Plotting

Basic Programming in Python

Sebastian Höffner Aline Vilks

Wed, 21 June 2017

Installing matplotlib

For Linux/Mac OS users `pip3 install matplotlib` should be enough.

For Windows users you should go to http://www.lfd.uci.edu/~gohlke/pythonlibs/ and download files for `numpy` and `matplotlib`. Install the files using `pip install FILES`.

Alternatively, if you used anaconda as we recommended, you can also run:

```
conda config --add channels conda-forge
conda install -c conda-forge matplotlib=2.0.2
```

All: For images, install `pillow` (again via pip or Gohlke's website).

Installing matplotlib

For Linux/Mac OS users `pip3 install matplotlib` should be enough.

For Windows users you should go to http://www.lfd.uci.edu/~gohlke/pythonlibs/ and download files for `numpy` and `matplotlib`. Install the files using `pip install FILES`.

Alternatively, if you used anaconda as we recommended, you can also run:

```
conda config --add channels conda-forge
conda install -c conda-forge matplotlib=2.0.2
```

All: For images, install `pillow` (again via pip or Gohlke's website).

Windows users: If you are unsure about the files,

- pick `cp36-cp36m` for Python 3.6 and adjust accordingly
- pick `win32` for 32 bit systems and `win_amd64` for 64 bit systems
- check 32/64 bit like this:

```
python -c import sys;print('64bit' if sys.maxsize > 2**32
else '32bit')
```

Of course you have to replace `FILES` with the paths to the files you downloaded.

First things first

You should have received an email about the course evaluation. You can either do it at home or now!

Here is the link:

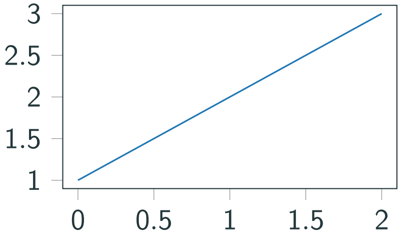
https://lehreval.psycho.uni-osnabrueck.de/evasys/indexstud.php

It is very valuable and important for me, so please take the time to answer it.

Plotting

```
plt.plot([1, 2, 3])
```

Output:



Plotting libraries

- `ggplot`: http://ggplot.yhathq.com
- `matplotlib`: https://matplotlib.org
- `vtk`: http://www.vtk.org

Plotting libraries

- `ggplot`: http://ggplot.yhathq.com
- `matplotlib`: https://matplotlib.org
- `vtk`: http://www.vtk.org

- ggplot is close to R's ggplot2 library
- matplotlib started out as a project to mimic MATLAB's plotting capabilities
- vtk is a library for 3D plots

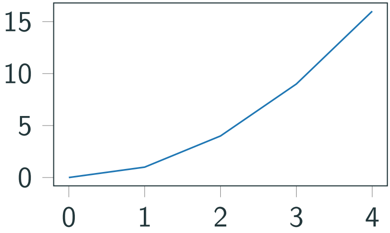
We focus on matplotlib.

Plotting data

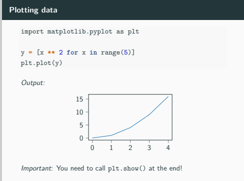
```
import matplotlib.pyplot as plt

y = [x ** 2 for x in range(5)]
plt.plot(y)
```

Output:



Important: You need to call `plt.show()` at the end!



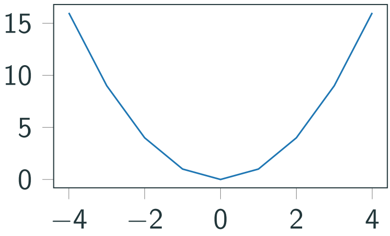
If you only supply one argument to `plot()`, it uses x from 0 to `N - 1` and assumes the data as `y` values.

For my automatic slide generation I had to leave out the call to `plt.show()`, which actually brings up the figure where we plot to. You have to add it whenever you want to see what you plotted. (There's an exception called "interactive mode", which allows you to play around with plots more natural. You can enable it with `plt.ion()`.)

```
import matplotlib.pyplot as plt

x = range(-4, 5)
y = [i ** 2 for i in x]
plt.plot(x, y)
```

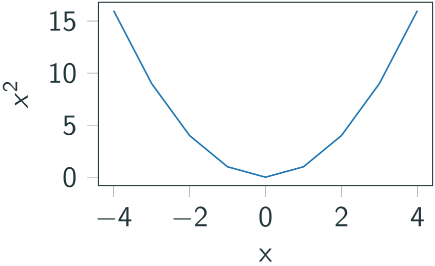
Output:



Adding labels

```
x = range(-4, 5)
y = [i ** 2 for i in x]
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('$x^2$')
```

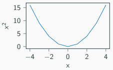
Output:



Adding labels

```
x = range(-4, 5)
y = [i ** 2 for i in x]
plt.plot(x, y)
plt.xlabel('x')
plt.ylabel('$x^2$')
```

Output:



Please note that for space reasons I remove the imports from now on. It's `import matplotlib.pyplot as plt`.

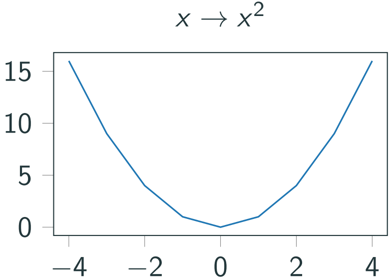
`pyplot` (which is our `plt`) always refers to the last active plot when making changes. We will see later, why this is important.

Matplotlib supports LaTeX math formulae for many labels, e.g. titles.

Adding labels

```
x = range(-4, 5)
y = [i ** 2 for i in x]
plt.plot(x, y)
plt.title(r'$x \rightarrow x^2$')
```

Output:

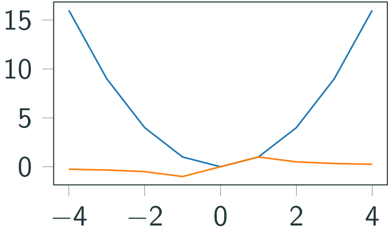


Adding multiple lines

```
x = range(-4, 5)
y1 = [i ** 2 for i in x]
x2 = [i for i in x if i != 0]
y2 = [1 / i for i in x2]

plt.plot(x, y1, x2, y2)
```

Output:



Getting a data set

Iris Data set (Fisher 1936):

```
import requests
with open('iris.data', 'w') as iris:
    iris.write(requests.get(
        'https://archive.ics.uci.edu/ml/' +
        'machine-learning-databases/iris/iris.data'
        ).text)
```

Reading iris data

File: iris_reader.py

```
def get_data():
    with open('iris.data') as iris:
        fields = ['sepal_length', 'sepal_width',
                  'petal_length', 'petal_width', 'class']
        data = list(DictReader(iris, fieldnames=fields))
        for d in data:
            for k in fields[:-1]:
                d[k] = float(d[k])
    return data

if __name__ == '__main__':
    print('\n'.join(map(str, get_data()[:3])))
```

Output:

```
OrderedDict([('sepal_length', 5.1), ('sepal_width', 3.5), ('petal_length', 1.4)
OrderedDict([('sepal_length', 4.9), ('sepal_width', 3.0), ('petal_length', 1.4)
OrderedDict([('sepal_length', 4.7), ('sepal_width', 3.2), ('petal_length', 1.3)
```

Reading iris data

File: iris_reader.py

```
def get_data():
    with open('iris.data') as iris:
        fields = ['sepal_length', 'sepal_width',
                  'petal_length', 'petal_width', 'class']
        data = list(DictReader(iris, fieldnames=fields))
        for d in data:
            for k in fields[:-1]:
                d[k] = float(d[k])
    return data

if __name__ == '__main__':
    print('\n'.join(map(str, get_data()[:3])))
```

Output:

```
OrderedDict([('sepal_length', 5.1), ('sepal_width', 3.5), ('petal_length', 1.4)
OrderedDict([('sepal_length', 4.9), ('sepal_width', 3.0), ('petal_length', 1.4)
OrderedDict([('sepal_length', 4.7), ('sepal_width', 3.2), ('petal_length', 1.3)
```

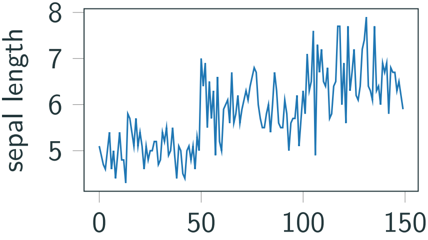
I will use this `iris_reader` on the following slides to get the data for the plots.

You can also use it to tag along.

Plotting iris data

```
y = [i['sepal_length'] for i in get_data()]
plt.plot(y)
plt.ylabel('sepal length')
```

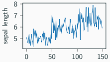
Output:



Plotting iris data

```
y = [i['sepal_length'] for i in get_data()]
plt.plot(y)
plt.ylabel('sepal length')
```

Output:

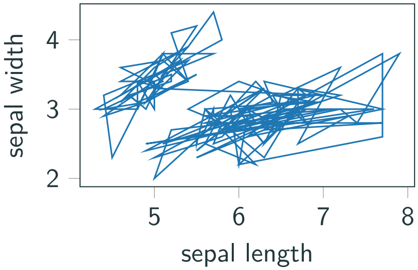


It makes not much sense to just plot the sepal lengths. Let's plot it in relation to something.

Plotting iris data

```
data = get_data()
x = [i['sepal_length'] for i in data]
y = [i['sepal_width'] for i in data]
plt.plot(x, y)
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

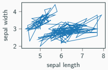
Output:



Plotting iris data

```python
data = get_data()
x = [i['sepal_length'] for i in data]
y = [i['sepal_width'] for i in data]
plt.plot(x, y)
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

Output:

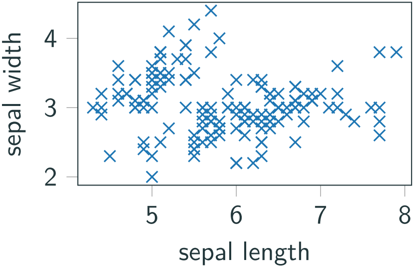


This still doesn't seem right, what should we change?

Scatter plots

```python
data = get_data()
x = [i['sepal_length'] for i in data]
y = [i['sepal_width'] for i in data]
plt.plot(x, y, 'x')  # changing "line" style
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

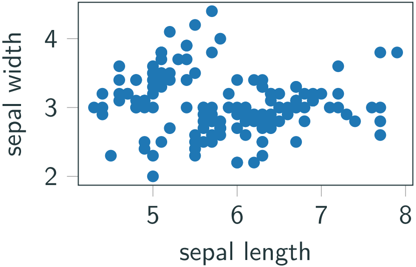
Output:



Default scatter plots

```
data = get_data()
x = [i['sepal_length'] for i in data]
y = [i['sepal_width'] for i in data]
plt.scatter(x, y)
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

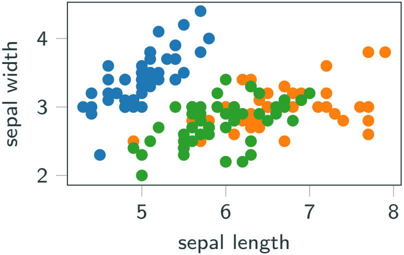
Output:



Multiple data rows

```
data = get_data()
for c in ['Iris-setosa', 'Iris-virginica', 'Iris-versicolor']:
    x = [i['sepal_length'] for i in data if i['class'] == c]
    y = [i['sepal_width'] for i in data if i['class'] == c]
    plt.scatter(x, y)
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

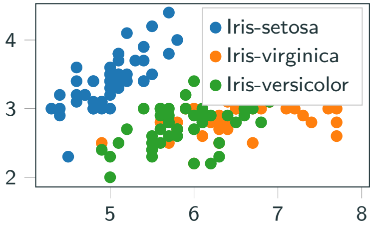
Output:

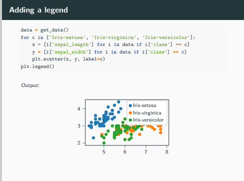


Adding a legend

```
data = get_data()
for c in ['Iris-setosa', 'Iris-virginica', 'Iris-versicolor']:
    x = [i['sepal_length'] for i in data if i['class'] == c]
    y = [i['sepal_width'] for i in data if i['class'] == c]
    plt.scatter(x, y, label=c)
plt.legend()
```

Output:



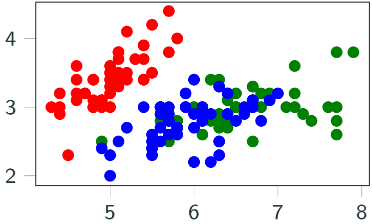


You can move the legend around using the keyword `loc`, e.g. to `'center right'` or `'lower center'`.

Changing colors

```
data = get_data()
for cl, co in zip(['Iris-setosa', 'Iris-virginica', 'Iris-versicolor'],
                  ['r', 'g', 'b']):
    x = [i['sepal_length'] for i in data if i['class'] == cl]
    y = [i['sepal_width'] for i in data if i['class'] == cl]
    plt.scatter(x, y, color=co)
```

Output:



Linear regression

Let's try to figure out how the sepal width depends on the sepal length for the Iris setosa.

We can do this with a simple linear regression:

$$y = \beta x + \alpha \tag{1}$$

$$\hat{\beta} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2} \tag{2}$$

$$\hat{\alpha} = \bar{y} - \hat{\beta}\,\bar{x} \tag{3}$$

Linear regression

File: `lin_reg.py`

```
import statistics

def linear_regression(x, y):
    mx = statistics.mean(x)
    my = statistics.mean(y)
    b = sum((xi - mx) * (yi - my) for (xi, yi) in zip(x, y))
    b /= sum((xi - mx) ** 2 for xi in x)
    return my - b * mx, b

if __name__ == '__main__':
    from iris_reader import get_data
    data = get_data()
    x = [i['sepal_length'] for i in data if i['class'] == 'Iris-setosa']
    y = [i['sepal_width'] for i in data if i['class'] == 'Iris-setosa']
    a, b = linear_regression(x, y)
    print(a, b)
```

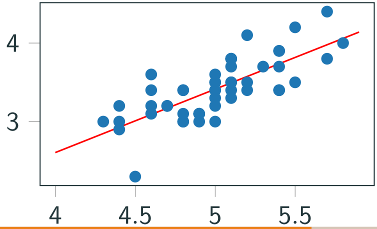
Output:

```
-0.6230117276042169 0.8072336651226961
```

Linear regression

```
data = get_data()
x = [i['sepal_length'] for i in data if i['class'] == 'Iris-seto
y = [i['sepal_width'] for i in data if i['class'] == 'Iris-setos
a, b = linear_regression(x, y)
plt.scatter(x, y)
x = [i * 0.1 for i in range(40, 60)]
y = [a + b * xi for xi in x]
plt.plot(x, y, 'r')
```

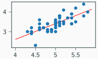
Output:



Linear regression

```
data = get_data()
x = [i['sepal_length'] for i in data if i['class'] == 'Iris-seto
y = [i['sepal_width'] for i in data if i['class'] == 'Iris-setos
a, b = linear_regression(x, y)
plt.scatter(x, y)
x = [i * 0.1 for i in range(40, 60)]
y = [a + b * xi for xi in x]
plt.plot(x, y, 'r')
```

Output:



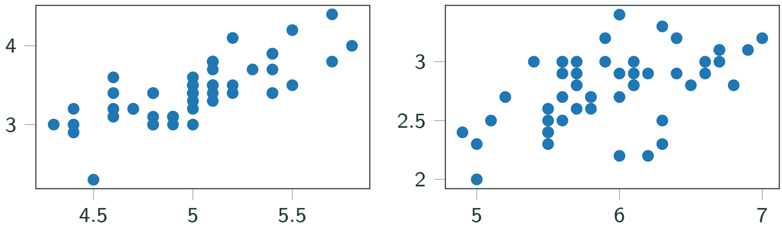
You can draw multiple plots into one “plot”.

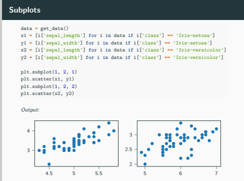
Subplots

```
data = get_data()
x1 = [i['sepal_length'] for i in data if i['class'] == 'Iris-setosa']
y1 = [i['sepal_width'] for i in data if i['class'] == 'Iris-setosa']
x2 = [i['sepal_length'] for i in data if i['class'] == 'Iris-versicolor']
y2 = [i['sepal_width'] for i in data if i['class'] == 'Iris-versicolor']

plt.subplot(1, 2, 1)
plt.scatter(x1, y1)
plt.subplot(1, 2, 2)
plt.scatter(x2, y2)
```

Output:



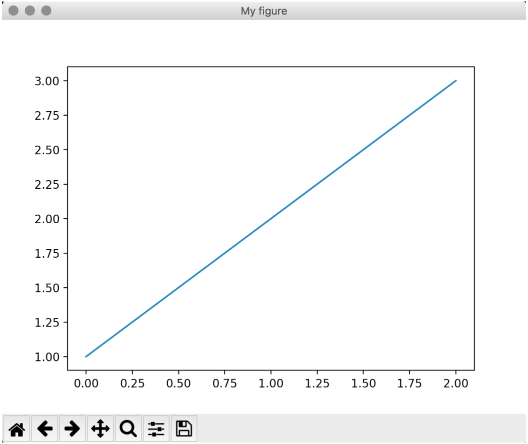


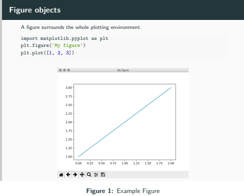
You can also specify subplots to span multiple "cells", but it gets tricky. E.g. a subplot which spans the second row would look like this: `plt.subplot(2, 1, 2)` (2 rows, 1 column, second position).

Figure objects

A figure surrounds the whole plotting environment.

```
import matplotlib.pyplot as plt
plt.figure('My figure')
plt.plot([1, 2, 3])
```





This does not work well with the auto slide creation, hence I included a screenshot.

The name `My figure` does not only set the title, it is also a unique identifier to reactivate the figure.

Using a figure window

From left to right:

- Home view (reset views, return to initial view)
- Previous view
- Next view
- Pan (move around the plot)
- Zoom (zooms to a user drawn rectangle)
- Subplot Configuration Tool (allows to change e.g. margin around plots)
- Save (save the plot as png)

Using a figure window


From left to right:

- Home view (reset views, return to initial view)
- Previous view
- Next view
- Pan (move around the plot)
- Zoom (zooms to a user drawn rectangle)
- Subplot Configuration Tool (allows to change e.g. margin around plots)
- Save (save the plot as png)

Depending on the “backend” your matplotlib uses, these might be slightly different in style or behavior.

A backend is, in a simplified fashion, the software your matplotlib uses to create windows and draw into them. There are also backends which can only create file outputs.

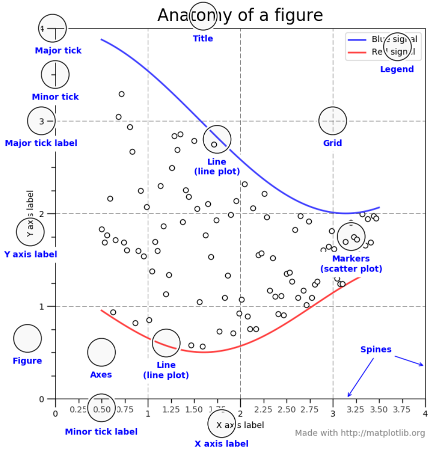


Figure 2: Parts of a Figure, Matplotlib FAQ

Parts of a Figure

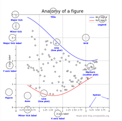


Figure 2: Parts of a Figure, Matplotlib FAQ

Everything inside the window is the `canvas`, the most important part of the figure.

All elements you can see here (except for “figure”) are drawn onto the canvas.

Object-oriented interface

Each call we made returned some objects!

```
figure = plt.figure('My figure')
lines = plt.plot([1, 2, 3])
print(figure)
print(lines)
```

Output:

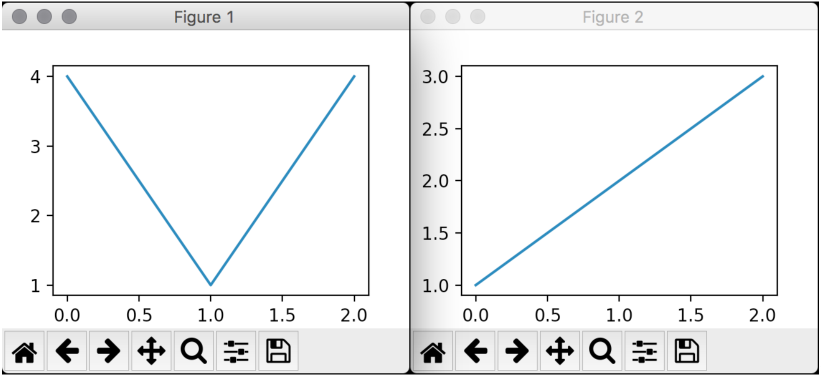
```
Figure(640x480)
[<matplotlib.lines.Line2D object at 0x10be265f8>]
```

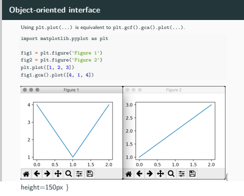
Object-oriented interface

Using `plt.plot(...)` is equivalent to `plt.gcf().gca().plot(...)`.

```
import matplotlib.pyplot as plt

fig1 = plt.figure('Figure 1')
fig2 = plt.figure('Figure 2')
plt.plot([1, 2, 3])
fig1.gca().plot([4, 1, 4])
```





`gcf` means “get current figure”, `gca` means “get current axes”

Each figure has an initial pair of axes (“x” and “y”) which can be selected for drawing.

Advantages of the Object-oriented interface

- Keeping references to different axes objects allows changing data in continuous programs
- It becomes easier to keep track to which figure is plotted
- We can build interactive figures much easier

Changing data of a plot

File: scatter_pause.py

```
import matplotlib.pyplot as plt
from iris_reader import get_data

data = get_data()  # iris data again
x = [i['sepal_length'] for i in data if i['class'] == 'Iris-setosa']
y = [i['sepal_width'] for i in data if i['class'] == 'Iris-setosa']

fig, ax = plt.subplots(1, 1, num='Iris setosa')
scatter = ax.plot([], [], 'ko')[0]
ax.set_xlim([4, 6])
ax.set_ylim([2, 4.5])

plt.ion()
plt.show()

for xi, yi in zip(x, y):
    scatter.set_data(xi, yi)  # Set data
    fig.canvas.draw()  # Update complete canvas
    plt.pause(0.25)  # Pause until next frame
```

Changing data of a plot

File: scatter_pause.py

```
import matplotlib.pyplot as plt
from iris_reader import get_data

data = get_data()  # iris data again
x = [i['sepal_length'] for i in data if i['class'] == 'Iris-setosa']
y = [i['sepal_width'] for i in data if i['class'] == 'Iris-setosa']

fig, ax = plt.subplots(1, 1, num='Iris setosa')
scatter = ax.plot([], [], 'ko')[0]
ax.set_xlim([4, 6])
ax.set_ylim([2, 4.5])

plt.ion()
plt.show()

for xi, yi in zip(x, y):
    scatter.set_data(xi, yi)  # Set data
    fig.canvas.draw()  # Update complete canvas
    plt.pause(0.25)  # Pause until next frame
```

We call each update a “frame”.

`canvas.draw()` forces the canvas to redraw everything on it.

Using animations

File: scatter_animation.py

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from iris_reader import get_data

data = get_data()  # iris data again
x = [i['sepal_length'] for i in data if i['class'] == 'Iris-setosa']
y = [i['sepal_width'] for i in data if i['class'] == 'Iris-setosa']

fig, ax = plt.subplots(num='Iris setosa')
scatter = ax.plot([], [], 'ko')[0]
ax.set_xlim([4, 6])
ax.set_ylim([2, 4.5])

def update(frame_number):
    scatter.set_data(x[frame_number], y[frame_number])
    return scatter,  # Note the comma!

ani = animation.FuncAnimation(fig, update, range(len(x)),
                              interval=250, blit=True)
plt.show()
```

Using animations

File: scatter_animation.py

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from iris_reader import get_data

data = get_data()  # iris data again
x = [i['sepal_length'] for i in data if i['class'] == 'Iris-setosa']
y = [i['sepal_width'] for i in data if i['class'] == 'Iris-setosa']

fig, ax = plt.subplots(num='Iris setosa')
scatter = ax.plot([], [], 'ko')[0]
ax.set_xlim([4, 6])
ax.set_ylim([2, 4.5])

def update(frame_number):
    scatter.set_data(x[frame_number], y[frame_number])
    return scatter,  # Note the comma!

ani = animation.FuncAnimation(fig, update, range(len(x)),
                              interval=250, blit=True)
plt.show()
```

A FuncAnimation takes a figure and an update function. The third parameter is the frame numbers, they are passed to the update function in turn. The interval is the time in milliseconds between two frames.

The update function should return all “artists” (plot elements) which should be updated.

The return value is important for “`blitting`”. It can give you immense speed ups if you have complex figures: It only updates what changed, not the complete canvas. To disable it, just pass `blit=False` to the `FuncAnimation`.

Important: Usually one would put everything into a class and not into the global scope as I did here!

Interactive figures: callback functions

We just used a callback function!

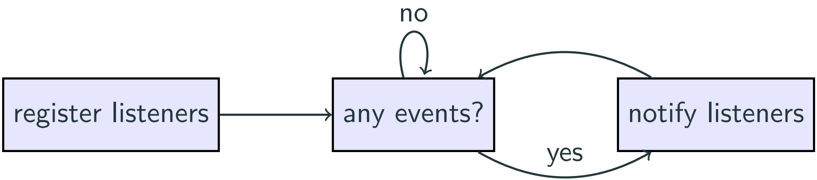
That means: we passed a function to another function (or class) to have it called by them!

```
def update(frame_number):
    scatter.set_data(x[frame_number], y[frame_number])
    return scatter,

ani = animation.FuncAnimation(fig, update, range(len(x)),
                              interval=250, blit=True)
```

Interactive figures: event loop

GUIs[1] have an event loop:



[1] Graphical User Interface

Interactive figures: Connecting with matplotlib

File: `drawing.py`

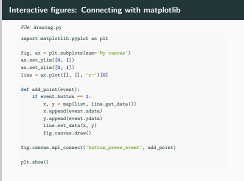
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(num='My canvas')
ax.set_ylim([0, 1])
ax.set_xlim([0, 1])
line = ax.plot([], [], 'r-')[0]

def add_point(event):
    if event.button == 1:
        x, y = map(list, line.get_data())
        x.append(event.xdata)
        y.append(event.ydata)
        line.set_data(x, y)
        fig.canvas.draw()

fig.canvas.mpl_connect('button_press_event', add_point)

plt.show()
```



You can find a list of all available events as well as some nice examples here: https://matplotlib.org/users/event_handling.html

References

Fisher, R. A. 1936. “The Use of Multiple Measurements in Taxonomic Problems.” *Annals of Human Genetics* 7 (2): 179–88.