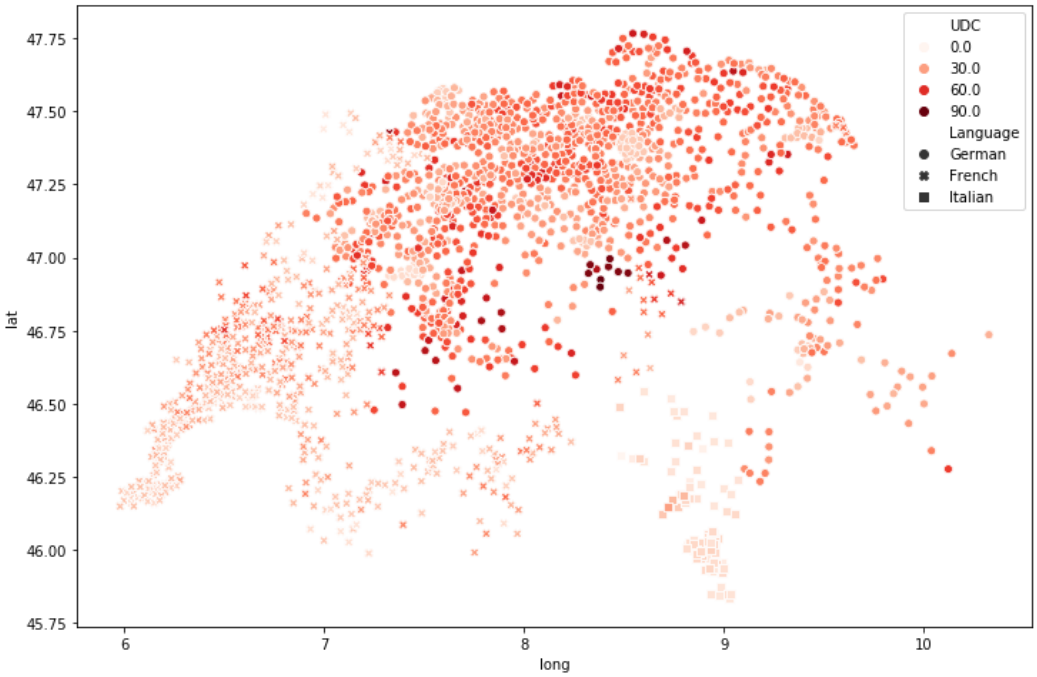
We can recover from some other place (Poste) the coordinates of each town. Again by merging we can add that information to our main table:

In [15]:
```
coords = pd.read_csv('Datasets/plz_verzeichnis_v2.csv', sep=';')[['ORTBEZ18','Geokoordinaten']]
coords['lat'] = coords.Geokoordinaten.apply(lambda x: float(x.split(',')[0]) if type(x)==str else np.nan)
coords['long'] = coords.Geokoordinaten.apply(lambda x: float(x.split(',')[1]) if type(x)==str else np.nan)
```

In [16]:
```
towns_language = pd.merge(towns_language,coords, left_on='Commune', right_on='ORTBEZ18')
```

So now we can in addition look at the geography of these parameters. For example, who votes for the right-wing party ?

In [17]:
```
fix, ax = plt.subplots(figsize = (12,8))
sns.scatterplot(data = towns_language, x= 'long', y = 'lat', hue='UDC',
style = 'Language', palette='Reds');
```



In [18]:
```
# MZ: if used to ggplot -> use 'plotnine' package
# same grammar as ggplot
```

7. Insight into Machine Learning

Several more advanced applications rely on Pandas structure to work. One example is the package scikit-learn, which has become one of the dominant machine learning resource in data science. We are now going to have a very quick look at how Pandas is used in that frame.

We are going to work again with our swiss towns infos and we will see if we can predict the result of a party based on that information.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import seaborn as sns
```

7.1 Data loading and features selection

We first load the data set:

In [2]:
```
towns = pd.read_excel('Datasets/2018.xls', skiprows=list(range(5))+list
(range(6,9)),
                      skipfooter=34, index_col='Commune',na_values=['*
','X'])
towns = towns.reset_index()
```

Now we have to get select what featurs we are going to use to predict the vote for UDC. We have to remove the results of the other parties, as those are of course correlated.

We also create a target by selecting only the UDC column

In [3]:
```
features = towns.drop('PDC', axis=1)
features = features.drop('PS', axis=1)
features = features.drop('Commune', axis=1)
features = features.drop('PVL', axis=1)
features = features.drop('PLR 2)', axis=1)
features = features.drop('PBD', axis=1)
features = features.drop('PST/Sol.', axis=1)
features = features.drop('PEV/PCS', axis=1)
features = features.drop('PES', axis=1)
features = features.drop('Petits partis de droite', axis=1)
features = features.drop('Code commune', axis=1)

features = features.dropna()
targets = features['UDC']
features = features.drop('UDC', axis=1)
```

In [4]:
```
features.head()
```

Out[4]:

	Habitants	Variation en %	Densité de la population par km²	Etrangers en %	0-19 ans	20-64 ans	65 ans ou plus	Taux brut de nuptialité	Ta div
0	1977	8.388158	249.936789	13.100658	20.586748	62.822458	16.590794	2.526529	3.03
1	11900	7.294203	1123.701605	27.848740	20.285714	62.201681	17.512605	5.167740	1.44
2	5435	5.349874	731.493943	14.149034	23.808648	60.717571	15.473781	5.389834	1.85
3	3571	6.279762	262.573529	14.533744	22.738729	60.403248	16.858023	4.540295	1.98
4	3687	8.123167	564.624809	14.971522	22.484405	62.110117	15.405479	7.622159	1.36

5 rows × 31 columns

In [5]:
```
targets.head()
```

Out[5]:
```
0    30.929249
1    33.785785
2    29.100156
3    34.937369
4    30.114599
Name: UDC, dtype: float64
```

7.2 Splitting the data

We need to be able to test whether our ML algorithm is capable of making predictions on data is has not been trained on. We therefore split our dataset into a training and a testing set. Luckily sklearn provides this out of the box if we pass the right dataframes.

In [6]:
```
from sklearn.model_selection import train_test_split
X, X_test, y, y_test = train_test_split(features, targets,
                                        test_size = 0.2,
                                        random_state = 42)
```

In [7]:
```
len(X_test)/len(X)
```

Out[7]:
```
0.25
```

7.3 Choosing an ML method

Sklearn offers a wide range of ML methods. We are not entering into details here and choose a Random Forest regression:

In [8]:
```
from sklearn.ensemble import RandomForestRegressor
```

Then we instantiate the model and train it (fit):

In [9]:
```
random_forrest = RandomForestRegressor(n_estimators=1000)
random_forrest.fit(X, y)
```

Out[9]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=1000, n_jobs=None,
           oob_score=False, random_state=None, verbose=0, warm_start=Fals
e)
```

Finally we can use it to make predictions. In particular we can apply it to our test sample and see how it performs:

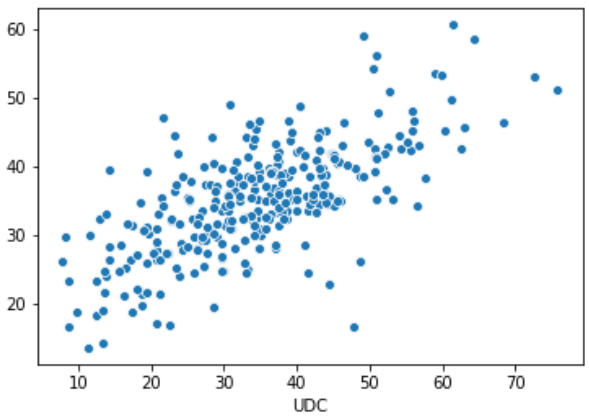
In [10]:
```
predictions = random_forrest.predict(X_test)
mae = np.mean(abs(predictions - y_test))
print(mae)
```

```
7.092857784914048
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/computation/check.py:1
9: UserWarning: The installed version of numexpr 2.4.3 is not supported i
n pandas and will be not be used
The minimum supported version is 2.6.1

  ver=ver, min_ver=_MIN_NUMEXPR_VERSION), UserWarning)
```

In [11]:
```
sns.scatterplot(x = y_test, y = predictions);
```



In [12]:
```
import scipy.stats
```

In [13]:
```
scipy.stats.pearsonr(y_test,predictions)
```

Out[13]:
```
(0.6900691335750526, 1.8302171574366992e-43)
```

7.4 Important features

A random forest classificer has the advantage that it can provide us information about how important each feature is. In other terms which features help the most in predicting:

In [14]:
```
print(random_forrest.feature_importances_)
```

```
[0.01642088 0.03087361 0.02929625 0.07733401 0.05875088 0.06434162
 0.02267082 0.03147107 0.02621289 0.02242997 0.02138494 0.01213671
 0.03465807 0.0244234  0.03648381 0.02957084 0.13884786 0.02812714
 0.02704483 0.02524017 0.01225019 0.02748501 0.01709968 0.01442152
 0.00932912 0.0553376  0.00979078 0.01229114 0.03309209 0.01542136
 0.03576174]
```

The larger the number, the better its predictions power. We can sort this list and see to what features they correspond in our feature Dataframe:

In [15]:
```
features.keys()[np.argsort(random_forrest.feature_importances_)]
```

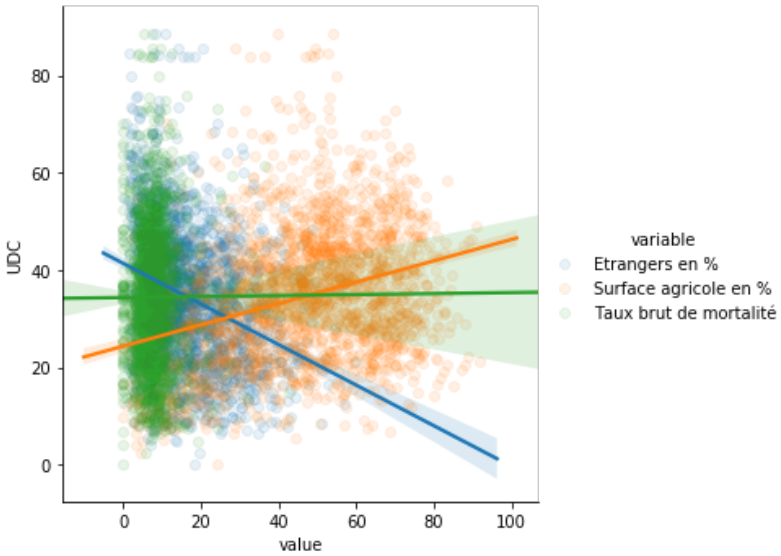
Out[15]:
```
Index(['Etablissements total', 'Secteur secondaire.1', 'Ménages privés',
       'Emplois total', 'Secteur tertiaire.1', 'Secteur tertiaire',
       'Nouveaux logements construits pour 1000 habitants', 'Habitants',
       'Secteur secondaire', 'Taux brut de mortalité', 'Taux brut de natalité',
       '65 ans ou plus', 'Surface totale en km²', 'Surface improductive en %',
       'Taux brut de divortialité', 'Surface boisée en %', 'Secteur primaire',
       'Variation en ha.1', 'Densité de la population par km²',
       'Variation en ha', 'Variation en %', 'Taux brut de nuptialité',
       'Taux de logements vacants', 'Taille moyenne des ménages en personnes',
       'Taux d'aide sociale', 'Surfaces d'habitat et d'infrastructure en %',
       'Secteur primaire.1', '0-19 ans', '20-64 ans', 'Etrangers en %',
       'Surface agricole en %'],
      dtype='object')
```

Finally, we can have a look at the actual correlations that seem to be indicated here. For this we select a few features and create a long format table for plotting:

In [16]:
```
towns_melt = pd.melt(towns, id_vars=['Commune','UDC'],
        value_vars=['Etrangers en %','Surface agricole en %','Taux brut de mortalité'])
```

In [17]:
```
sns.lmplot(data = towns_melt, x = 'value', y = 'UDC', hue = 'variable',
scatter_kws={'alpha' : 0.1});
```

```
/usr/local/lib/python3.5/dist-packages/scipy/stats/stats.py:1713: FutureW
arning: Using a non-tuple sequence for multidimensional indexing is depre
cated; use `arr[tuple(seq)]` instead of `arr[seq]`. In the future this wi
ll be interpreted as an array index, `arr[np.array(seq)]`, which will res
ult either in an error or a different result.
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



There are indeed strong correlations where they are expected! Notice alos that we can learn things here: the right-wing party is most successful where there's the least foreigners...

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Exercise

For these exercices we are using a dataset (https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data/kernels) provided by Airbnb for a Kaggle competition. It describes its offer for New York City in 2019, including types of appartments, price, location etc.

1. Create a dataframe

Create a dataframe of a few lines with objects and their poperties (e.g fruits, their weight and colour). Calculate the mean of your Dataframe.

In [2]:
```
dict_of_list = {'fruit_name': ["apple", "pear", "watermelon"], 'weight
':[100, 94, 95], 'colour':['green', "yellow", "rosa"]}
fruits = pd.DataFrame(dict_of_list)
```

In [3]:
```
fruits.describe()
# calculates common statistical values
# and makes it only for the columns that make sense
```

Out[3]:

	weight
count	3.000000
mean	96.333333
std	3.214550
min	94.000000
25%	94.500000
50%	95.000000
75%	97.500000
max	100.000000

In [4]:
```
fruits.mean()
```

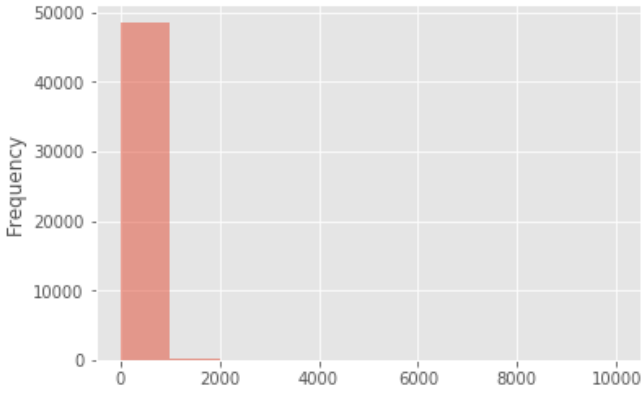
Out[4]:
```
weight    96.333333
dtype: float64
```

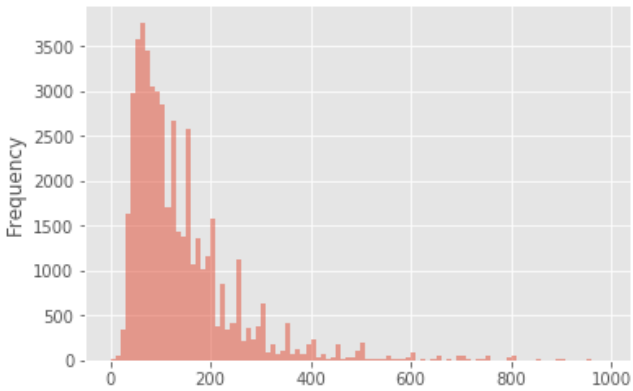
2. Import

- Import the table called `AB_NYC_2019.csv` as a dataframe. It is located in the Datasets folder. Have a look at the beginning of the table (head).
- Create a histogram of prices

In [5]:
```
mydata = pd.read_csv('Datasets/AB_NYC_2019.csv')
# mydata
```

In [6]:
```
plt.style.use('ggplot')
mydata['price'].plot.hist(alpha = 0.5)
plt.show()
# to have nicer plot (more bars)
mydata['price'].plot.hist(alpha = 0.5, bins=range(0,1000,10))
plt.show()
```





3. Operations

Create a new column in the dataframe by multiplying the "price" and "availability_365" columns to get an estimate of the maximum yearly income.

In [7]:
```
mydata['max_yearly_income'] = mydata['price'] * mydata['availability_365
']
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/computation/check.py:1
9: UserWarning: The installed version of numexpr 2.4.3 is not supported i
n pandas and will be not be used
The minimum supported version is 2.6.1

  ver=ver, min_ver=_MIN_NUMEXPR_VERSION), UserWarning)
```

In [8]:
```
# what can be done with numpy can be done
# np.log(mydata['price'])
```

In [9]:
```
# mydata
```

3b. Subselection and plotting

Create a new Dataframe by first subselecting yearly incomes between 1 and 100'000. Then make a scatter plot of yearly income versus number of reviews

In [10]:
```
#mydata_sub = mydata[ (mydata['max_yearly_income'] >= 1) and (mydata['ma
x_yearly_income'] <= 100000) ]
#mydata_sub = mydata[ (mydata.max_yearly_income >= 1) and (mydata.max_ye
arly_income <= 100000) ]
mydata_sub = mydata[ (mydata['max_yearly_income'] >= 1) & (mydata['max_y
early_income'] <= 100000) ].copy()
# mydata[(mydata.max_yearly_income>=1)&(mydata.max_yearly_income <= 1000
00)].copy()
# mydata_sub
```

In [11]:
```
mydata_sub.plot(x = 'number_of_reviews', y = 'max_yearly_income',kind =
'scatter')
max(mydata_sub['max_yearly_income'])
```

Out[11]: 99900

4. Combine

We provide below an additional table that contains the number of inhabitants of each of New York's boroughs ("neighbourhood_group" in the table). Use `merge` to add this population information to each element in the original dataframe.

In [12]:
```
borough_dt = pd.read_excel('Datasets/ny_boroughs.xlsx')
#borough_dt
```

In [13]:
```
#mydata
```

In [14]:
```
merged_dt = pd.merge(mydata, borough_dt, left_on='neighbourhood_group',
right_on='borough', how='left')
#merged_dt
```

5. Groups

- Using `groupby` calculate the average price for each type of room (room_type) in each neighbourhood_group. What is the average price for an entire home in Brooklyn ?
- Unstack the multi-level Dataframe into a regular Dataframe with `unstack()` and create a bar plot with the resulting table

```
In [15]: merged_dt.groupby(['neighbourhood_group','room_type']).price.mean()
```

```
Out[15]: neighbourhood_group  room_type
         Bronx                Entire home/apt    127.506596
                              Private room        66.788344
                              Shared room         59.800000
         Brooklyn             Entire home/apt    178.327545
                              Private room        76.500099
                              Shared room         50.527845
         Manhattan            Entire home/apt    249.239109
                              Private room       116.776622
                              Shared room         88.977083
         Queens               Entire home/apt    147.050573
                              Private room        71.762456
                              Shared room         69.020202
         Staten Island        Entire home/apt    173.846591
                              Private room        62.292553
                              Shared room         57.444444
         Name: price, dtype: float64
```

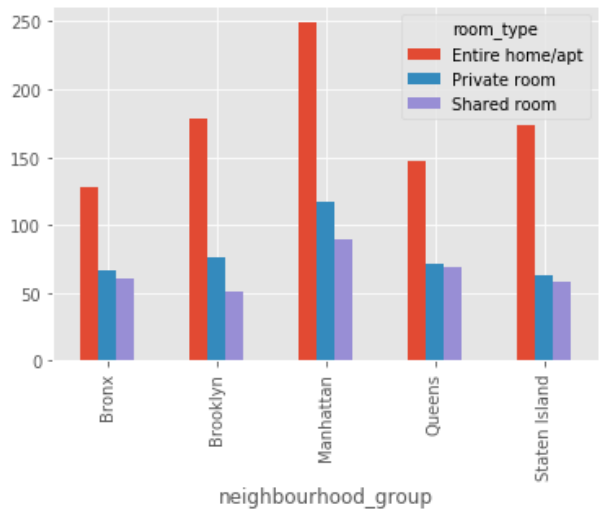
```
In [16]: merged_dt.groupby(['neighbourhood_group','room_type']).price.mean()['Bro
         oklyn']['Entire home/apt']
```

```
Out[16]: 178.32754472225128
```

```
In [17]: merged_dt.groupby(['neighbourhood_group','room_type'])['price'].mean()['
         Brooklyn']['Entire home/apt']
```

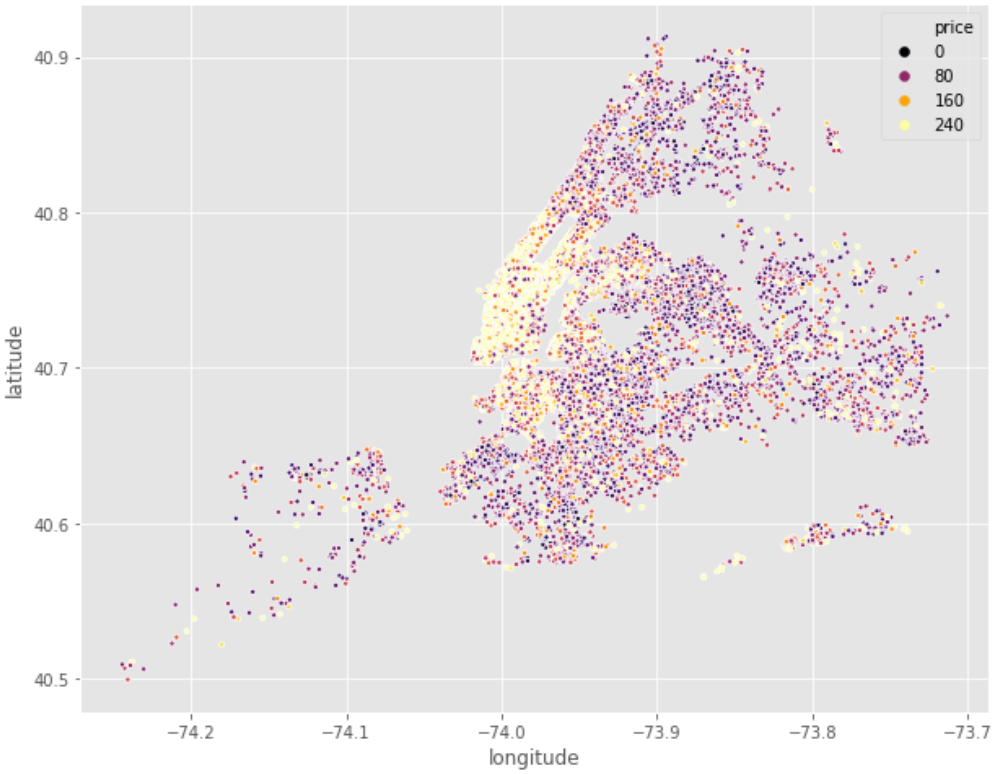
```
Out[17]: 178.32754472225128
```

```
In [18]: unstd_dt = merged_dt.groupby(['neighbourhood_group','room_type']).price.
         mean().unstack()
         unstd_dt.plot(kind = 'bar');
```



6. Advanced plotting

In [19]:
```
fig, ax = plt.subplots(figsize=(10,8))
g = sns.scatterplot(data = merged_dt, y = 'latitude', x = 'longitude', h
ue = 'price',
                    hue_norm=(0,200), s=10, palette='inferno')
```



Using Seaborn, create a scatter plot where x and y positions are longitude and lattitude, the color reflects price and the shape of the marker the borough (neighbourhood_group). Can you recognize parts of new york ? Does the map make sense ?

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Exercise

For these exercices we are using a dataset (https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data/kernels) provided by Airbnb for a Kaggle competition. It describes its offer for New York City in 2019, including types of appartments, price, location etc.

1. Create a dataframe

Create a dataframe of a few lines with objects and their poperties (e.g fruits, their weight and colour). Calculate the mean of your Dataframe.

In [2]:
```
fruits = pd.DataFrame({'fruits':['strawberry', 'orange','melon'], 'weigh
t':[20, 200, 1000], 'color': ['red','orange','yellow']})
```

In [3]:
```
fruits
```

Out[3]:

	color	fruits	weight
0	red	strawberry	20
1	orange	orange	200
2	yellow	melon	1000

In [4]:
```
fruits.mean()
```

Out[4]:
```
weight    406.666667
dtype: float64
```

2. Import

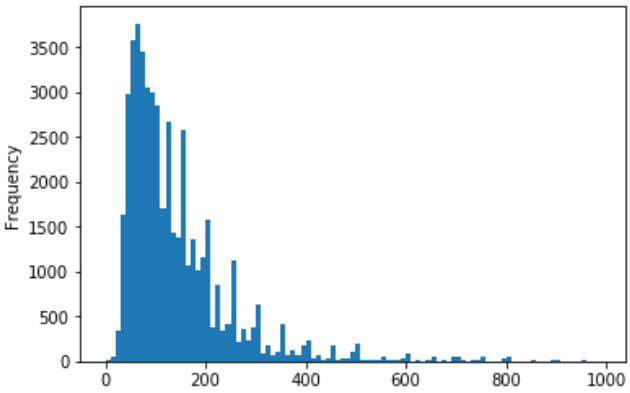
- Import the table called `AB_NYC_2019.csv` as a dataframe. It is located in the Datasets folder. Have a look at the beginning of the table (head).
- Create a histogram of prices

In [5]:
```
airbnb = pd.read_csv('Datasets/AB_NYC_2019.csv')
```

In [6]:
```
# airbnb.head()
```

In [7]:
```
airbnb.price.plot(kind = 'hist', bins = range(0,1000,10))
```

Out[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4d11f5ef28>
```



3. Operations

Create a new column in the dataframe by multiplying the "price" and "availability_365" columns to get an estimate of the maximum yearly income.

In [8]:
```
airbnb['yearly_income'] = airbnb['price']*airbnb['availability_365']
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/computation/check.py:1
9: UserWarning: The installed version of numexpr 2.4.3 is not supported i
n pandas and will be not be used
The minimum supported version is 2.6.1

  ver=ver, min_ver=_MIN_NUMEXPR_VERSION), UserWarning)
```

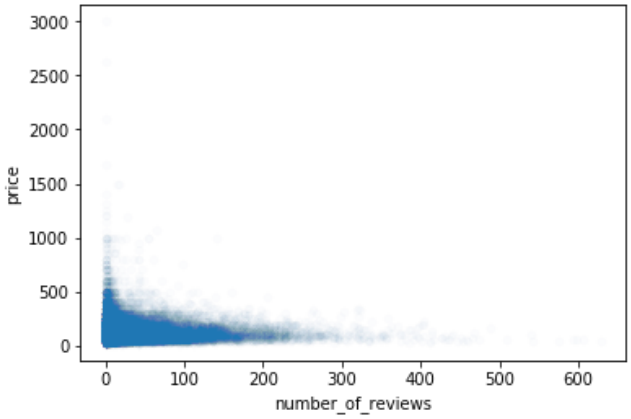
In [9]:
```
# airbnb['yearly_income']
```

3b. Subselection and plotting

Create a new Dataframe by first subselecting yearly incomes between 1 and 100'000 and then by suppressing cases with 0 reviews. Then make a scatter plot of yearly income versus number of reviews

In [10]:
```
sub_airbnb = airbnb[(airbnb.yearly_income>1)&(airbnb.yearly_income<10000
0)].copy()
```

In [11]:
```
sub_airbnb.plot(x = 'number_of_reviews', y = 'price', kind = 'scatter',
alpha = 0.01)
plt.show()
```



4. Combine

We provide below and additional table that contains the number of inhabitants of each of New York's boroughs ("neighbourhood_group" in the table). Use `merge` to add this population information to each element in the original dataframe.

In [12]:
```
boroughs = pd.read_excel('Datasets/ny_boroughs.xlsx')
```

In [13]:
```
boroughs
```

Out[13]:

	borough	population
0	Brooklyn	2648771
1	Manhattan	1664727
2	Queens	2358582
3	Staten Island	479458
4	Bronx	1471160

In [14]:
```
merged = pd.merge(airbnb, boroughs, left_on = 'neighbourhood_group', rig
ht_on='borough')
```

In [15]:
```
merged.head()
```

Out[15]:

	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude
0	2539	Clean & quiet apt home by the park	2787	John	Brooklyn	Kensington	40.64749
1	3831	Cozy Entire Floor of Brownstone	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514
2	5121	BlissArtsSpace!	7356	Garon	Brooklyn	Bedford-Stuyvesant	40.68688
3	5803	Lovely Room 1, Garden, Best Area, Legal rental	9744	Laurie	Brooklyn	South Slope	40.66829
4	6848	Only 2 stops to Manhattan studio	15991	Allen & Irina	Brooklyn	Williamsburg	40.70837

5. Groups

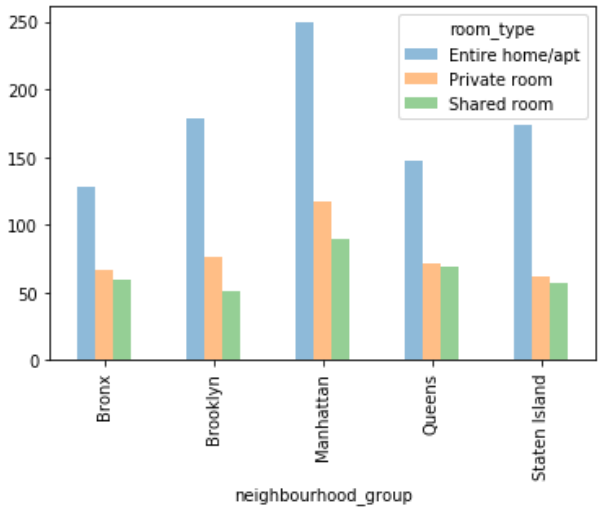
- Using `groupby` calculate the average price for each type of room (room_type) in each neighbourhood_group. What is the average price for an entire home in Brooklyn ?
- Unstack the multi-level Dataframe into a regular Dataframe with `unstack()` and create a bar plot with the resulting table

In [16]:
```
summary = airbnb.groupby(['neighbourhood_group','room_type']).mean().pri
ce
```

In [17]:
```
summary[('Brooklyn','Entire home/apt')]
```

Out[17]: 178.32754472225128

In [18]:
```
summary.unstack().plot(kind = 'bar', alpha = 0.5)
plt.show()
```



6. Advanced plotting

Using Seaborn, create a scatter plot where x and y positions are longitude and lattitude, the color reflects price and the shape of the marker the borough (neighbourhood_group). Can you recognize parts of new york ? Does the map make sense ?

In [19]:
```
fig, ax = plt.subplots(figsize=(10,8))
g = sns.scatterplot(data = airbnb, y = 'latitude', x = 'longitude', hue
= 'price',
                    hue_norm=(0,200), s=10, palette='inferno')
```