7 Plotting with Python

Visualisation is a universal method for investigating a dataset or verifying results of computations. Several modules for producing plots and charts are available in Python. One of the most popular general-purpose tools is `matplotlib`, a powerful library allowing, among others, to generate 2D and 3D graphs.

In this section, the programmatic approach to producing plots will be described. The programmatic approach uses code to create graphs.

7.1 Matplotlib module

`Matplotlib` provides multiple entry points to the library with different programming interfaces: a stateful API (application programming interface) and an object-oriented API. Both are provided by the sub-module `matplotlib.pyplot`. Stateful API is simple and suitable for small examples, but context-dependency makes it hard to reuse the code. Matplotlib documentation itself[1] recommends using the stateful approach mostly for quick plotting and Jupyter Notebooks. It is therefore recommended to use the object-oriented API in scripts and reusable functions.

Similar to `NumPy`, `matplotlib` is not included in core Python installation (requires a seperate installation), but it is one of Anaconda packages. It is established, that `matplotlib.pyplot` is imported with an alias `plt`. While working in JupyterNotebooks, it is important to execute a line `% matplotlib inline` enabling displaying a graph in the notebook rather than a new window.

The crucial part of a graph is a `Figure` instance and one or more `Axes` instances within the `Figure`. The `Figure` is the main object that includes all the other elements and provides canvas area for drawing. `Axes` can be interpreted as individual plots with some data presented: coordinate systems assigned to fixed regions of the figure canvas. The figure canvas has a simple coordinate system, which is used when placing elements such as axes, with (0, 0) in lower-left corner and (1, 1) in a upper-right. A `Figure` can contain multiple `Axes` instances, but `Axes` can belong to only one `Figure`. `Axes` can be placed on canvas manually or added automatically. In addition to providing a coordinate system for plotting data, an `Axes` instance determines how the elements of a graph (axis, ticks) are displayed.

An `Axes` usually contains `Axis` (easy to confuse!). `Axis` represent a number line on the plot and take care of the data limit (think of x, y and z axis on a plot). `Axis` set limits of the plot and generate ticks (marks with numbers on the plot lines). Overall, everything that appears on canvas, a line, a marker, an `Axes`, an `Axis` and even `Figure` is an object called `Artist`.

A standard plotting procedure requires a computation of a number of points lying on the plotted curve and then drawing a line between them. If we have enough data points, the resulting plot looks like a smooth curve. To store coordinates of the points on the plot, it is recommended to use an `array` introduced in the previous section, which are more efficient than regular lists. When the data for the graph is prepared, a `Figure` and `Axes` instances are created, then the plot method on the `Axes` instance is used and the basic properties of a graph are set. Example `[1]` uses an object-oriented approach. The same result can be obtained with a stateful approach presented in example `[2]` (notice that the `Figure` instance is not created explicitly). The result is presented in Figure 7.1.

[1] `https://matplotlib.org/stable/tutorials/introductory/usage.html`

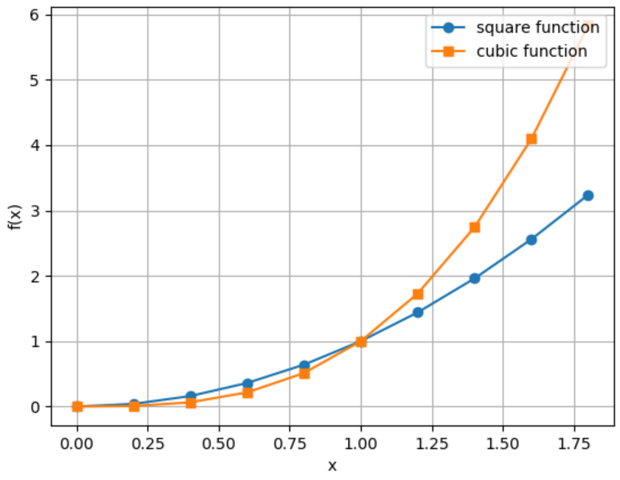

Figure 7.1. Graph produced with code in examples `[1]` and `[2]`

In [1]:
```python
import matplotlib.pyplot as plt # popular import statement for matplotlib.pyplot
import numpy as np

x = np.arange(0, 2, 0.2) # data for the plot
y1 = x ** 2
y2 = x ** 3
fig, ax = plt.subplots()     # creating instance of Figure and Axes
ax.plot(x, y1, "o-", label="square function")
ax.plot(x, y2, "s-", label="cubic function")
ax.set_xlabel('x')           # add label to x axis
ax.set_ylabel ('f(x)')  # add label to y axis
ax.grid(True)        # add coordinate grid
ax.legend(loc=1)     # add legend in location 1, labels added to curves
```

In [2]:
```python
import matplotlib.pyplot as plt
from numpy import arange

x = arange(0, 2, 0.2)
plt.plot(x, x**2, 'o-', x, x**3, 's-')  # plot two functions with specified marker]
plt.xlabel('x')       # add label to x axis
plt.ylabel ('f(x)')   # add label to y axis
plt.grid(True)        # add coordinate grid
plt.legend(('square function', 'cubic function'), loc=1)  # add legend in location 1
plt.show()            # show plot on screen
```

In example `[1]`, `plt.subplots()` function was used to create a `Figure` and `Axes` instance. This function can also be used to create grids of `Axes` objects within a `Figure` by giving values to `ncols` and `nrows` parameters. Once the `Axes` object is available, all the remaining operations regarding a graph are invoked on the `Axes` instance. A graph in Figure 7.1 is readable, but there is room for improvement. Almost any component of the graph – axes, titles, grids, ticks, font and fontsize – can be modified according to user's needs.

The graphical user interface (GUI) for displaying a graph is useful for interactive use with Python script files, it allows an interaction with a figure with zooming and panning. While using an interactive backend, it is necessary to call `plt.show()` method to show a graph window on the screen. Using an `inline` backend in JupyterNotebooks, the interactive aspect of the GUI is lost, but embedding the image in the notebook has an advantage of keeping the code together with its result and eliminating the need to rerun the code to display a figure. It is then not necessary to use `plt.show()` in Jupyter inline backend. Examples in this section are assumed to be run in `Jupyter Notebooks`, hence `plt.show()` is be omitted. In an interactive environment the call `plt.show()` is needed.

7.2 Figure and Axes

In `matplotlib`, `Figure` object is used to represent a graph. Besides providing a canvas on which `Axes` are placed, the `Figure` object has methods and attributes for performing actions on figures and modifying its properties.

A `Figure` object can be created using `plt.Figure` function which accepts keyword argument such as `figsize` (`(width, height)` specifying canvas size in inches) or `facecolor` (defining the color of figure canvas). When a `Figure` is created, the `Axes` instances can be added using `add_axes` method (example `[3]`). `add_axes` requires a list, containing the coordinates of the lower-left corner, the width and the height of the axes, as an argument. The `Axes` width and height are expressed as fractions of total canvas width and height. To set an overall figure title, a `fig.suptitle` method is used. To save a graph to a file, use `plt.savefig` function. `plt.savefig` enables saving a file in many formats, for example, `png`, `pdf`, `eps`, `svg`. The resolution can be set with a `dpi` argument.

In [3]:
```
x = arange(0, 2, 0.2) # data for the plot
y1 = x ** 2
y2 = x ** 3

fig = plt.Figure(figsize=(8, 4))
ax = fig.add_axes((0.1, 0.1, 0.8, 0.8))     # centered Axes
ax.plot(x, y1, "o-", label="square function")
ax.plot(x, y2, "s-", label="cubic function")
ax.set_xlabel('x')           # add label to x axis
ax.set_ylabel ('f(x)')  # add label to y axis
ax.legend(loc=0)      # add legend in location 1, labels added to curves
plt.savefig('example_plot.png', format='png', dpi=100) # save plot to png file
```

Most of the graph options are available as methods of an `Axes` object. `Axes` provides the coordinate system for plotting data and mathematical functions, contains the `Axis` objects, determines there the axis labels and ticks are placed. Drawing different types of plots are also implemented as `Axes` methods.

To create a grid of `Axes` instances, a function `plt.subplots(ncols=n, nrows=m)` can be used, which takes as arguments number of rows and number of columns in axes grid. The function returns a tuple `(fig, axes)`, where `axes` is a `numpy` array of size `ncols, nrows`. We can also specify that the `Axes` grid shares x or y axis, using `sharex` and `sharey` with values set to `True` or `False`.

7.3 Graph types

`Matplotlib` provides many types of plotting methods. All plotting functions expect an input given as `numpy` arrays. Common graph types include:

- scatter plot,
- histogram,
- log and semilog plot,

Table 7.1. Properties of graphs

Argument	Description	Example values
`alpha`	amount of transparency	value between 0.0 (transparent) and 1.0 (completely opaque)
`color`	colour specification	string with color name (`"blue"`, `"yellow"`), RGB code (`"#aabbaa"`)
`linestyle`, `ls`	style of a line	`'-'` – solid line, `'--'` – dashed line, `':'` – dotted line, `'-.'` dash-dot line
`linewidth`, `lw`	width of a line	float number
`marker`	each data point can be represented with a marker	`'o'` – circle marker, `'`ˆ, `1`, `2`, . . . – triangle marker, `'*'` – star marker, `'+'` – cross marker,, `'.'` – small dot marker,`'s'` – square marker, `'h'` – hexagon marker, `'x'` – x marker
`markersize`	marker size	float number
`markerfacecolor`	fill colour of a marker	colour specification
`markeredgewidth`	line width of a marker edge	float number
`markeredgecolor`	colour of the marker edge	colour specification

- bar plot,
- pie chart,
- polar plot.

7.4 Plot styles

Properties of each graph, such as line colour or line width, can be configured with a set of keyword arguments. Many plotting methods have their specific arguments, but basic properties are shared among the plotting methods. These properties are listed in Table 7.1.

Using different colours or line types is important to make different lines readable and distinctive. However, the colors of subsequent plots on the same axes, if not defined, are changed automatically.

Another important option that makes graphs identifiable is a legend. A legend can be added to `Axes` with a `legend` method. Only lines with an assigned labels are included in a legend description. Location of the legend can be easily changed by providing one of the numbers 0–4 to an argument `loc`. If no option is included, the location is chosen automatically. A legend can be placed outside the `Axes` object, in an arbitrary place of canvas with a method `bbox_to_anchor`.

7.4.1 Axes properties

Methods to change basic properties of `Axes` object include:

`ax.set_xlabel`, `ax.set_ylabel` – sets a label for a x/y axis,

`ax.set_xlim`, `ax.set_ylim` – changes the limits of coordinates on the x/y axis,

`ax.set_title` – sets title for a `Axes` instance,

`ax.grid` – turn on grid and control its appearance,

`ax.set_xticks`, `ax.set_yticks` – changes ticks on the x / y axis,

Fundusze Europejskie Wiedza Edukacja Rozwój
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny

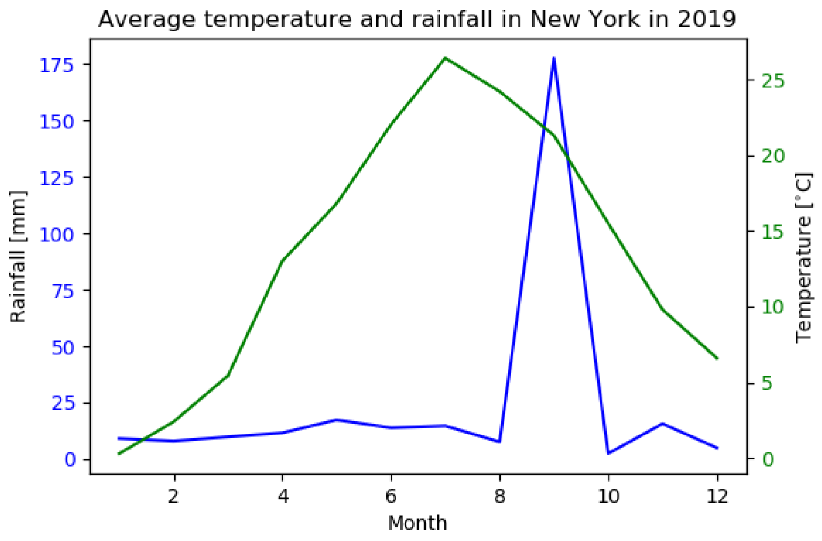

Figure 7.2. Twin axes feature (example [4])

`ax.xaxis.set_major_locator`, `ax.xaxis.set_minor_locator` – set major or minor ticks choosing one of the placement strategies from `mpl.Ticker` module.

An interesting feature of `matplotlib` is the twin axis, which allow displaying two independent axes overlaid on each other. This option is useful in plotting two different quantities, usually with different units, sharing one of the axes, within the same graph. In example [4], `twinx` method was used, but `twiny` is also available. The result is presented in Figure 7.2

In [4]:
```python
m = np.arange(1, 13)
temp = np.array([0.3, 2.4, 5.44, 13.0, 16.8, 22.0, 26.4, 24.2, 21.3, 15.5, 9.8,
    6.6])
rain = np.array([9.09, 7.98, 9.83, 11.58, 17.32, 13.84, 14.68, 7.62, 177.80, 2.41,
    15.65, 4.95])

fig, ax = plt.subplots(figsize=(6, 3))
ax.set_title("Average temperature and rainfall in New York in 2019")
ax.set_xlabel("Month")
ax.set_ylabel("Rainfall [mm]")
ax.plot(m, rain, color="blue")
for label in ax.get_yticklabels(): #coloring ticks
    label.set_color("blue")

ax2 = ax.twinx()
ax2.plot(m, temp, color="green")
ax2.set_ylabel(r"Temperature [$^{\circ}$C]")
for label in ax2.get_yticklabels():
    label.set_color("green")
```

7.4.2 Text in graphs

To check the current configuration of fonts in the graph, print `mpl.rcParams`. Updating a parameter requires assigning a new value to the corresponding item in a dictionary `mpl.rcParams`. Text properties can also be set from within the plotting function by passing a keyword arguments:

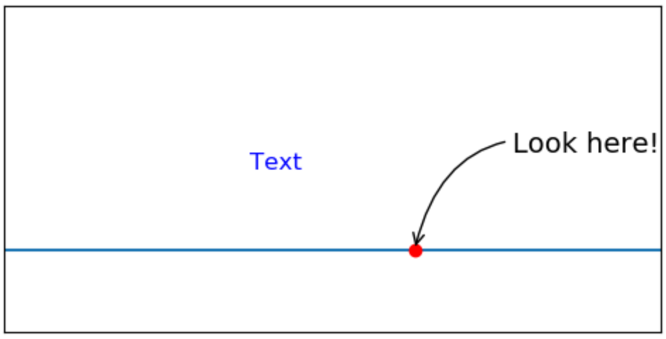

Figure 7.3. Adding text and annotations to figure (example `[5]`)

`fontsize` – size of font in points,

`color` – colour specification,

`family` – font type,

`alpha` – transparency of the font colour,

`backgroundcolor` – colour specification for text background,

`rotation` – rotation angle of the text.

`Matplotlib` provides support for LaTeX markup within text labels. LaTeX math should be enclosed with $ signs. To prevent interpreting \characters in LaTeXmarkup as escape characters by Python, it is advised to use raw strings, which are string expressions prefixed with a letter `r`, for example: `r"`$f(x) = \backslash frac\{1\}\{2\}x$`"`.

To add text and annotations to a figure, functions `ax.text` and `ax.annotate` should be used (example `[5]`, Figure 7.3).

In [5]:
```python
fig, ax = plt.subplots(figsize=(6, 3))
ax.set_yticks([])     # no ticks
ax.set_xticks([])
ax.set_xlim(-1, 3)
ax.set_ylim(-0.1, 0.3)
ax.axhline(0)  # horizontal line at y=0
ax.plot(1.5, 0, "ro")
ax.text(0.5, 0.1, "Text", fontsize=12, color="blue") # text label
ax.annotate("Look here!", fontsize=14, xy=(1.5, 0), xycoords="data", xytext=(+50,
    +50), textcoords="offset points", arrowprops=dict(arrowstyle="->",
    connectionstyle="arc3, rad=.5"))
```

7.5 Subplots

To create more than one plot in a figure, using a `subplots` method is necessary (example `[6]`, Figure 7.4).

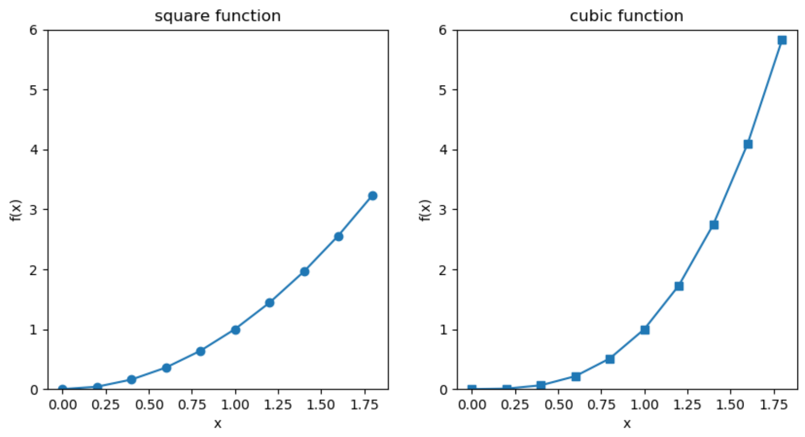

Figure 7.4. Creating subplots in a figure (example [6])

In [6]:
```python
x = np.arange(0, 2, 0.2)
fig, ax = plt.subplots(1,2) # create a plot in two rows and one column of plots,
    make it a current plot, commas may be omitted
ax[0].plot(x, x**2, 'o-')
ax[0].set_xlabel('x')        # add label to x axis
ax[0].set_ylabel ('f(x)')    # add label to y axis
ax[0].set_ylim([0, 6])
ax[0].set_title('square function')  # add title

ax[1].plot(x, x**3, 's-')
ax[1].set_xlabel('x')        # add label to x axis
ax[1].set_ylabel ('f(x)')    # add label to y axis
ax[1].set_title('cubic function')   # add title
ax[1].set_ylim([0, 6])
plt.show()             # show plot on screen
```

7.6 Exercises

Exercise 7.1. Define two lists, `y` and `x` of length 10, containing numeric data.

1. Use the defined lists to plot `y(x)`. Adjacent points should be matched with a line. Add a title to a plot and describe the axis.
2. Change the figure size to `width=13` and `height=8`.
3. Change the markers for data points to green triangles.
4. Use the defined lists to plot only datapoints, not connected with a line, use `scatter`. Datapoints should be marked with black circles.

Exercise 7.2. 1. Plot two datasets on one plot, use only one `plot` function call. The first dataset is $y(x)$ from exercise 7.1, the second is a function $f(x)$:

$$f(x) = x^2 - 5$$

$f(x)$ should be computed for 9 equally spaced points in the range $< 2, 8 >$.

2. Plot the same data as in point 1, add legend. This time, use two `plot` function calls. You can modify the colour and width of the lines if you want.

Exercise 7.3. Crate a figure with two subfigures. In the first subfigure plot the information from the previous task, in the second subfigure plot a quadratic function (x^2) in range $x \in< -5, 5 >$. Add a title to both subfigures and a title to the whole set (`suptitle`).

1. Put subfigures next to each other (two columns); use `add_subplot`;
2. Put subfigures next to each other (two columns); use `add_subplot`;
3. Do the task from points 1 and 2, using `plt.subplots` or `plt.subplot()`

Exercise 7.4. • Create a plot bar including nutrition information in a chosen product. For example, 1 lemon covers 90% of the vitamin C daily value and 12% of dietary fiber. Change the color of the bars. Include data description.

• Create the same plot, but put bars horizontally.

Nutrition information can be found, for example in WolframAlpha (1 apple): `https://www.wolframalpha.com/input/?i=1+apple`

Exercise 7.5. Create a histogram from randomly generated data (at least 1000 data points). Save a figure to a file. Try using different file extensions. Add a keyword argument `bins`, divide data into 10, 20 and 50 bins.

Exercise 7.6. The sine function can be approximated by a Taylor series according to the formula:

$$\sin x \approx S(x,n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!}.$$

The error in the approximation $S(x,n)$ decreases as n increases. Visualize the quality of $S(x,n)$ as n increases together with a sine function on $[0,\ 4\pi]$. Fill the space between two plots with a colour.

7.7 Useful links

- **[Recommended]** Basic usage: `https://matplotlib.org/stable/tutorials/introductory/usage.html`
- Lifecycle of a plot (some plot customisation): `https://matplotlib.org/stable/tutorials/introductory/lifecycle.html`
- SciPy lectures on plotting: `http://scipy-lectures.org/intro/matplotlib/index.html`
- Ten simple rules for better plots: `https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003833`
- Introduction to Matplotlib: `https://realpython.com/python-matplotlib-guide/` (REquires logging in, but it is possible to read the article in Private / Incognito mode of the browser)
- Matplotlib backend: `https://www.aosabook.org/en/matplotlib.html`
- Matplotlib documentation and tutorial:
  - Pyplot `https://matplotlib.org/tutorials/introductory/pyplot.html`
  - Differences between Pyplot API and Object-orineted API: `https://matplotlib.org/tutorials/introductory/lifecycle.html#the-lifecycle-of-a-plot`
  - `https://matplotlib.org/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-us`

References

Beazley, D. M. (2009). *Python essential reference*. Addison-Wesley Professional.

Ceder, N. (2018). *The quick Python book*. Simon and Schuster.

Downey, A. B. (2016). *Think Python. How to Think Like a Computer Scientist*. O'Reilly Media, 2nd edition.

Lubanovic, B. (2014). *Introducing Python: Modern Computing in Simple Packages*. O'Reilly Media, Inc.

Lutz, M. (2013). *Learning Python*. O'Reilly Media, 5th edition.

Python (2021). Python 3.10.0 documentation. `https://docs.python.org/3/` [Accessed 17 October 2021].

Fundusze Europejskie Wiedza Edukacja Rozwój
Politechnika Warszawska
Unia Europejska Europejski Fundusz Społeczny