Introduction to Pandas

Tabular data and more

Programme: four blocks

Introduction to Series and DataFrames

First operations, reading/writing

Plotting capacities

Timeseries analysis

```python
import pandas as pd
```

Part I: pandas objects

Series and Dataframes

pd.Series: introduction

- List of values with indices

Indices of the values	List of values for a given variable
Elf	200
Dwarf	120
Hobbit	110

```
Elf       220
Dwarf     120
Hobbit    110
dtype: int64
```

pd.Series: index and data types

- Variables can have a **type** or not

```
index 0    4
index 1    7
index 2    5
index 3    7
index 4    2
dtype: int32
```

```
index 0    1.684448
index 1    0.592174
index 2    0.938397
index 3    4.136875
index 4    1.495691
dtype: float64
```

```
index 0        These
index 1          are
index 2    different
index 3      strings
index 4            !
dtype: string
```

```
index 0      Hello !
index 1         0.41
index 2           12
index 3    [1, 2, 5]
index 4           x²
dtype: object
```

- Indices can also have different **types**

```
index 0    3
index 1    6
index 2    2
index 3    1
index 4    0
dtype: int32
```

```
0    8
1    9
2    1
3    1
4    5
dtype: int32
```

```
2012-01-01    9
2013-01-01    5
2014-01-01    8
2015-01-01    7
2016-01-01    4
Freq: YS-JAN, dtype: int32
```

pd.Series: creating an object

- Required: a list of values + a list of indices
- Use **pd.Series()**

```
characters = ['Elf', 'Dwarf', 'Hobbit']
size = [200, 120, 110]
pd.Series(data=size, index=characters, name='size', dtype='int')
```

```
Elf       200
Dwarf     120
Hobbit    110
Name: size, dtype: int32
```

Required

If not specified: 0,1,2 ,…

Not required

pd.Series: access data

```
Elf       200
Dwarf     120
Hobbit    110
Name: size, dtype: int32
```

```
size_series.loc['Elf']
```

```
200
```

```
size_series.iloc[1]
```

```
120
```

- Access data using **indices**
- Access data using **position**

- Access the data as a numpy array

```
size_series.values
```

```
array([200, 120, 110])
```

pd.DataFrame: introduction

- Multiple pd.Series **aligned** on the same indices
- Each Series has a name

pd.Series

Index

	size	weight	life_expectancy	home
Elf	200	80	1000	Rivendale
Dwarf	120	120	300	Moria
Hobbit	110	40	120	The Shire

pd.DataFrame: creating an object

Input data:

```python
characters = ['Elf', 'Dwarf', 'Hobbit']
```

```python
size = [200, 120, 110]
weight = [80, 120, 40]
life_expectancy = [1000, 300, 120]
home = ['Rivendale', 'Moria', 'The Shire']
```

Object creation:

```python
pd.DataFrame(data={"size":size,
                   "weight":weight,
                   "life_expectancy":life_expectancy,
                   "home":home},
             index=characters)
```

	size	weight	life_expectancy	home
Elf	200	80	1000	Rivendale
Dwarf	120	120	300	Moria
Hobbit	110	40	120	The Shire

pd.DataFrame: creating an object, option 2

Input data:

```python
characters = ['Elf', 'Dwarf', 'Hobbit']
```

```python
data=[[200, 80, 1000, 'Rivendale'],
      [120, 120, 300, 'Moria'],
      [110, 40, 120, 'The Shire']],
```

Object creation:

```python
pd.DataFrame(data=[[200, 80, 1000, 'Rivendale'],
                   [120, 120, 300, 'Moria'],
                   [110, 40, 120, 'The Shire']],
             columns=['size','weight','life_expectancy','home'],
             index=characters)
```

	size	weight	life_expectancy	home
Elf	200	80	1000	Rivendale
Dwarf	120	120	300	Moria
Hobbit	110	40	120	The Shire

pd.DataFrame: access the data

- **Select data from index/columns or positions**
  - Using **positions**:

```
df.iloc[1,3]
```

```
'Moria'
```

  - From **index/column**

```
df.loc['Dwarf','home']
```

```
'Moria'
```

  - Multiple selection

```
df.loc[['Dwarf', 'Elf'],['home','weight']]
```

	home	weight
Dwarf	Moria	120
Elf	Rivendale	80

	size	weight	life_expectancy	home
Elf	200	80	1000	Rivendale
Dwarf	120	120	300	Moria
Hobbit	110	40	120	The Shire

pd.DataFrame: access the data

- **Access data at one index**
  - Returns a pd.Series

```
df.loc['Elf']
```

```
size                    200
weight                   80
life_expectancy        1000
home              Rivendale
Name: Elf, dtype: object
```

```
df.iloc[1]
```

```
size                 120
weight               120
life_expectancy      300
home               Moria
Name: Dwarf, dtype: object
```

- **Access data from one pd.Series**
  - Returns a pd.Series

```
df.weight
```

```
df['weight']
```

```
Elf        80
Dwarf     120
Hobbit     40
Name: weight, dtype: int64
```

```
df.iloc[:,3]
```

```
Elf       Rivendale
Dwarf         Moria
Hobbit    The Shire
Name: home, dtype: object
```

pd.DataFrame: append data

- **Add a column - pd.Series**
  - Add the Series as if you want to access the data of a column

```
df['example_name'] = ['Legolas','Gimli','Frodo']
df
```

	size	weight	life_expectancy	home	example_name
Elf	200	80	1000	Rivendale	Legolas
Dwarf	120	120	300	Moria	Gimli
Hobbit	110	40	120	The Shire	Frodo

- The values of the series can also be **updated** this way.

pd.DataFrame: append data

- **Add a row**
  - Add the row as if you want to access the data of a line

```
df.loc['Human'] = [170, 70, 75,'Gondor','Aragorn']
df
```

	size	weight	life_expectancy	home	example_name
Elf	200	80	1000	Rivendale	Legolas
Dwarf	120	120	300	Moria	Gimli
Hobbit	110	40	120	The Shire	Frodo
Human	170	70	75	Gondor	Aragorn

- The values of a line can also be updated this way.

Reading/writing files: supported formats

- Pandas supports many **tabular-like** data formats
  - csv
  - excel (xls, xlsx) : *requiert openpyxl*
  - txt
  - json
  - …
- Everything that looks like tabular data
- Can open directly from internet using url

Reading/writing files: reading

- To **open** a file, use `pd.read_format()`

```
penguins = pd.read_json('../data/penguins_dataset.json')
```

```
planets = pd.read_excel('../data/exoplanets_discoveries.xlsx')
```

```
boats = pd.read_csv('../data/fishing_boats.csv')
```

```
pd.read_csv('https://odre.opendatasoft.com/temperature-quotidienne-regionale.csv')
```

- Some important **arguments**
  - `index_col` → columns corresponding to index
  - `skiprows` → skip the first n rows
  - `nrows` → only read the first n rows avec the skipped rows

```
boats = pd.read_csv('../data/fishing_boats.csv', index_col=1, skiprows=5, nrows=10)
```

Reading/writing files: writing

- To **write** a file, use pd.to_format()

```
personnages_lotr.to_csv('../data/lotr_personnages.csv')
```

```
personnages_lotr.to_json('../data/lotr_personnages.json')
```

```
personnages_lotr.to_excel('../data/lotr_personnages.xlsx')
```

Part I: Summary

pd.Series : one variable only

pd.DataFrame : multiple variables aligned

- Easy access to data using **index** values
- Objects are flexible and can be **modified**

Part I: Practicals

Go to the jupyter notebook

Part II: First analyses

Statistical operations, filtering, groups and others

Filter data

Data where mean radius > 1000km?

```
df.meanRadius

eName
Moon                33.0
Phobos              33.0
Deimos              33.0
Io                1821.5
Europa            1560.8
                   ...
S/2017 J 8           0.5
S/2017 J 9           1.5
Ersa                 1.5
Ultima Thule        33.0
101955 Bennu        33.0
Name: meanRadius, Length: 265, dtype: float64
```

- **Extracting** data

```
df.loc[df.meanRadius > 1000].meanRadius

eName
Io                 1821.5000
Europa             1560.8000
Ganymede           2631.2000
Callisto           2410.3000
Tethys             1066.0000
```

- **Masking** data

```
df.where(df.meanRadius > 1000).meanRadius

eName
Moon                   NaN
Phobos                 NaN
Deimos                 NaN
Io                  1821.5
Europa              1560.8
```

Remove rows that contain **nans**

```
df.bondAlbido.dropna()
```

Filter data

Multiconditions:

AND: &

```
df.loc[(df.meanRadius > 10000) & (df.isPlanet)]
```

eName	isPlanet	semimajorAxis	perihelion	aphelion
Uranus	True	2870658186	2147483647	2147483647
Neptune	True	4498396441	2147483647	2147483647
Jupiter	True	778340821	740379835	816620000
Saturn	True	1426666422	1349823615	1503509229

OR: |

```
df.loc[(df.meanRadius > 10000) | (df.isPlanet)]
```

eName	isPlanet	semimajorAxis	perihelion	aphelion
1 Ceres	True	413690250	382620000	445410000
136199 Eris	True	10180122852	2147483647	2147483647
Uranus	True	2870658186	2147483647	2147483647
Pluto	True	5906440628	2147483647	2147483647

Numpy-like operations: how to

- Operation on one series:
  `series.mean()`

```
df.meanRadius.mean()
```

```
3481.5921071698117
```

- Ignores nans by default `np.nanmean`
- Works on **Series** and **Dataframe**

```
df.max(numeric_only=True)
```

```
isPlanet                True
semimajorAxis   152000000000
perihelion        2147483647
aphelion          2147483647
eccentricity          0.7512
inclination            179.8
```

- Operation between aligned series:
  `series1 + series2`

```
df.mass_kg / df.volume
```

Numpy-like operations: examples of existing operations

Statistical operations returning **one** metric

`mean`	Mean value	`min`	Lowest value	`max`	Highest value	`median`	Median value
`count`	Number of non nan values	`skew`	Skewness of the series	`kurt`	Kurtosis of the series	`mode`	Mode of the series
`var`	Variance of the series	`quantile`	Find quantiles	`sum`	Sum all elements in the series	`std`	Standard deviation

Some operations returning **multiple** values

`rank`	Ranks each elements in the series	`unique`	Returns unique values of elements	`cumsum`	Cumulative sum of the elemens	`cumprod`	Cumulative product of the elements

Get location of specific elements

`idxmin`	Get the **index** of the minimum value	`idxmax`	Get the **index** of the maximum value	`argmin`	Get the **position** of the minimum value	`argmax`	Get the **position** of the maximum value

Custom and multiple aggregations

- Apply **custom** function

```
def second_highest(data):
    sorted_data = data.sort_values(ascending=False)
    return sorted_data.iloc[1]
df.apply(second_highest).meanRadius
```

```
69911.0
```

- Apply **multiple** functions with `agg`

```
df.meanRadius.agg(['mean','std','var','max','min'])
```

```
mean    3.481592e+03
std     4.314101e+04
var     1.861146e+09
max     6.963420e+05
min     3.000000e-01
Name: meanRadius, dtype: float64
```

- Implemented `describe`

```
df.describe()
```

	semimajorAxis	perihelion	aphelion
count	2.650000e+02	2.650000e+02	2.650000e+02
mean	1.330740e+09	5.999915e+07	6.149610e+07
std	1.048573e+10	3.332062e+08	3.365186e+08
min	0.000000e+00	0.000000e+00	0.000000e+00
25%	2.946720e+05	0.000000e+00	0.000000e+00
50%	2.015529e+07	0.000000e+00	0.000000e+00
75%	2.392800e+07	0.000000e+00	0.000000e+00
max	1.520000e+11	2.147484e+09	2.147484e+09

Big warning: no loops !

- Many different available operations
- Potential to use **any** custom operation

If your code looks like this … **DON'T!**

```
results = []
for i in range(df.index.size):
    line = df.iloc[i]
    results.append(line.meanRadius**2)
```

Do this instead !

```
results = df.meanRadius**2
```

1000x faster

There is (almost) always a way to replace a slow loop by a fast pandas function

Sorting data

- Sort by index:

```
df.sort_index()
```

- Sort by one of the columns

```
df.sort_values('meanRadius', ascending=False).meanRadius
```

```
eName
Sun           696342.0
Jupiter        69911.0
Saturn         58232.0
Uranus         25362.0
Neptune        24622.0
```

Grouping data using groupby

- We can create **groups** depending on a **column** and apply operations on each of the groups **separatly**

```
df.groupby('isPlanet').meanRadius.mean()
```

```
isPlanet
False     2867.689286
True     15381.862185
Name: meanRadius, dtype: float64
```

- Using bins : `pd.cut`

```
bins = pd.cut(df.meanRadius, [0, 1000, 10000, 100000])
bins
```

```
eName
Moon                 (0, 1000]
Phobos               (0, 1000]
Deimos               (0, 1000]
Io               (1000, 10000]
Europa           (1000, 10000]
```

```
df.groupby(bins).gravity.mean()
```

```
meanRadius
(0, 1000]              0.013682
(1000, 10000]          2.128938
(10000, 100000]       13.812500
Name: gravity, dtype: float64
```

Grouping data using groupby

- **Multiple** criteria

```
df.groupby([df.isPlanet, bins]).meanRadius.max()
```

```
isPlanet  meanRadius
False     (0, 1000]              788.9000
          (1000, 10000]         2631.2000
          (10000, 100000]             NaN
True      (0, 1000]              725.0000
          (1000, 10000]         6371.0084
          (10000, 100000]      69911.0000
Name: meanRadius, dtype: float64
```

Combining dataframes

- Concatenate using the indices

```
pd.concat([df1,df2])
```

df1

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011

df2

	method	number	orbital_period	mass	distance	year
3	Radial Velocity	1	326.03	19.4	110.62	2007
4	Radial Velocity	1	516.22	10.5	119.47	2009
5	Radial Velocity	1	185.84	4.8	76.39	2008

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009
5	Radial Velocity	1	185.840	4.80	76.39	2008

Combining dataframes

- Concatenate using columns

```
pd.concat([df1,df2], axis=1)
```

	method	number	orbital_period
0	Radial Velocity	1	269.300000
1	Radial Velocity	1	874.774000
2	Radial Velocity	1	763.000000
3	Radial Velocity	1	326.030000
4	Radial Velocity	1	516.220000

df1

	mass	distance	year
0	7.10	77.40	2006
1	2.21	56.95	2008
2	2.60	19.84	2011
3	19.40	110.62	2007
4	10.50	119.47	2009

df2

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300000	7.10	77.40	2006
1	Radial Velocity	1	874.774000	2.21	56.95	2008
2	Radial Velocity	1	763.000000	2.60	19.84	2011
3	Radial Velocity	1	326.030000	19.40	110.62	2007
4	Radial Velocity	1	516.220000	10.50	119.47	2009

One cool operation

- Compute correlation between all variables

```
df.corr(numeric_only=True)
```

	isPlanet	semimajorAxis	perihelion	aphelion	ec
isPlanet	1.000000	0.037617	0.792468	0.802667	
semimajorAxis	0.037617	1.000000	0.065412	0.064542	
perihelion	0.792468	0.065412	1.000000	0.999449	
aphelion	0.802667	0.064542	0.999449	1.000000	
eccentricity	-0.032432	-0.089975	0.012518	0.011152	

Part II summary: first operations

Some of the available methods:

Access data using index/columns + filters	`df.loc[index]` `df.loc[condition]`
Numpy-like **operations**	`df.mean()` `df.apply(function)` `df.agg(['std'])`
Sort data by index/values	`df.sort_values()` `df.sort_index()`
Operations on **groups** of data	`df.groupby(columns).mean()`
Combine multiple dataframes	`pd.concat([df1, df2])`
Correlation	`df.corr()`

And so much more …

Bonus: « one-liners »

- Each method usually returns a **new pandas object** (Series, DataFrame)
- We can keep **adding** more and more operations

```
df.loc[df.gravity > df.gravity.loc['Mercury']]\
    .groupby(pd.cut(df.semimajorAxis_AU, [0,5,40]))\
    .meanRadius\
    .agg(['mean','std'])
```

	mean	std
semimajorAxis_AU		
(0, 5]	5270.769467	1637.026147
(5, 40]	44531.750000	23062.817411

Part II: Practicals

Go to the jupyter notebook

Part III: Plotting

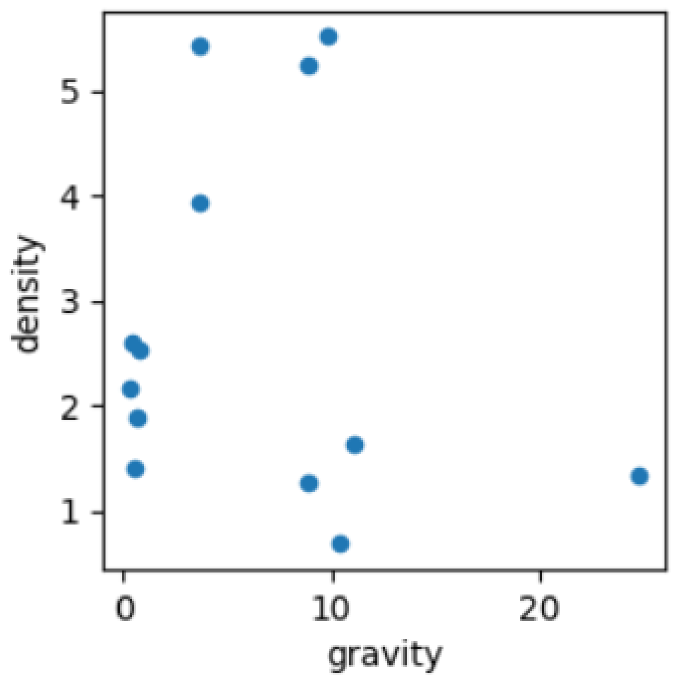
Plotting interface, integration with matplotlib and seaborn

Plotting interface

- Direct integration of high level **plotting libraries** such as **matpltolib**

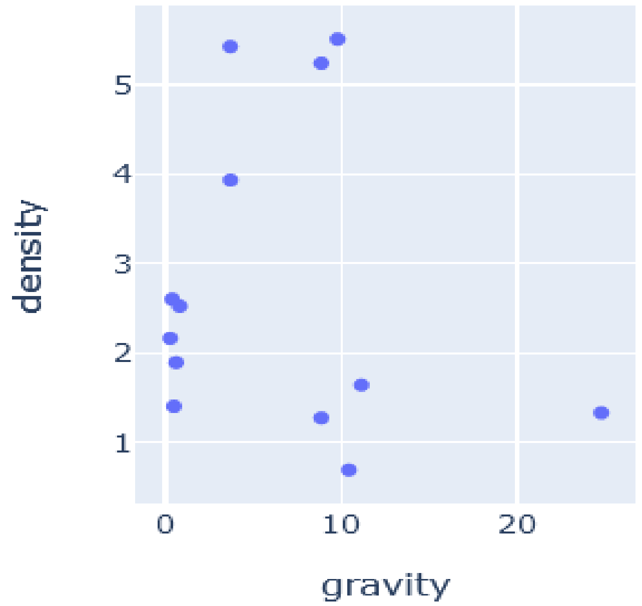
```
df_planet.plot.scatter(x='gravity', y='density', figsize=(3,3))
```

```
<AxesSubplot:xlabel='gravity', ylabel='density'>
```



Other backends possible (**Plotly**, **bokeh**, **altair**,…)

```
pd.options.plotting.backend = "plotly"
```



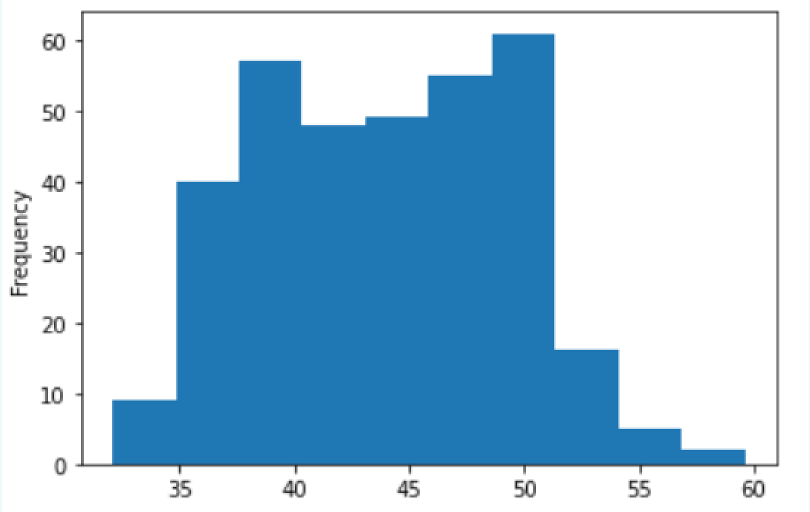
Plotting statistics for Series

- Integrated plotting methods:

```
series.plot.[type of plot]()
```

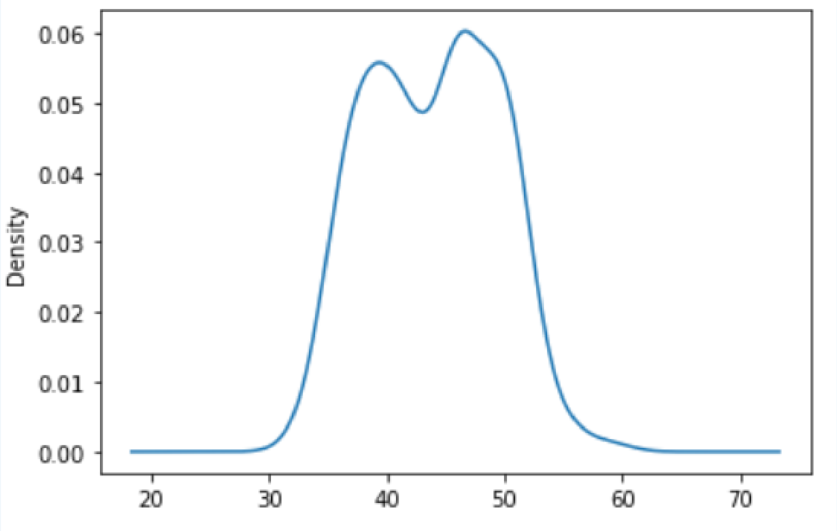
```
penguins.bill_length_mm.plot.hist()
```

```
<AxesSubplot:ylabel='Frequency'>
```



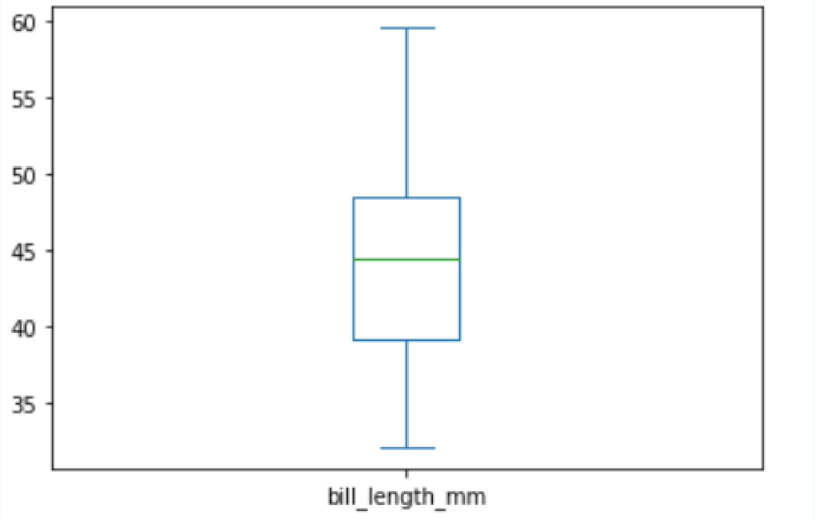

```
penguins.bill_length_mm.plot.kde()
```

```
<AxesSubplot:ylabel='Density'>
```




```
penguins.bill_length_mm.plot.box()
```

```
<AxesSubplot:>
```

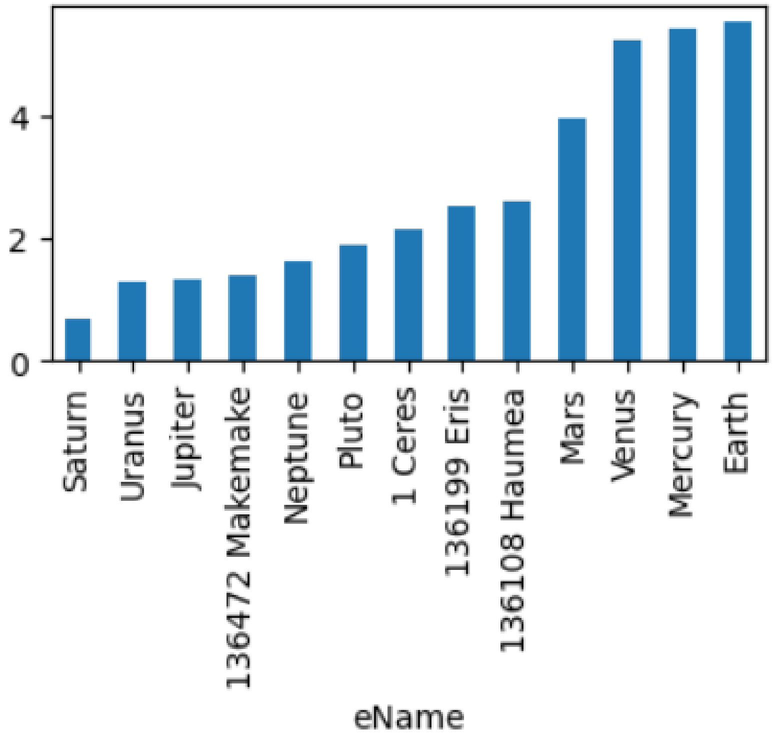


- Pandas calls the **matplotlib** function: same arguments

Plotting categorical data with barplots

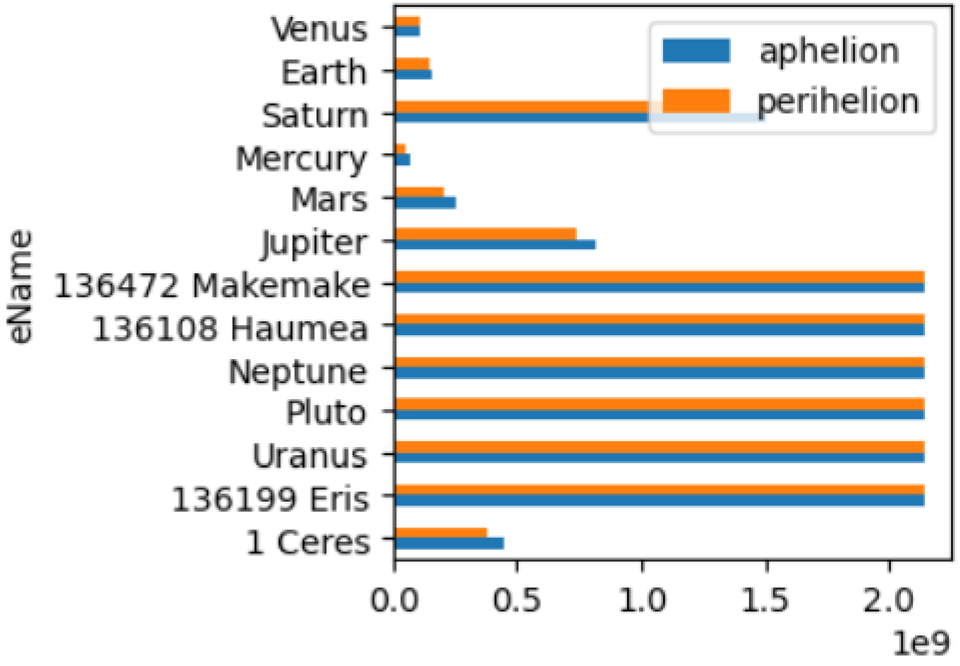
```
df_planet.density.sort_values().plot.bar(figsize=(4,2))
```

```
<AxesSubplot:xlabel='eName'>
```




```
df_planet[['aphelion','perihelion']].plot.barh(figsize=(3,3))
```

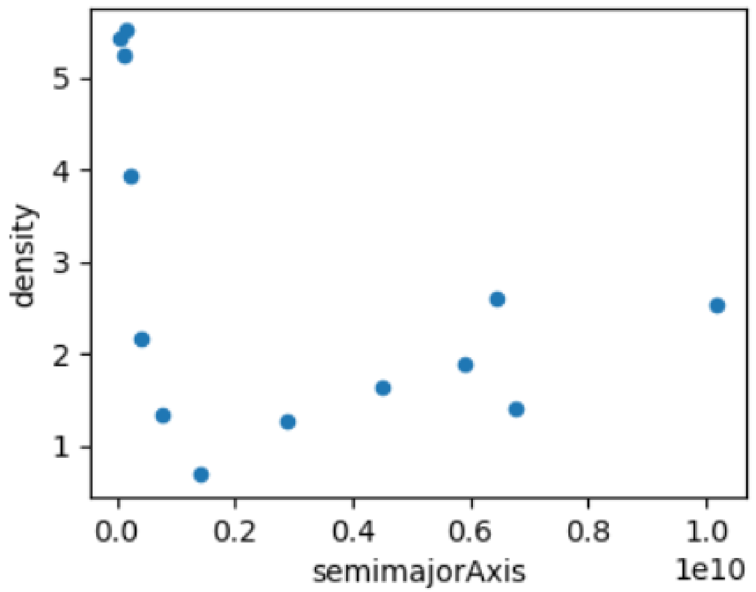
```
<AxesSubplot:ylabel='eName'>
```



Plotting relations between data: scatter/hexbin

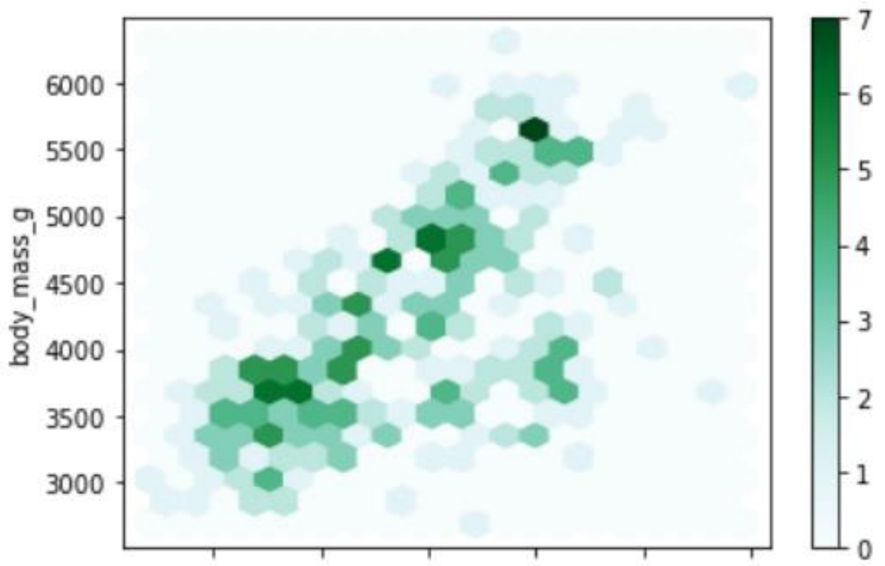
```
df_planet.plot.scatter(x='semimajorAxis', y='density', figsize=(4,3))
```

```
<AxesSubplot:xlabel='semimajorAxis', ylabel='density'>
```




```
penguins.plot.hexbin(x='bill_length_mm', y='body_mass_g',
                     gridsize=20)
```

```
<AxesSubplot:xlabel='bill_length_mm', ylabel='body_mass_g'>
```



Plotting functions available

kind : ***str***

The kind of plot to produce:

- 'line' : line plot (default)
- 'bar' : vertical bar plot
- 'barh' : horizontal bar plot
- 'hist' : histogram
- 'box' : boxplot
- 'kde' : Kernel Density Estimation plot
- 'density' : same as 'kde'
- 'area' : area plot
- 'pie' : pie plot
- 'scatter' : scatter plot (DataFrame only)
- 'hexbin' : hexbin plot (DataFrame only)

Integrating within a matplotlib workflow

Setup **figure** and **axes**:

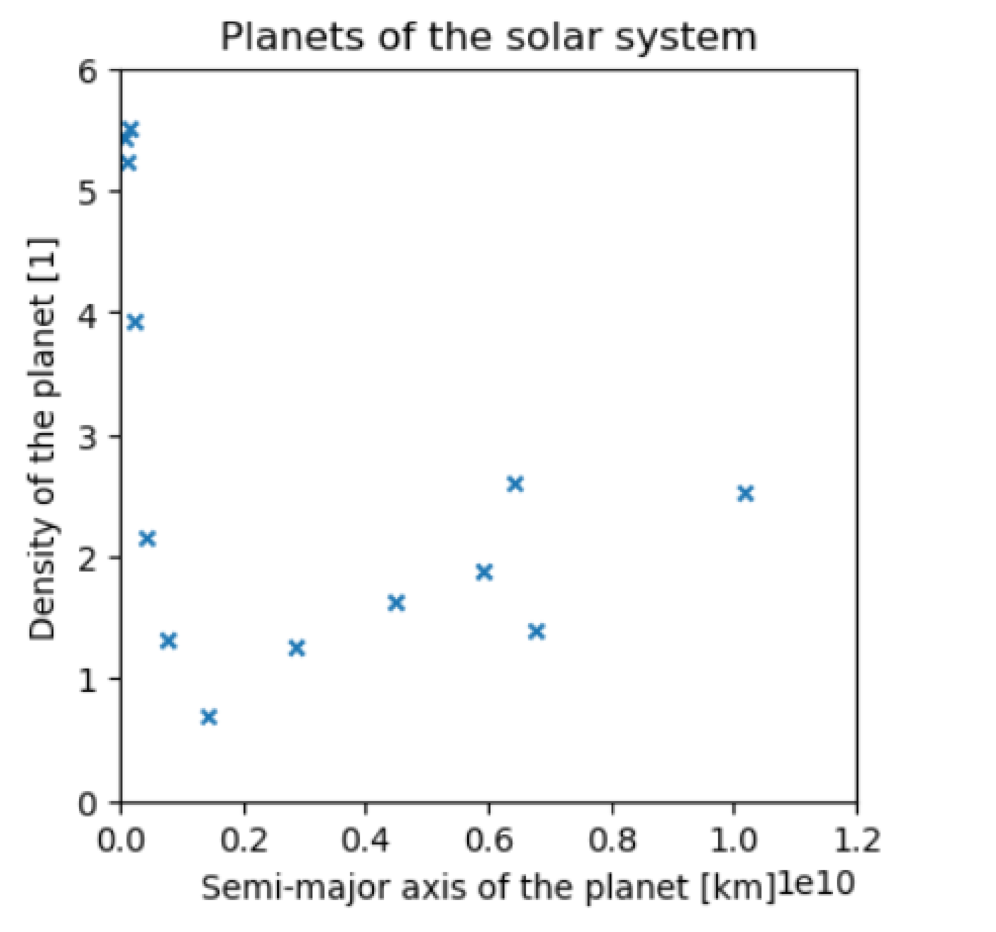
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(4,4))
```

Plot with pandas on existing axis:

```
df_planet.plot.scatter(x='semimajorAxis',
                       y='density',
                       marker='x',
                       ax=ax)
```

Change title, labels, limits, etc…

```
ax.set_title('Planets of the solar system')
ax.set_ylim(0,6)
ax.set_xlim(0,1.2e10)
ax.set_ylabel('Density of the planet [1]')
ax.set_xlabel('Semi-major axis of the planet [km]')
```



What about seaborn?

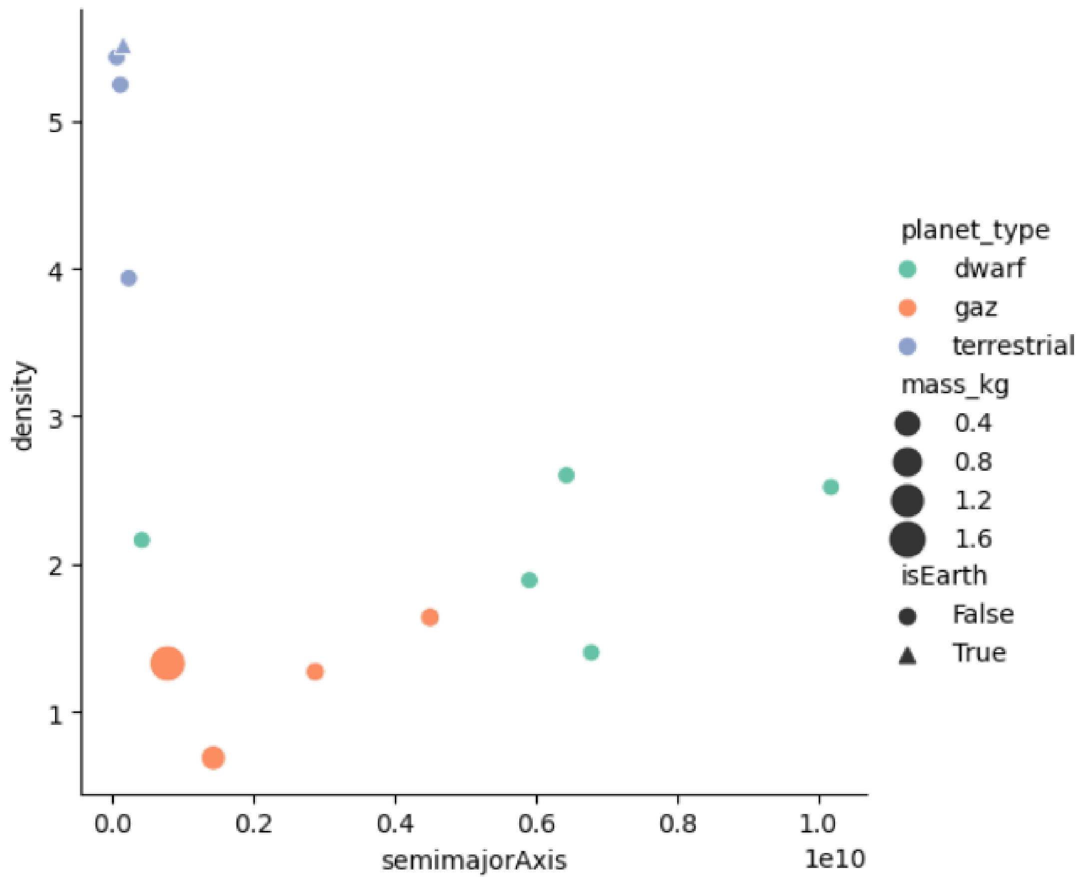
seaborn

Visualization library
optimised for dataframes

Coming soon in the Atelier numérique de l'OMP

```python
import seaborn as sns
sns.relplot(data=df_planet_type,
            x='semimajorAxis', y='density',
            hue='planet_type',palette='Set2',
            size='mass_kg',sizes=(50,200),
            style=pd.Series(df_planet_type.index=='Earth', index=df_planet_type.index).rename('isEarth'),
            markers=['o','^'],
           )
```

```
<seaborn.axisgrid.FacetGrid at 0x21b03227c40>
```



Part III: Practicals

Go to the jupyter notebook

Part IV: Timeseries

Example of data

- Series or DataFrame with **datetime index**
- Constant time step or not
- *Example*: temperature logger in the ocean

	Temperature
2019-04-13 14:00:00	28.4728
2019-04-13 14:00:10	28.4780
2019-04-13 14:00:20	28.4818
2019-04-13 14:00:30	28.5127
2019-04-13 14:00:40	28.5199
...	...
2020-10-02 03:19:09	28.6340
2020-10-02 03:19:19	28.6339
2020-10-02 03:19:29	28.6339
2020-10-02 03:19:39	28.6341
2020-10-02 03:19:49	28.6339

How to get a datetime index?

- **Create** a new index:

```
pd.date_range(start = '2014', end = '2018', freq = 'YS')
```

```
DatetimeIndex(['2014-01-01', '2015-01-01', '2016-01-01', '2017-01-01',
               '2018-01-01'],
              dtype='datetime64[ns]', freq='YS-JAN')
```

```
pd.date_range(start = '2014-01-03 12:00:00', freq = 'min', periods = 10)
```

```
DatetimeIndex(['2014-01-03 12:00:00', '2014-01-03 12:01:00',
               '2014-01-03 12:02:00', '2014-01-03 12:03:00',
               '2014-01-03 12:04:00', '2014-01-03 12:05:00',
               '2014-01-03 12:06:00', '2014-01-03 12:07:00',
               '2014-01-03 12:08:00', '2014-01-03 12:09:00'],
              dtype='datetime64[ns]', freq='min')
```

- **Frequencies**?
  - `YS, Q, MS, W, D, h, min, s, ms, us, ns`
  - Can also add a **number**: `10h`

- From **strings**

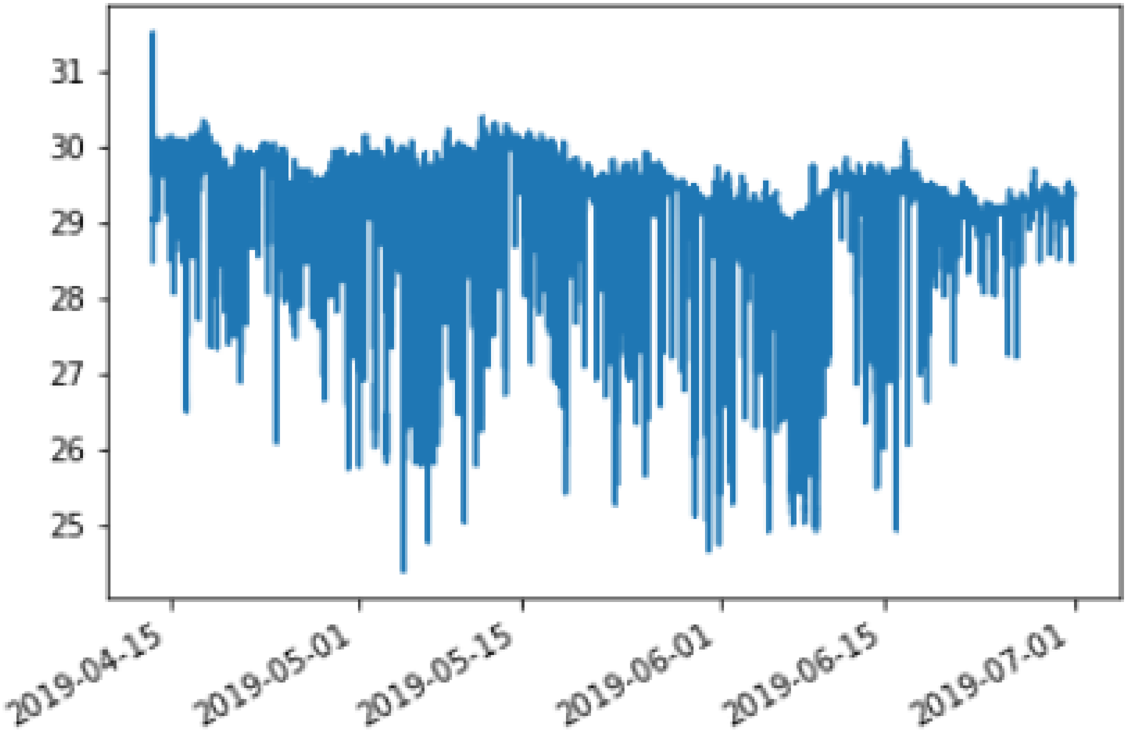
```
pd.to_datetime(['2010','2014','2023'])
```

```
pd.to_datetime(['2010-01-03','2014-02-12','2023-05-05'])
```

Plot the timeseries

```
df.Temperature.plot()
```

```
<AxesSubplot:>
```

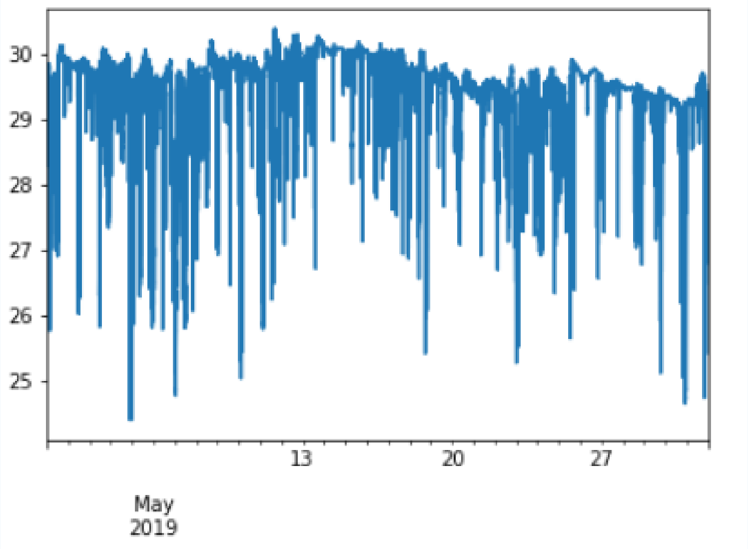


Select data based on time

- Use `df.loc[time]` for **specific** times

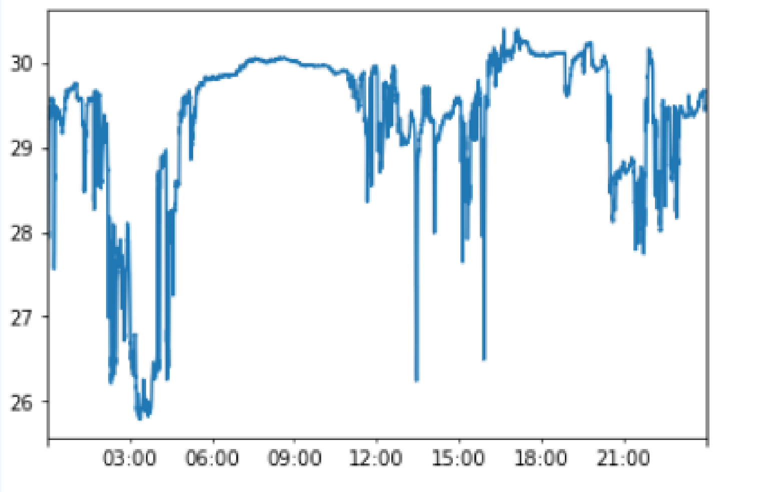
```
df.Temperature.loc['2019-05'].plot()
```

<AxesSubplot:>



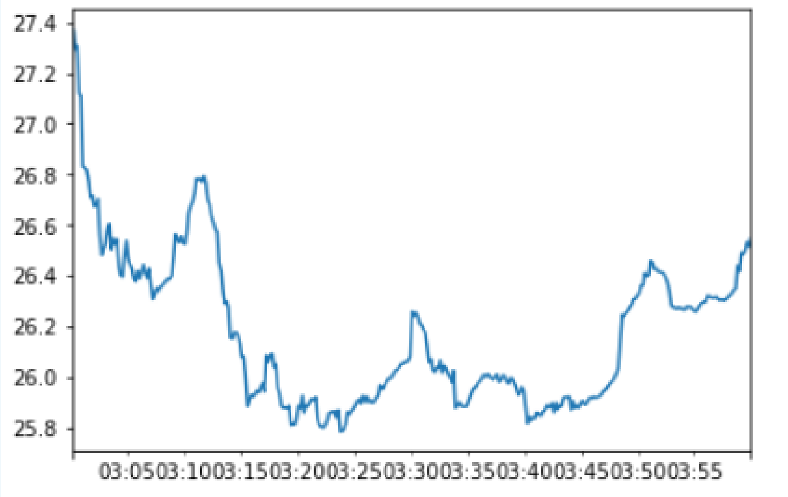

```
df.Temperature.loc['2019-05-11'].plot()
```

<AxesSubplot:>



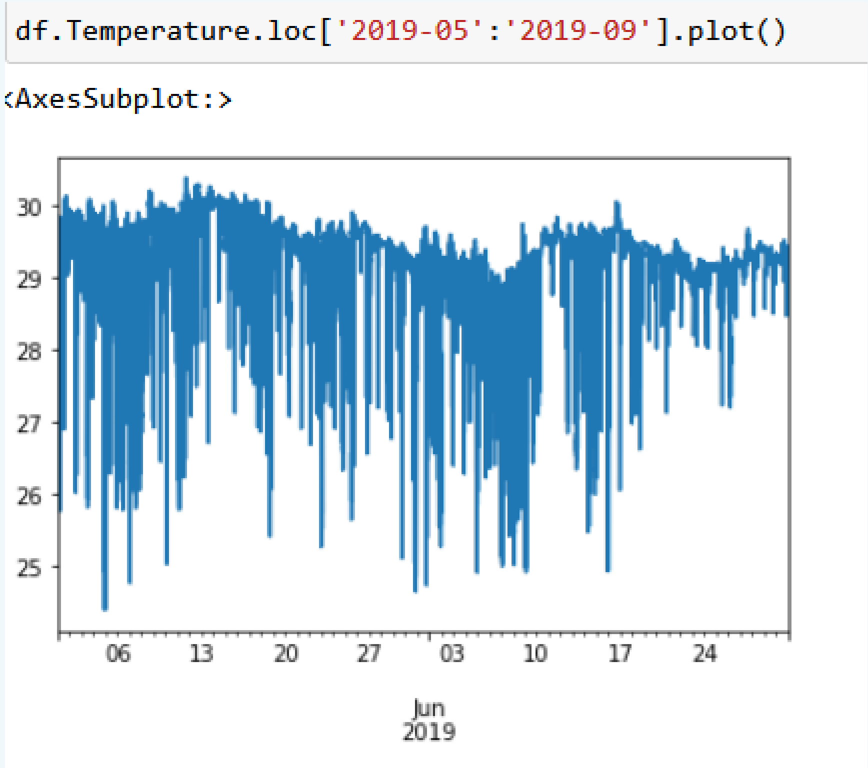

```
df.Temperature.loc['2019-05-11 3H'].plot()
```

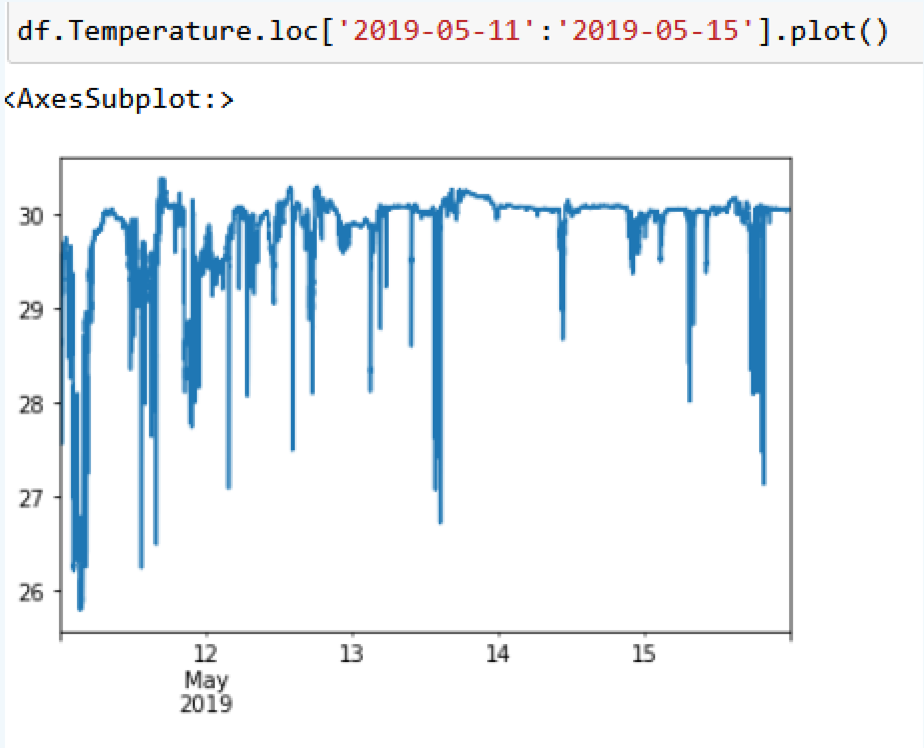
<AxesSubplot:>



Select data based on time

- Use `df.loc[time1:time2]` for **intervals**





Resampling

- Use the same **frequencies** as in date_range:

```
df.Temperature.resample('H')
```

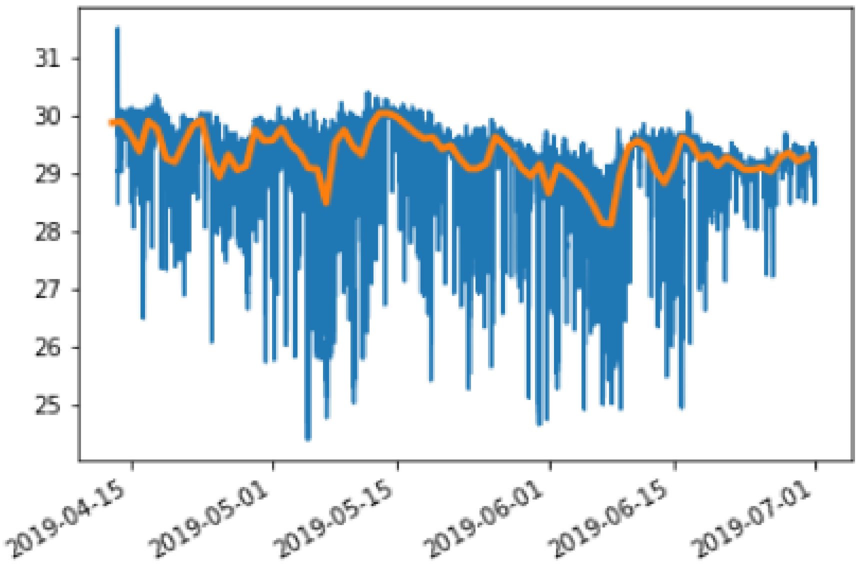
- Returns an object that **expects an operation**
  - `mean`, `max`, `quantile`, `count`, … or `apply()`

```
df.Temperature.resample('D').mean()
```

```
2019-04-13    29.869072
2019-04-14    29.883743
2019-04-15    29.671858
2019-04-16    29.371685
2019-04-17    29.896633
```

```
df.Temperature.plot()
df.Temperature.resample('D').mean().plot(lw=3)
```

```
<AxesSubplot:>
```

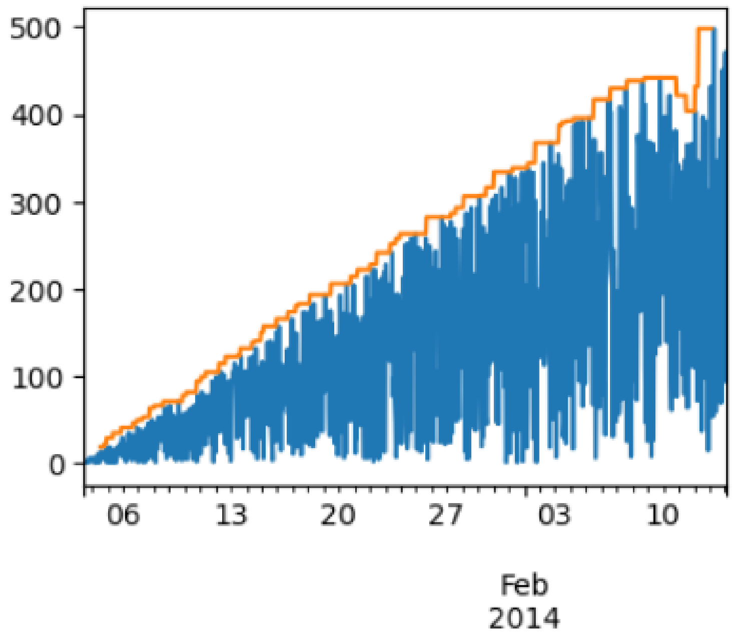


Rolling windows

- Keeps the **same sampling rate**
- Returns an object as in groupby: **expects an operation**
  - `mean`, `max`, `quantile`, `count`, … or `apply()`

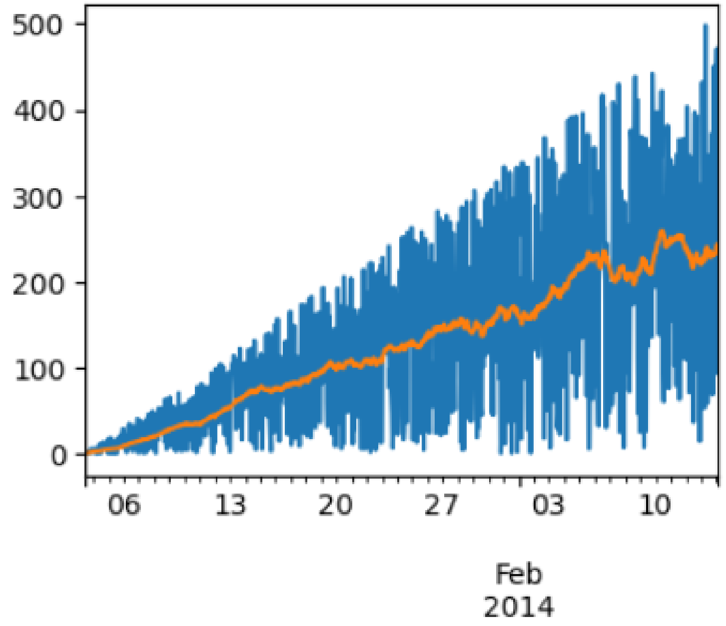
```
s.plot()
s.rolling(50, center=True).max().plot(figsize=(4,3))
```

```
<AxesSubplot:>
```




```
s.plot()
s.rolling('2D').mean().plot(figsize=(4,3))
```

```
<AxesSubplot:>
```



Dealing with missing data

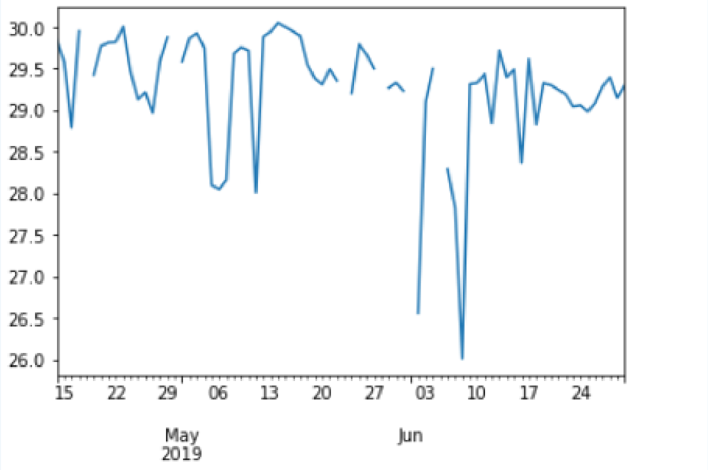
	Temperature
2019-04-14 01:04:09	29.8508
2019-04-14 16:16:29	29.8557
2019-04-15 05:10:19	30.0335
2019-04-15 07:51:29	29.4874
2019-04-15 14:08:49	29.8838

- Can we get **daily** data?

```
.resample('D').mean()
```

```
df_irregulier.Temperature.resample('D').mean().plot()
```

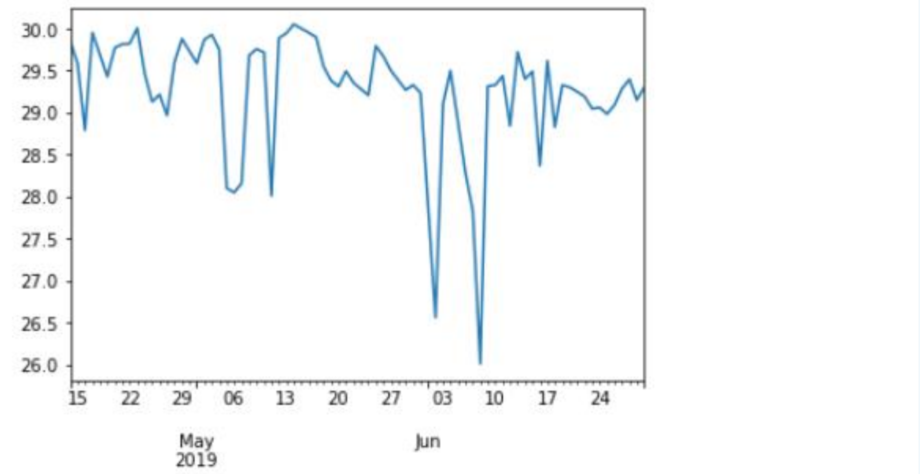
<AxesSubplot:>




```
.interpolate()
```

```
df_irregulier.Temperature.resample('D').mean().interpolate().plot()
```

<AxesSubplot:>



Access date information

- Many attributes available for a **DatetimeIndex**
  - `month`, `year`, `dayofweek`, `is_month_start`, `is_leap_year`, ...

```
df.index.month
```

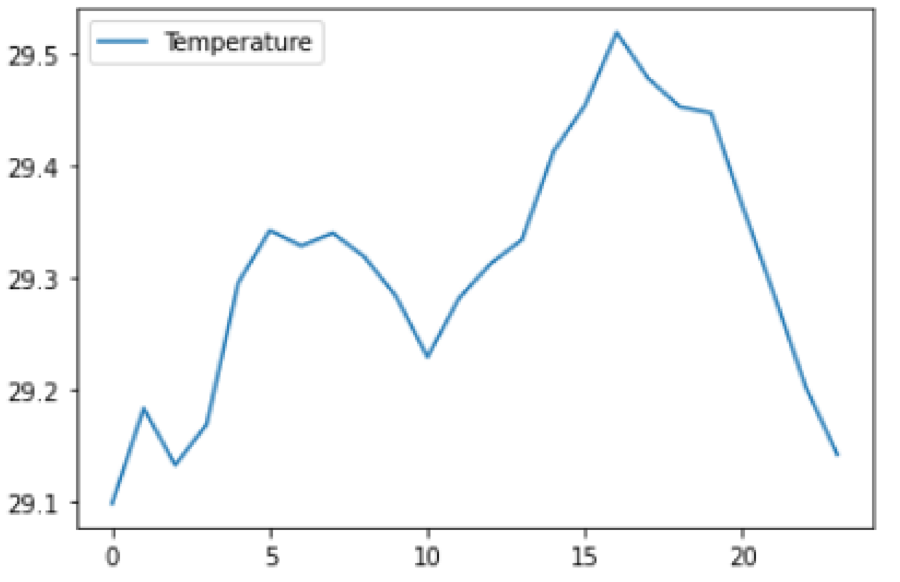
```
Int64Index([13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
            ...
            30, 30, 30, 30, 30, 30, 30, 30, 30, 30],
           dtype='int64', length=677521)
```

Groupby for temporal data?

- **Daily** temperature cycle:

```
df.groupby(df.index.hour).mean().plot()
```

```
<AxesSubplot:>
```



Part IV summary

series/dataframe with datetime index (numpy datetime/pandas date_range, …)

Some available methods:

Temporal selection: `df.loc["2012-10-04"]`

Period selection: `df.loc["2008-07" : "2015-07"]`

Resampling: `df.resample("D").mean()`

Rolling windows: `df.rolling( 5 ).mean()` `df.rolling( ‘2D’ ).mean()`

Fill the nans: `df.interpolate( )`

Datetime infos: `df.index.hour` `df.index.day_of_year`

Part IV: Practicals

Go to the jupyter notebook