File I/O & Plotting

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

BCCMS
Bremen Center for Computational Materials Science

Universität Bremen

https://www.bccms.uni-bremen.de/people/b-aradi/wissen-progr/python/2023

Outline

- Reading and writing files

- Plotting with matplotlib

You might need to install matplotlib and scipy in your Miniconda installation to try the examples

```
conda install matplotlib scipy
```

File I/O workflow

- Open file
- Do read/write operations
- Close file

```
fp = open("test.txt", "r")
txt = fp.read()
fp.close()
```

- The closing of a file is optional (although recommended)
- Using context manager blocks by

  with … **as** …

  the file can be closed automatically
- File closed upon exiting the context manager block

```
with open("test.txt", "r") as fp:
    txt = fp.read()
print("File closed automatically")
```

Reading text from a file

```
with open("test.txt", "r") as fp:
...
```

- **Iterating** over file handler returns the lines in the file as strings (including the newline character a the line ends):

```
for line in fp:
    print(line)
```

- The **readlines()** method returns a list of the lines in the file:

```
lines = fp.readlines()
print(lines)
```

- The **readline()** method returns the next line in the file (and empty string if all lines had been read):

```
line = fp.readline()
while line:
    print(line)
    line = fp.readline()
```

- The **read()** method returns the entire file content as one string:

```
txt = fp.read()
print(txt)
```

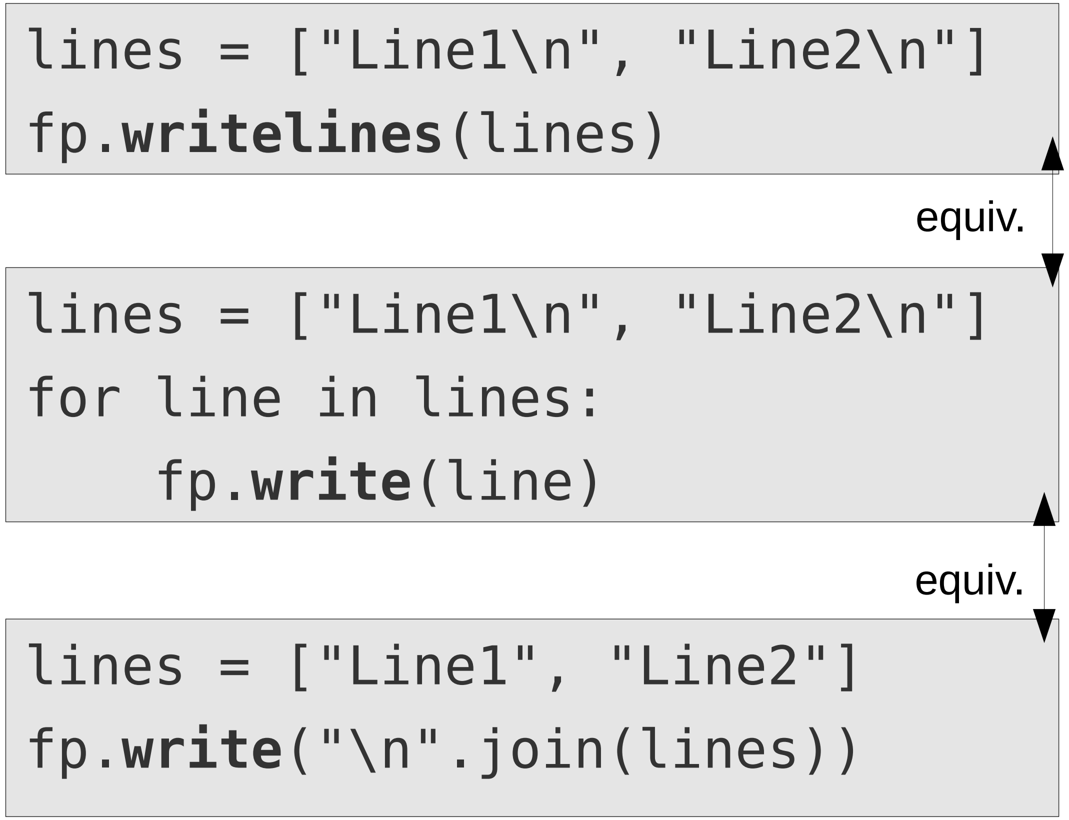
Writing text to a file

```
with open("test.txt", "w") as fp:
...
```

- The **write()** method writes a given string into a file

- The **writelines()** method writes a list of strings into a file

```
fp.write("Line 1\n")
```



```
lines = ["Line1\n", "Line2\n"]
fp.writelines(lines)
```

equiv.

```
lines = ["Line1\n", "Line2\n"]
for line in lines:
    fp.write(line)
```

equiv.

```
lines = ["Line1", "Line2"]
fp.write("\n".join(lines))
```

Reading / writing arrays

- Numpy/Scipy have special routines to read/write data arrays in text form (and also in other formats)

numpy.loadtxt()	Reads data from a file into an array
numpy.savetxt()	Writes array data into a file

test.dat:

```
# Some comment
1  2
3 4
```

```
data = np.loadtxt("test.dat")
data
```

```
array([[ 1.,  2.],
       [ 3.,  4.]])
```

```
data2 = np.array([1, 2, 3])
np.savetxt("test2.dat", data2)
```

→

test2.dat

```
1.000000000000000000e+00
2.000000000000000000e+00
3.000000000000000000e+00
```

Path manipulation (os.path)

os.path module

- Module with very helpful functions for file name and path manipulations
- **os.path.join()**: Joining path names:

```
import os.path

directory = "schroedinger/harmonic"
fname = "energies.dat"
fname_full = os.path.join(directory, fname)
fname_full
'schroedinger/harmonic/energies.dat'
```

See also: os.path module documentation

Path manipulation (pathlib)

pathlib module

- Object oriented path handling methods
- Path object offers methods and overriden operators to query and manipulate paths

```
from pathlib import Path
```

```
directory = Path("dir1/dir2")
```

```
PosixPath('dir1/dir2')
```

Path-object

```
fname = "data.dat"
```

```
'data.dat'
```

String

```
fname_full = directory / fname
```

```
PosixPath('dir1/dir2/data.dat')
```

- Path object can be used in the **open()** statement instead of string file name

```
file = Path("test.dat")
with open(file, "r") as fp:
    fp.read()
```

See also: pathlib module documentation

Plotting with matplotlib

Matplotlib interfaces

- Fully object oriented interface (should be favored)
- Matlab-like simplified interface with global state

Matplotlib render engines

- Embedding plots into the IPython/Jupyter notebook

```
%matplotlib inline
```

In JupyterLab this is already the default

- Showing plots in separate windows (when using from script or from IPython-console
- Creating graphical files (pdf, jpg, etc.)

Self-containing plotting example

```
import numpy as np
import matplotlib.pyplot as plt

xx = np.linspace(0.0, 4.0 * np.pi, 200, endpoint=True)
y1 = np.cos(xx)
y2 = np.sin(xx)

fig, ax = plt.subplots()
ax.plot(xx, y1, color='red', linewidth=1.0, linestyle="--", label='cos(x)')
ax.plot(xx, y2, color='blue', linewidth=1.0,linestyle="-", label='sin(x)')
ax.legend()
plt.show()
```

Generating x/y values

Create Figure and Axes objects (multiple subplots possible)

Plot curves through given x/y values

Create legend box

Render plot/figure (optional in JupyterLab)

- If you do not use **plt.show()** in Jupyter, append semicolon (“**;**”) to last line of the cell to suppress additional non-graphical output

- If you use a GUI-backend, you can also use **fig.show()** to render a figure

Self-containing plotting example

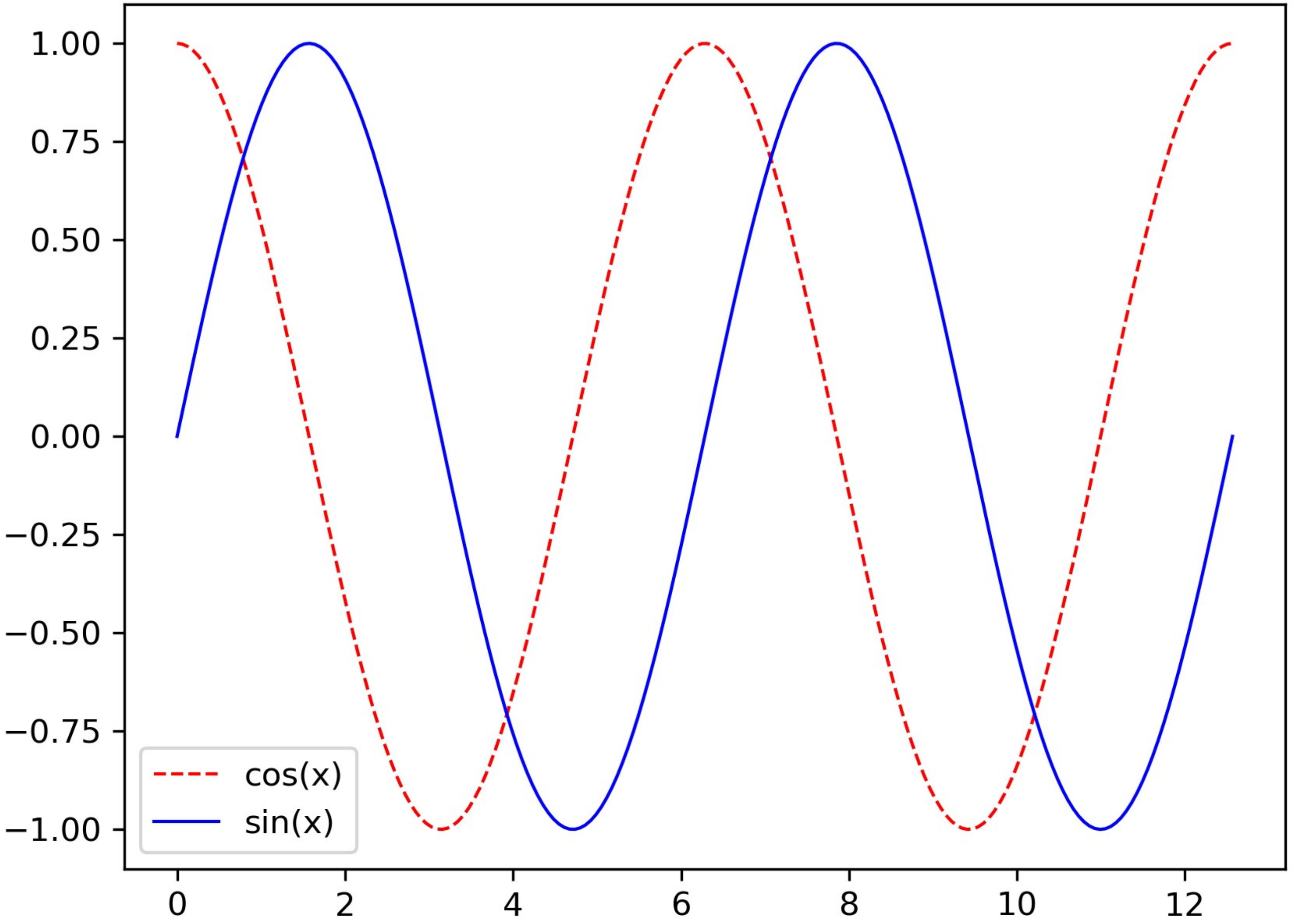


Figure and Axes objects

Figure

- A Figure object instance represents the figure
- Figure objects enables to manipulate the global figure parameters or to execute global actions

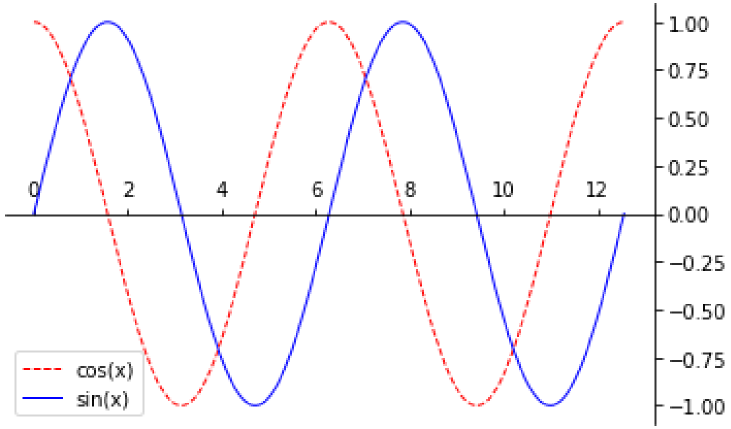
```
fig.set_size_inches(10, 8)
fig.set_dpi(300)
```

```
fig.savefig('plot.pdf')
```

Axes

- An Axes-object instance represents one plot within the figure
- Axes-object enables very detailed tuning of the resulting plot

```
ax.xaxis.set_ticks_position('top')
ax.yaxis.set_ticks_position('right')
ax.spines['top'].set_position(('data', 0))
ax.spines['bottom'].set_color('none')
ax.spines['left'].set_color('none')
```



Mulitple subplots

- The **subplots()** command can create multiple subfigures
- It returns individual Axes objects (one for each subfigure)

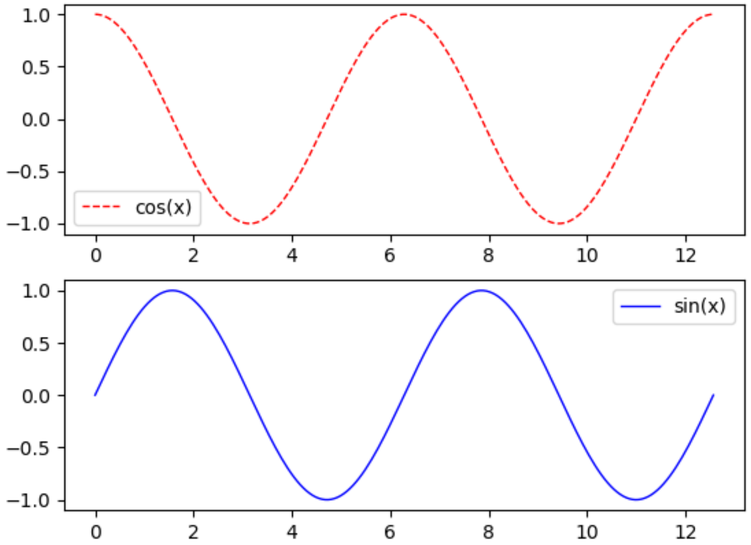
```
fig, (ax1, ax2) = plt.subplots(2, 1)

ax1.plot(xx, y1, color='red', linewidth=1.0, linestyle="--", label='cos(x)')
ax1.legend()

ax2.plot(xx, y2, color='blue', linewidth=1.0,
         linestyle="-", label='sin(x)')
ax2.legend()

plt.show()
```

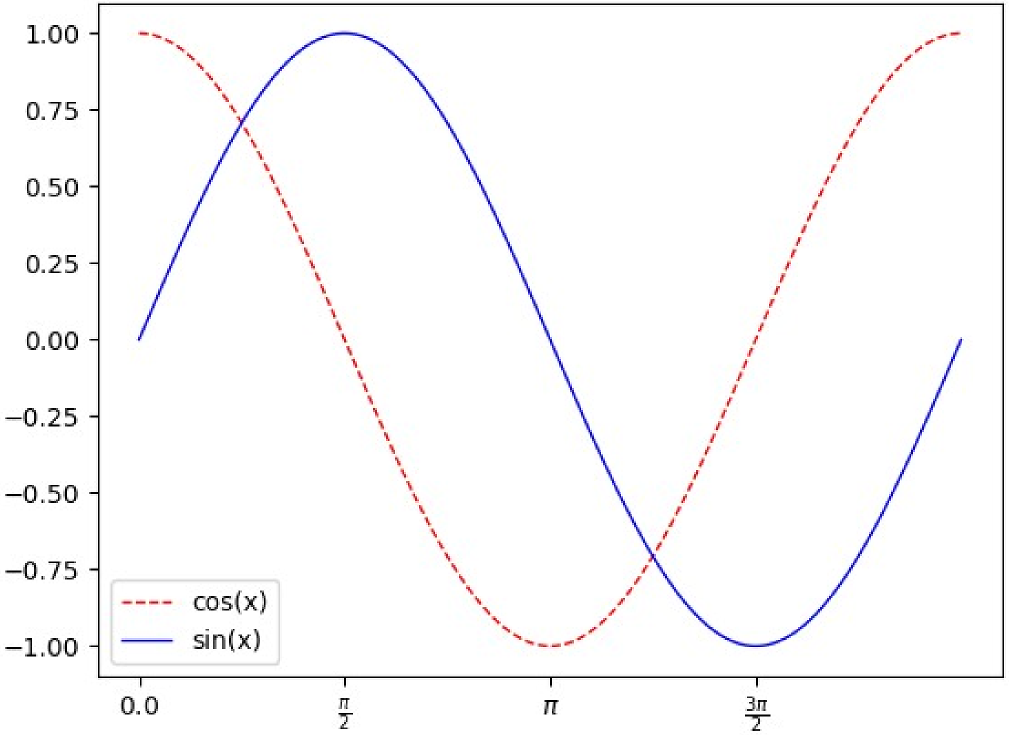
Two rows, one column (2 figures)



Rendering TeX within plots

```
ax.set_xticks(
    [0.0, np.pi / 2, np.pi, 3 * np.pi / 2],
    [r'$0.0$', r'$\frac{\pi}{2}$', r'$\pi$', r'$\frac{3\pi}{2}$']
)
```

- Matplotlib can render TeX sequences in plots
- TeX-sequences should be delimited by $
- It is advisable to put TeX-sequences into **raw-strings** (**r’something’**)
- In raw-strings, backslashes are interpreted literally and not as special Python commands (e.g. \n as “\” “n” and not as newline)
- Useful when passing backslash commands to various enginens (TeX-sequences in Matplotlib, regular expressions, ...)



Further useful Axes methods

ax.set_xlim(), **ax.set_ylim()**	Setting/Querying x/y limits
ax.set_xticks(), **ax.set_yticks()**	Setting customized ticks (and tick labels)
ax.annotate()	Write text into the plot
ax.plot()	Curve plot
ax.scatter()	Scatter plot
ax.bar()	Bar plot
ax.contour()	Contour plot
ax.imshow()	Bitmap image
ax.pie()	Pie charts
ax.quiver()	Quiver plots

:

- Various excellent tutorials on Matplotlib available
- See for example Matplotlib Quick Start Guide or Matplotlib: Plotting (in Scipy Lecture Notes)
- Some tutorials (e.g. Scipy-lectures) use the global interface access (easy to convert)

Have fun!

Next time we will need:

- A proper Python source code editor (e.g. Visual Studio Code)
- Git (can be installed via Conda)