Data processing and visualization with python

Data analysis or data processing is one of the core requirements in any scientific field. Python is a general purpose language that has an extensive library of importable modules for nearly any field.

In this workshop we will talk about using the most popular of these python modules to understand how to handle data, process it and plot.

Making sure we have the right environment

Open up a python interpreter and type in each of the following lines

```
Please check out: http://continuum.io/thanks and
https://anaconda.org
>>> import matplotlib
>>> import numpy
>>> import scipy
>>> import pandas
>>>
```

No error should be raised if all these packages are correctly installed. The anaconda package installer should have installed all of these correctly.

SciPy - Scientific Computing Tools for Python

Check more here https://www.scipy.org/about.html

> SciPy refers to several related but distinct entities:

> - The SciPy Stack, a collection of open source software for

scientific computing in Python, and particularly a specified set of core packages.
- The community of people who use and develop this stack.
- Several conferences dedicated to scientific computing in Python - SciPy, EuroSciPy and SciPy.in.
- The SciPy library, one component of the SciPy stack, providing many numerical routines.

Taken from https://www.safaribooksonline.com/library/view/python-for-data/9781449323592/ch04.html

SciPy is a collection of packages addressing a number of different standard problem domains in scientific computing. Here is a sampling of the packages included:

- scipy.integrate: numerical integration routines and differential equation solvers
- scipy.linalg: linear algebra routines and matrix decompositions extending beyond those provided in numpy.linalg.
- scipy.optimize: function optimizers (minimizers) and root finding algorithms
- scipy.signal: signal processing tools
- scipy.sparse: sparse matrices and sparse linear system solvers
- scipy.special: wrapper around SPECFUN, a Fortran library implementing many common mathematical functions, such as the gamma function
- scipy.stats: standard continuous and discrete probability distributions (density functions, samplers, continuous distribution functions), various statistical tests, and more descriptive statistics
- scipy.weave: tool for using inline C++ code to accelerate array computations

NumPy http://www.numpy.org/

> NumPy is the fundamental package for scientific computing with Python. It contains among other things:
>
> - a powerful N-dimensional array object
> - sophisticated (broadcasting) functions
> - tools for integrating C/C++ and Fortran code
> - useful linear algebra, Fourier transform, and random number capabilities

> Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases. NumPy is licensed under the BSD license, enabling reuse with few restrictions.

> Taken from https://www.safaribooksonline.com/library/view/python-for-data/9781449323592/ch04.html
>
> NumPy, short for Numerical Python, is the foundational package for scientific computing in Python. The majority of this book will be based on NumPy and libraries built on top of NumPy. It provides, among other things:
>
> - A fast and efficient multidimensional array object ndarray
> - Functions for performing element-wise computations with arrays or mathematical operations between arrays
> - Tools for reading and writing array-based data sets to disk
> - Linear algebra operations, Fourier transform, and random number generation
> - Tools for integrating connecting C, C++, and Fortran code to Python

> Beyond the fast array-processing capabilities that NumPy adds to Python, one of its primary purposes with regards to data analysis is as the primary container for data to be passed between algorithms. For numerical data, NumPy arrays are a much more efficient way of storing and manipulating data than the other built-in Python data structures. Also, libraries written in a lower-level language, such as C or Fortran, can operate on the data stored in a NumPy array without copying any data.

pandas http://pandas.pydata.org/

> pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.
>
> pandas is a NUMFocus sponsored project. This will help ensure the success of development of pandas as a world-class open-source project, and makes it possible to donate to the project.

> Taken from https://www.safaribooksonline.com/library/view/python-for-data/9781449323592/ch04.html
>
> pandas provides rich data structures and functions designed to make working with structured data fast, easy, and expressive. It is, as you will see, one of the critical ingredients enabling Python to be a powerful and productive data analysis environment. The primary object in pandas that will be used in this book is the DataFrame, a two-dimensional tabular, column-oriented data structure with both row and column labels:

```
    total_bill  tip   sex     smoker  day  time
size
1   16.99       1.01  Female  No      Sun  Dinner  2
2   10.34       1.66  Male    No      Sun  Dinner  3
```

```
3   21.01       3.5   Male    No      Sun  Dinner  3
4   23.68       3.31  Male    No      Sun  Dinner  2
5   24.59       3.61  Female  No      Sun  Dinner  4
6   25.29       4.71  Male    No      Sun  Dinner  4
7   8.77        2     Male    No      Sun  Dinner  2
8   26.88       3.12  Male    No      Sun  Dinner  4
9   15.04       1.96  Male    No      Sun  Dinner  2
10  14.78       3.23  Male    No      Sun  Dinner  2
```

> pandas combines the high performance array-computing features of NumPy with the flexible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data. pandas is the primary tool that we will use in this book.

matplotlib https://matplotlib.org/

> Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, the jupyter notebook, web application servers, and four graphical user interface toolkits.
>
> Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code.

Using modules in Python

In python we are able to use modules that allow us to do things the basic python module doesn't. For this we must first import them into our

environment.

The module SciPy, for example, is highly useful for scientific computation. Many signal processing tools are already present in the package. The documentation can be acessible online or through the __doc__ docstrings inside the interpreter.

```
>>> scipy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'scipy' is not defined
>>> import scipy
>>> print scipy.__doc__
>>> print scipy.signal.__doc__
```

A useful subpackage of SciPy is the scipy.constants package containing many useful physical constants.

```
>>> import scipy.constants as const
>>> from scipy import constants as const
>>> print const.__doc__
>>> const.m_e
9.10938356e-31
>>> print const.unit('electron mass')
kg
>>>
```

> ***Question:*** What other constants can you think of? SciPy most likely has them.

```
>>> import numpy as np
>>> a = np.array([0,1,2,3,4,5])
>>> b = a**2
```

```
>>> b
array([ 0,  1,  4,  9, 16, 25])
```

Introduction to NumPy

Python lists are great to store a wide variety of data as they are very flexible. However, such flexibility comes at a cost of computational efficiency.

Imagine you have a range of x values for which you want to calculate y = f(x).

Let's consider

```
>>> x = [0,1,2,3,4]
>>> y = x**2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or
pow(): 'list' and 'int'
>>>
```

We get an error because we cannot directly calculate on each value of the list. We have to iterate over each value. For example:

```
>>> y = [x**2 for x in x]
>>> y
[0, 1, 4, 9, 16]
>>>
```

> ***Question:*** What happens when you want to multiply a value

> elementwise for all the elements in an array?

```
>>> a = [0,1,2,3,4,5]
>>> b = a**2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or
pow(): 'list' and 'int'
```

> ***Question:*** What about when using the standard array module in Python?

```
>>> import array
>>> a = array.array('f',[0,1,2,3,4,5])
>>> a
array('f', [0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
>>> a*2
array('f', [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 0.0, 1.0,
2.0, 3.0, 4.0, 5.0])
>>>
```

Still doable, but more complicated. This complicates much more when you want more complicated data processing with different variables.

Advantages of NumPy

For this reason the NumPy Numerical Python package was created. NumPy enables easy computing of multidimensional arrays of data and also includes the most used processing functions such as Fourier transforms, linear algebra, etc.

From *"A primer on scientific programming with Python"*

> - All elements must be of the same type, preferably integer,

real, or complex numbers, for efficient numerical computing and storage.
- The number of elements must be known when the array is created.
- Arrays are not part of standard Python – one needs an additional package called Numerical Python, often abbreviated as NumPy. The Python name of the package, to be used in import statements, is numpy.
- With numpy, a wide range of mathematical operations can be done directly on complete arrays, thereby removing the need for loops over array elements. This is commonly called vectorization
- Arrays with one index are often called vectors. Arrays with two indices are used as an efficient data structure for tables, instead of lists of lists. Arrays can also have three or more indices.

The number of elements in an array can be changed but at a high computational cost.

For this kind of procedure you should use NumPy (https://docs.scipy.org/doc/numpy/reference/index.html)

A list can be converted to a numpy array as long as every element can be casted to the same data type.

```
>>> import numpy as np
>>> a = np.array([0,1,2,3,4,5])
>>> b = a**2
>>> b
array([ 0,  1,  4,  9, 16, 25])
```

Vectorization

Instead of looping over very value in the array, numpy can use vectorization, which means that the operation is executed over the whole array directly, without expliciting the loop.

Vectorization is a great procedure as it allows complicating the mathematical operation without code complexity.

```
>>> x = np.linspace(0,2,201)
>>> y = np.sin(np.pi*x)*np.cos(x)*np.exp(-x**2) + 2 +
x**2
>>>
```

Functions can also receive numpy arrays as arguments and execute operations on them

```
>>> def f(x):
...     return np.exp(x**2)
...
>>> f(x)
```

Creating arrays

```
>>> a = np.arange(0,10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a = np.arange(0,10,2)
>>> a
array([0, 2, 4, 6, 8])
>>> a = np.linspace(0,10)
>>> a
array([  0.        ,   0.20408163,   0.40816327,
0.6122449 ,
         0.81632653,   1.02040816,   1.2244898 ,
```

```
1.42857143,
        1.63265306,   1.83673469,   2.04081633,
2.24489796,
        2.44897959,   2.65306122,   2.85714286,
3.06122449,
        3.26530612,   3.46938776,   3.67346939,
3.87755102,
        4.08163265,   4.28571429,   4.48979592,
4.69387755,
        4.89795918,   5.10204082,   5.30612245,
5.51020408,
        5.71428571,   5.91836735,   6.12244898,
6.32653061,
        6.53061224,   6.73469388,   6.93877551,
7.14285714,
        7.34693878,   7.55102041,   7.75510204,
7.95918367,
        8.16326531,   8.36734694,   8.57142857,
8.7755102 ,
        8.97959184,   9.18367347,   9.3877551 ,
9.59183673,
        9.79591837,  10.        ])
>>> a = np.linspace(0,10,5)
>>> a
array([  0. ,   2.5,   5. ,   7.5,  10. ])
>>> a = np.arange(10).reshape(2,5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> np.arange(0,2,0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,
0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,
1.8,  1.9])
>>>
```

Creating regular arrays using NumPy:

- arange([start,] stop[, step,][, dtype]) ... Return evenly spaced values within a given interval.
- linspace(start, stop[, num, endpoint, ...]) ... Return evenly spaced numbers over a specified interval.
- logspace(start, stop[, num, endpoint, base, ...]) ... Return numbers spaced evenly on a log scale.
- geomspace(start, stop[, num, endpoint, dtype]) ... Return numbers spaced evenly on a log scale (a geometric progression).

Creating arrays with zeros and ones:

- empty(shape[, dtype, order]) ... Returna new array of given shape and type, without initializing entries.
- eye(N[, M, k, dtype]) ... Return a 2-D array with ones on the diagonal and zeros elsewhere.
- identity(n[, dtype]) ... Return the identity array.
- ones(shape[, dtype, order]) ... Return a new array of given shape and type, filled with ones.
- zeros(shape[, dtype, order]) ... Return a new array of given shape and type, filled with zeros.
- full(shape, fill_value[, dtype, order]) ... Return a new array of given shape and type, filled with fill_value.

```
>>> np.dot(a,a)
55
>>> c = np.array([[1,1,1,1,1,1],[2,2,2,2,2,2]])
>>> np.dot(a,c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shapes (6,) and (2,6) not aligned: 6 (dim
0) != 2 (dim 0)
>>> np.inner(a,c)
array([15, 30])
>>> c.size
12
```

```
>>> c.shape
(2, 6)
>>> len(c)
2
>>>
```

More linear algebra using python can be checked at https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

```
>>> t = np.linspace(0,1,1000)
>>> f = 123
>>> y = np.sin(2*np.pi*f*t)
>>>
```

Handling simple .csv files using NumPy

Data can be stored and loaded in pretty much any kind of structure using Python, from comma separated values, json, binary files, databases or compressed files.

Here we will introduce how to store and load data in the common comma separated value .csv format.

These .csv files can consist of solely the data you want to load but also a header row that determines the value of each column. In this chapter we will only consider clean, data-only .csv files.

For the purpose of this tutorial we gathered the Lisbon hourly temperature between July 11th to July 25th 2017 from meteoblue.com. You can find this data in the data/lisbon_temperature.csv file.

Each column in the file is the temperature in Lisbon at each hour,

starting from 0 to 23, and each row is the respective day 11 to 25 July 2017.

The file looks something like this:

```
16.93,16.54,16.29,16.12,15.99,15.83,15.73, ...
18.18,18.16,18.04,17.92,17.75,17.66,17.63, ...
18.94,18.44,18.27,18.23,18.33,18.19,17.92, ...
...
```

Now we want to load this data into our python script so that we can use it. Luckily there are many functions to handle .csv files in python, namely our familiar NumPy.

We can import the data by running

```
filename = 'lisbon_temperature.csv'
data = np.genfromtxt(filename, delimiter=',')
```

Numpy tries to automatically intepret the kind of data it loads, defaulting most of the times to floating point. And we can check the data that we just loaded

```
print data
print data.shape
print len(data)
print data.T
print data[0]
print data[:,0]
```

Analysing the data

> ***Question:*** How is the maximum temperature achieved in these two weeks? What about the minimum and average temperatures?

```
print np.mean(data)
print np.max(data)
print np.min(data)
```

> ***Question:*** How can we calculate the average daily temperature? What about the maximum and minimum temperatures in each day?

```
# Hourly temperatures (Column based)
print np.mean(data, axis=0)
print np.max(data, axis=0)
print np.min(data, axis=0)

# Daily temperatures (Row based)
print np.mean(data, axis=1)
print np.max(data, axis=1)
print np.min(data, axis=1)
```

Visualizing data with matplotlib

All the data seems to be there. But it is hard to make sense of just numbers. It is much easier if we can visualize all the data that we just loaded. Python has the matplotlib package for plotting any kinds of data using a range of different backends.

Importing matplotlib is as simple as:

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

x = np.linspace(0,4,401)
y = np.sin(np.pi*x)
plt.plot(x,y)
plt.show()
```

A window should open up after plt.show() with the plotted x,y data. You can use this window to zoom in on the plotted data, make some small adjustments and save the figures.

> ***Note:*** I suggest using these tools to verify some data, but any details in publication figures should be explicit in the respective python script or notebook!

We can also decorate our matplotlib plots in any way we want, using data labels, legends, colors, etc.

Visualizing .csv data

All the data seems to be there. But it is hard to make sense of just numbers. It is much easier if we can visualize all the data that we just loaded. For this we plot the data in our script.

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
    plt.plot(data)
    plt.show()
```

> ***Question:*** What is wrong with our plot?

> ***Question:*** How to plot a single day?

> ***Question:*** How to plot all the days correctly overlapping?

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
    for day_data in data:
        plt.plot(day_data)
    plt.show()
```

We only have temperature data and we know that hourly temperature is distributed along the columns and that days starting at 11 correspond to each row. We can create day and hour auxiliary vectors to compensate for this.

> ***Question:*** How can we make these auxiliary hour and days vectors? We know hours go from 0 to 23 and the days are from 11 to 25. Should we use floats, int?

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
    hour = np.arange(0, 24, 1)
    days = np.arange(11,26)
    print hour
    print days
    for day_temp in data:
        plt.plot(hour, day_temp)
```

```
    plt.show()
```

Decorating with axis labels

Now we can decorate our figure by adding x and y labels and a title.

```
    plt.title('Lisbon temperature')
    plt.xlabel('Time of day')
    plt.ylabel('Temperature [C]')
```

Processing the data to get the average, maximum and minimum temperatures

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
    hour = np.arange(0, 24, 1)
    days = np.arange(11,26)

    temp_mean = np.mean(data, axis=0)
    temp_max = np.max(data, axis=0)
    temp_min = np.min(data, axis=0)

    plt.plot(hour, temp_mean)
    plt.plot(hour, temp_max)
    plt.plot(hour, temp_min)
    plt.title('Lisbon temperature')
    plt.xlabel('Time of day')
    plt.ylabel('Temperature [C]')
    plt.show()
```

Adding legends to each plot

Adding legends to the data that we have plotted is as simple as doing

```
    plt.legend(['Average','Maximum','Minimum'])
```

However, matplotlib has a great feature that you can add labels to the individual plots and never have to worry about the sequence of data being plotted. This is aone of my favorite features coming from matlab.

You can even disregard the label of a single plot if you do not want it to show.

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
    hour = np.arange(0, 24, 1)
    days = np.arange(11,26)

    temp_mean = np.mean(data, axis=0)
    temp_max = np.max(data, axis=0)
    temp_min = np.min(data, axis=0)

    plt.plot(hour, temp_mean, label='Average')
    plt.plot(hour, temp_max, label='Maximum')
    plt.plot(hour, temp_min, label='Minimum')

    plt.legend()

    plt.title('Lisbon temperature')
    plt.xlabel('Time of day')
    plt.ylabel('Temperature [C]')
    plt.show()
```

Color and line style decorations

We can also change the colors of our plots besides the default colormaps. Consider that we want the maximum temperate to be red and the minimum blue while the average is green, for example. We can use english style colors as these are internally matched to the correct color:

```
    plt.plot(hour, temp_mean, label='Average',
color='Green')
    plt.plot(hour, temp_max, label='Maximum',
color='Red')
    plt.plot(hour, temp_min, label='Minimum',
color='Blue')
```

We can make the average line thicker:

```
    plt.plot(hour, temp_mean, label='Average',
color='Green', linewidth=5)
```

And we can change the alpha of the plot:

```
    plt.plot(hour, temp_mean, label='Average',
color='Green', linewidth=5, alpha=0.5)
```

And add markers

```
    plt.plot(hour, temp_mean, label='Average',
color='Green', linewidth=5, alpha=0.5, marker='o')
```

> ***Question:*** What happens if you resize the plot of each one?

2D plotting

We can also plot this data in a 2D plot. For this we also use the internal matplotlib functions pcolormesh or imshow.

My favorite is pcolormesh for scientific data since it allows for different xy aspects while imshow tries to maintain xy aspect ratio of 1, assuming that each data cell is a pixel of an image. Let us compare both on a *new* script file:

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
    plt.figure()
    plt.imshow(data)
    plt.figure()
    plt.pcolormesh(data)
    plt.show()
```

2D data is actually 3 dimensional data while the third dimension is represented by the color of the 2D plot. In order to get a scale of the color we require a colorbar to be plotted alongside our mesh. We remove the unused imshow and add the colorbar

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
```

```
    plt.pcolormesh(data)
    plt.colorbar(label='Temperature [C]')
    plt.ylabel('Day')
    plt.xlabel('Time of day')
    plt.show()
```

Using subplots

Subplots are a great tool when you want to compare different data in the same figure. Here we will use both the average daily and hourly temperatures as well as the 2D plot to represent our data and gain some interesting insight.

First lets get acquainted with the subplot syntax:

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    fig, axarray = plt.subplots(2, 2)
    plt.plot([1, 2, 3, 1, 2, 12, 13])
    plt.show()
```

We see that a 2x2 grid of axes were created. And that our plot was plotted into the last axis on the lower right.

Now we want to reference each of these axes to plot our data to it. The syntax changes slightly for our subplots but once we get used to it, they are very versatile.

The reference to each axis in the 2x2 grid is stored in the (2,2) axarray variable. We select the current axis by indexing inside the axarray. The [0,0] position is the top left, [0,1] is top right, and the [1,1] position is the lower right. If we only use a 2x1 or 1x2 grid, for example, our axarray is a one dimension vector.

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    fig, axarray = plt.subplots(2, 2)
    plt.sca(axarray[0,0])
    plt.plot([1, 2, 3, 1, 2, 12, 13])
    plt.sca(axarray[0,1])
    plt.plot([1, 2, 3, 1, 2, 12, 13])
    plt.show()
```

Now lets start a subplot with the 2D temperature plot on the top left and the hourly average, maximum and minimum data on the bottom left

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
    hour = np.arange(0, 24, 1)
    days = np.arange(11,26)

    temp_mean = np.mean(data, axis=0)
    temp_max = np.max(data, axis=0)
    temp_min = np.min(data, axis=0)

    fig, axarray = plt.subplots(2, 2)

    # Bottom left
    plt.sca(axarray[1,0])
    plt.pcolormesh(hour,days,data)

    # Top left
    plt.sca(axarray[0,0])
    plt.plot(hour, temp_mean, label='Average')
    plt.plot(hour, temp_max, label='Maximum')
```

```
    plt.plot(hour, temp_min, label='Minimum')

    plt.show()
```

Lets add the daily temperature data to the top right:

```
    # Bottom right
    plt.sca(axarray[1,1])
    day_mean = np.mean(data, axis=1)
    day_max = np.max(data, axis=1)
    day_min = np.min(data, axis=1)
    plt.plot(days, day_mean, label='Average')
    plt.plot(days, day_max, label='Maximum')
    plt.plot(days, day_min, label='Minimum')
```

But actually lets swap x and y in this plot so it matches the top left 2D data:

```
    # Bottom right
    plt.sca(axarray[1,1])
    day_mean = np.mean(data, axis=1)
    day_max = np.max(data, axis=1)
    day_min = np.min(data, axis=1)
    plt.plot(day_mean, days, label='Average')
    plt.plot(day_max, days, label='Maximum')
    plt.plot(day_min, days, label='Minimum')
```

But if we zoom in on any of the plots, we only affect that one! We can, however, make so that all the subplots share each other's axis and limits. To do this we alter the suplots method to include how the axes should be shared:

```
    fig, axarray = plt.subplots(2, 2, sharex='col',
sharey='row')
```

We can hide the top right axes since we are not using it with

```
    axarray[0,1].axis('off')
```

Final script with decorations

And we can add the respective x and y labels to each of the plots. The final script is something like:

```
import matplotlib.pyplot as plt
import numpy as np
if __name__ == "__main__":
    filename = 'lisbon_temperature.csv'
    data = np.genfromtxt(filename, delimiter=',')
    hour = np.arange(0, 24, 1)
    days = np.arange(11,26)

    temp_mean = np.mean(data, axis=0)
    temp_max = np.max(data, axis=0)
    temp_min = np.min(data, axis=0)

    fig, axarray = plt.subplots(2, 2, sharex='col',
sharey='row')

    # Bottom left
    plt.sca(axarray[1,0])
    plt.pcolormesh(hour,days,data)
    plt.ylabel('Day')
    plt.xlabel('Time of day')
```

```
    # Top left
    plt.sca(axarray[0,0])
    plt.plot(hour, temp_mean, label='Average')
    plt.plot(hour, temp_max, label='Maximum')
    plt.plot(hour, temp_min, label='Minimum')
    plt.ylabel('Temperature [C]')

    # Bottom right
    plt.sca(axarray[1,1])
    day_mean = np.mean(data, axis=1)
    day_max = np.max(data, axis=1)
    day_min = np.min(data, axis=1)

    plt.plot(day_mean, days, label='Average')
    plt.plot(day_max, days, label='Maximum')
    plt.plot(day_min, days, label='Minimum')
    plt.xlabel('Temperature [C]')

    # Hiding top right frame
    axarray[0,1].axis('off')

    plt.suptitle('Lisbon temperature in July 2017')
    plt.show()
```

Saving figures

matplotlib also allows saving figures to most output file formats. We can save the figure directly from our figure window. However for reproducible images, we should embed the figure saving into our script. This is as simple as calling savefig at the end of the script. The output image file format is automatically determined from the filename extension. Output file formats can be png, pdf, eps, svg, etc. We can also set the output dpi.

```
    fig.savefig('lisbon_temperature.png', dpi=300)
```

Summary