Python for atmospheric measurements Documentation

Release 0.1

I.B.

Nov 15, 2017

Contents

Contents:

CHAPTER 1

Introduction

CHAPTER 2

Installing python

In order to use python for scientific computing you will need to install:

1. The python language
2. A set of modules that allow you to manipulate data, plot, etc.

2.1 Basic Python

If you are using Linux or Mac OS you most probably have python already installed. If you are using Windows you can download python from http://www.python.org/getit/. From there download and run “Python 2.7.3 Windows Installer”.

Python has now two active versions, 2.7.3 and 3.2.3. While the 3.* versions are the future of python, right now most modules support the 2.* version, so we will use that.

2.2 Scientific modules

There are various useful modules for scientific programming with python. Among other:

- Numpy - Allows fast N-dimensional array manipulation (http://numpy.scipy.org/)
- Scipy - A collection open-source software for mathematics, science, and engineering (http://www.scipy.org/)
- Matplotlib - A library for plotting data (http://matplotlib.sourceforge.net/)

While installing all these is relatively easy, it’s even easier to install a collection of these and many other useful modules for scientific computing.

A very good collection is Anaconda. Anaconda can be installed in Windows, Linux, and MacOS X, and includes:

- The above mentioned modules
- Spyder - “a powerful interactive development environment for the Python language with advanced editing, interactive testing, debugging and introspection features”

- ipython - “a powerful Python shells” suitable for interactive scientific computing.
- and hundreds of more modules that you (might) need.

After installing Anaconda you are ready to start using python!

CHAPTER 3

Python quick start - Part 1

The best way to understand python is to start using it. We will use here ipython, which is an advanced terminal that has many interesting features for scientific computing.

To start ipython open a terminal (in windows go to Start -> Run and type cmd) and type:

```
ipython
```

3.1 Python as calculator

You can first start using python as a calculator. Try the following:

```
# Simple addition
In [1]: 1+1
Out[1]: 2

# Powers
In [2]: 2**10
Out[2]: 1024

# Integer division. This will return only the quotient of the division
In [3]: 10/4
Out[3]: 2

# If you divide an integer by a float (decimal) number (in this case 4.0)
# you will get the full division.
In [4]: 10/4.0
Out[4]: 2.5
```

Numbers, and the results of your calculations can be stored in variables:

```
In [5]: a = 5
In [6]: a**3
Out[6]: 125
```

Python handles also complex numbers:

```
In [7]: a = 1 + 4j

In [8]: b = 3 - 2j

In [9]: c = a*b

In [10]: c
Out[10]: (11+10j)
```

You can get the real and imaginary part of your results easily:

```
In [11]: c.real
Out[11]: 11

In [12]: c.imag
Out[12]: 10
```

Note: Using the dot after anything in python, like you did in c.real for example, lets you explore a subpart of that thing, in a way, lets you look “inside” the thing you are using.

3.2 Python and arrays

Python’s mathematical capabilities are great, but there are a lot missing. To really use python for computations your will need to import numpy:

```
>>> import numpy
```

Note: From now on, we will avoid writing the “In [ ]:” part before input for brevity. We will put >>> in front of your input.

Now you can use many useful functions that are included in numpy by typing numpy.<function_name>:

```
>>> numpy.exp(10)
22026.465794806718

>>> numpy.log10(100)
2.0
```

The core of numpy are numerical arrays that allow you to perform fast calculations:

```
# You can create an array like this
>>> a = numpy.array([1,2,3])
>>> a
array([1, 2, 3])
```

Now a is array that can be manipulated:

```
>>> a**3
array([ 1,  8, 27])
```

You can create a list of numbers faster using the arange function:

```
>>> b = numpy.arange(10.0)
```

You can do even more complex calculations like $b^3 - \frac{10}{b+5} + 5$

```
>>> c = b**3 - 10/(b + 5) + 5
```

3.3 Plotting

We can plot our arrays by using the matplotilb module. It is a very feature-full module that, lucky for us, has a set of convenient commands that make plotting easy. We will use only this subset of commands to start using matplotlib type:

```
from matplotlib import pyplot as plt
```

Don't worry if you don't understand this. In sort, inside matplotlib exists a submodule that is called pyplot. We import this and give it the name "plt" for convenience.

Now try the following:

```
# Create the plot based on c
>>> plt.plot(c)

# Show it on screen
>>> plt.show()
```

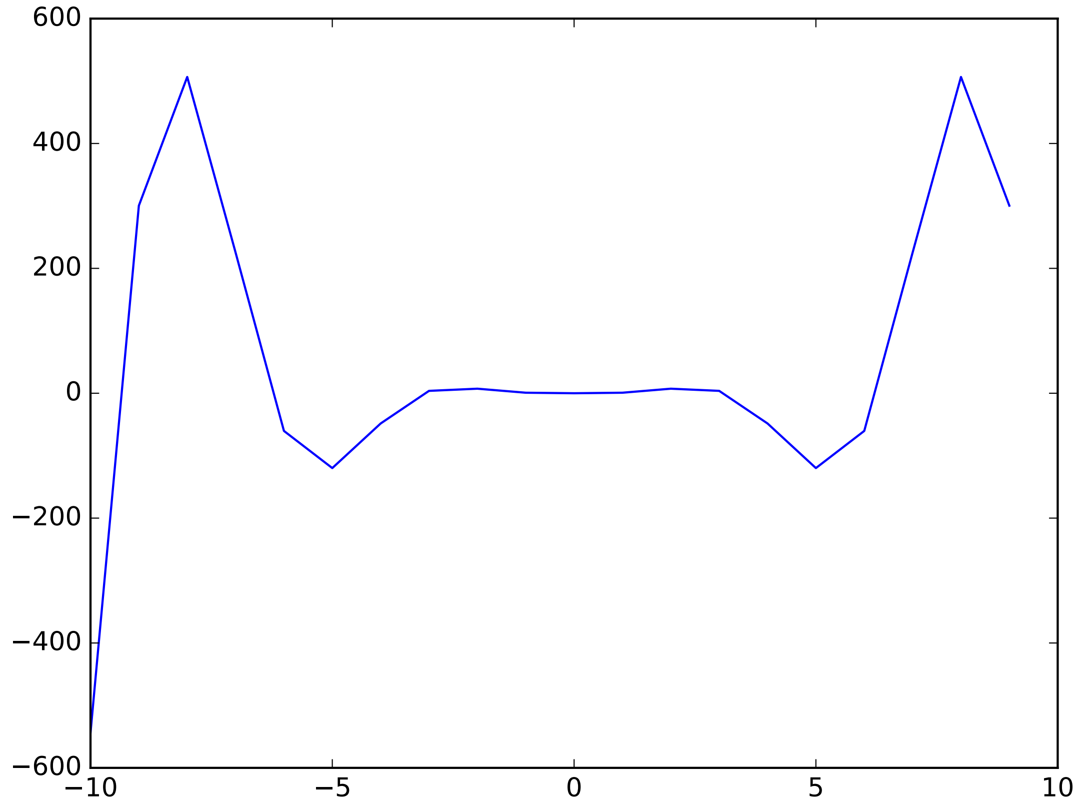
This should put the values of the c array in the y axis. Let's try something more interesting:

```
# Get an array of numbers from -10 to 10
>>> x = numpy.arange(-10.0, 10.0)

# Calculate a more complicated y
>>> y = numpy.sin(x) * x**3

# Now plot y versus x
>>> plt.plot(x,y)

# Show the results
>>> plt.show()
```



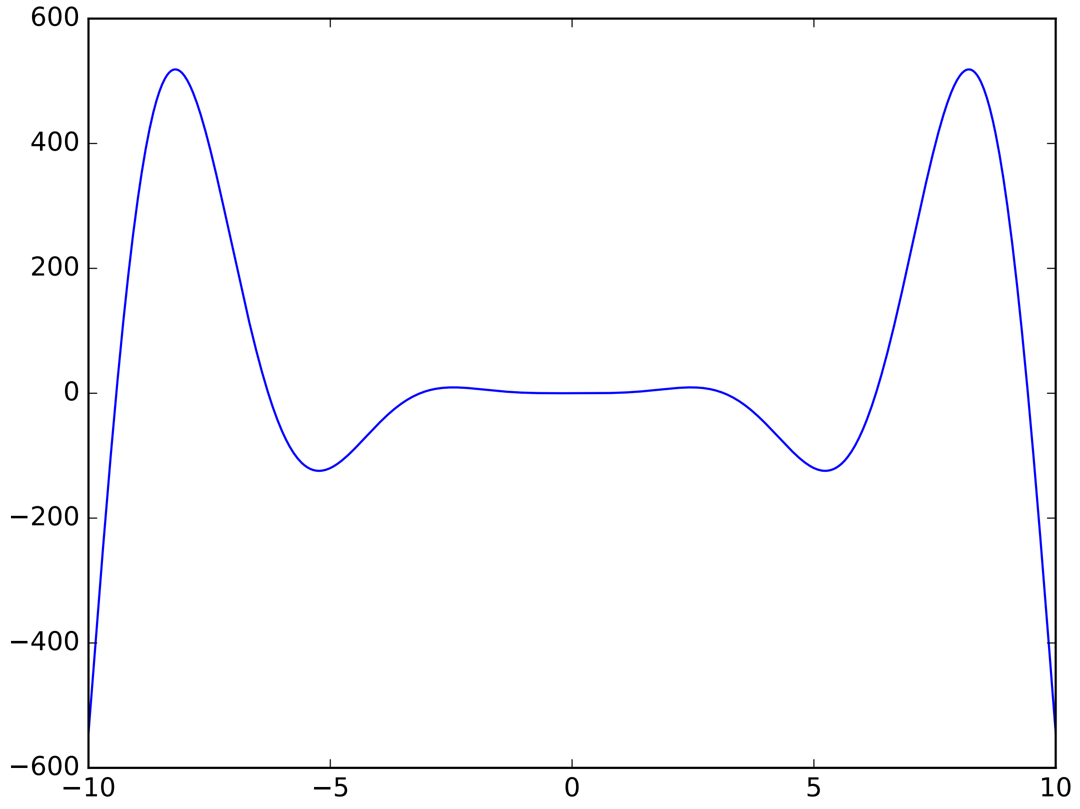
This should give you a plot, that looks very rough. This is because we are plotting only on integer numbers. We can improve this by using the numpy command "linspace" that can give you linearly spaced numbers. Close the previous plot and try:

```
# Get an array of 1000 numbers from -10 to 10
>>> x = numpy.linspace(-10.0, 10.0, 1000)

# Calculate a more complicated y
>>> y = numpy.sin(x) * x**3

# Now plot y versus x
>>> plt.plot(x,y)

# Show the results
>>> plt.show()
```



This looks much better!

CHAPTER 4

Python quick start - Part 2

What we have seen in the first part of this quick start can be made easier. *Ipython* has the option to import when it starts many useful packages for numerical and scientific computing so they are ready for you to use. To use this options you need to start ipython with the command:

```
ipython --pylab
```

Using this option numpy as matplotlib are a already imported and ready to use.

4.1 Reading files

A common task when working with measurements is to read a file in order to process and visuallise some data. Lets suppose that you need to open a text file ("measurements.txt") that contains columns of numerical data:

```
1    23.4
2    43.8
3    55.1
```

You can read this file using the **loadtxt** command from numpy:

```
>>> m = loadtxt("measurements.txt")
```

Note: Remember that the *loadtxt* command is available to you, without importing numpy explicitly, because we are using the –pylab option.

The variable *m* is now an array that contains the numerical data. You can see its contents by simply typing:

```
>>> print m
# or
>>> m
```

You can also check its shape by typing:

```
>>> m.shape
(3,2)
```

This means that array has three rows and two columns.

An common case is that your text file contains some column titles, for example:

```
No.  Value
----------
1    23.4
2    43.8
3    55.1
```

If you try to read such a file as before you will get an error as the **loadtxt** command expects to find only columns of numbers. You can read this file by writing:

```
>>> m = loadtxt("measurements.txt", skiprows = 2)
```

This instructs python to skip the first two rows and start reading columnar data after that.

4.2 Array slicing

A very important feature of an array is that you can select the data that you need to access. The following commands will return arrays that hold only the selected subsets of data.

Warning: You need to take care as the numbering of rows and columns starts from 0 and not from 1.

For example you can get these single-element arrays:

```
>>> a[0,0] # the first row and the first column
1
>>> a[2,1] # third row, second column
55.1
```

You can select more than one numbers at once using the following notation:

```
>>> a[0:2, 0]
array([ 1.,  2.])
```

This returns the rows 0 and 1 from column 0.

You can select a complete axis by simply using ":":

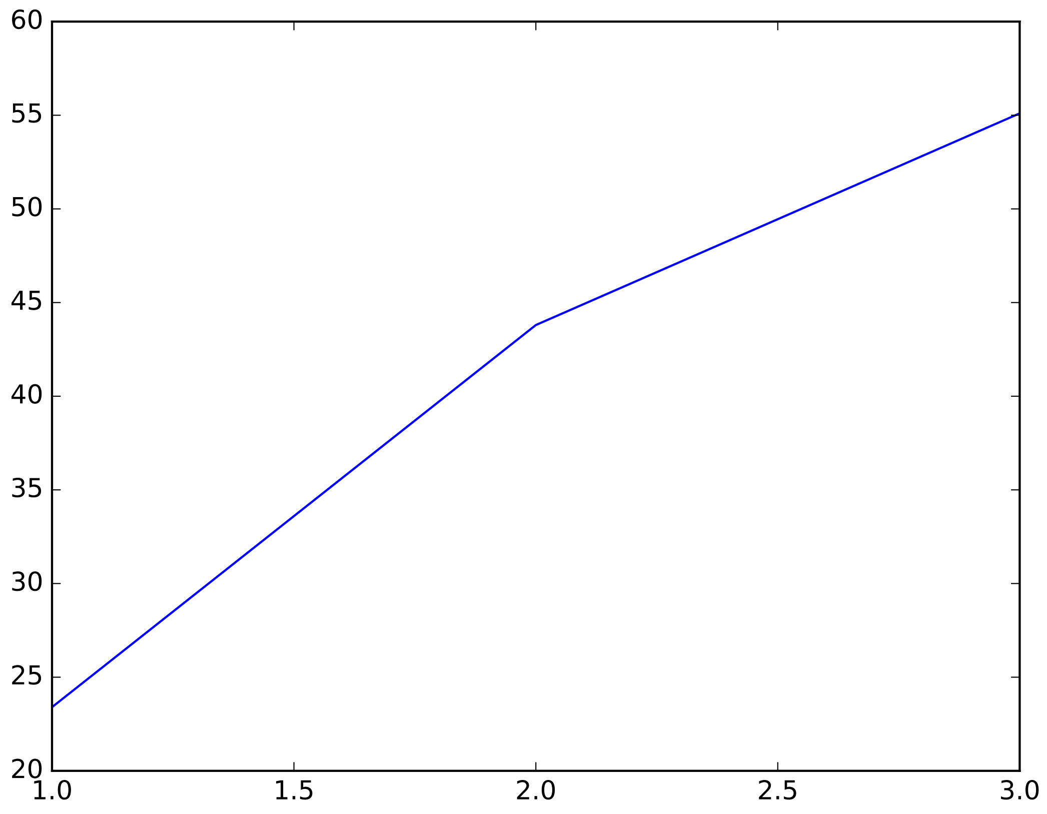
```
>>> a[:,0] # This returns all elements of column 0
array([ 1.,  2.,  3.])
>>> a[1,:] # This returns all elements of row 1
array([  2. ,   43.8])
```

With these simple notation you can easily access the subset of your data that you need to use.

4.3 More on Plotting

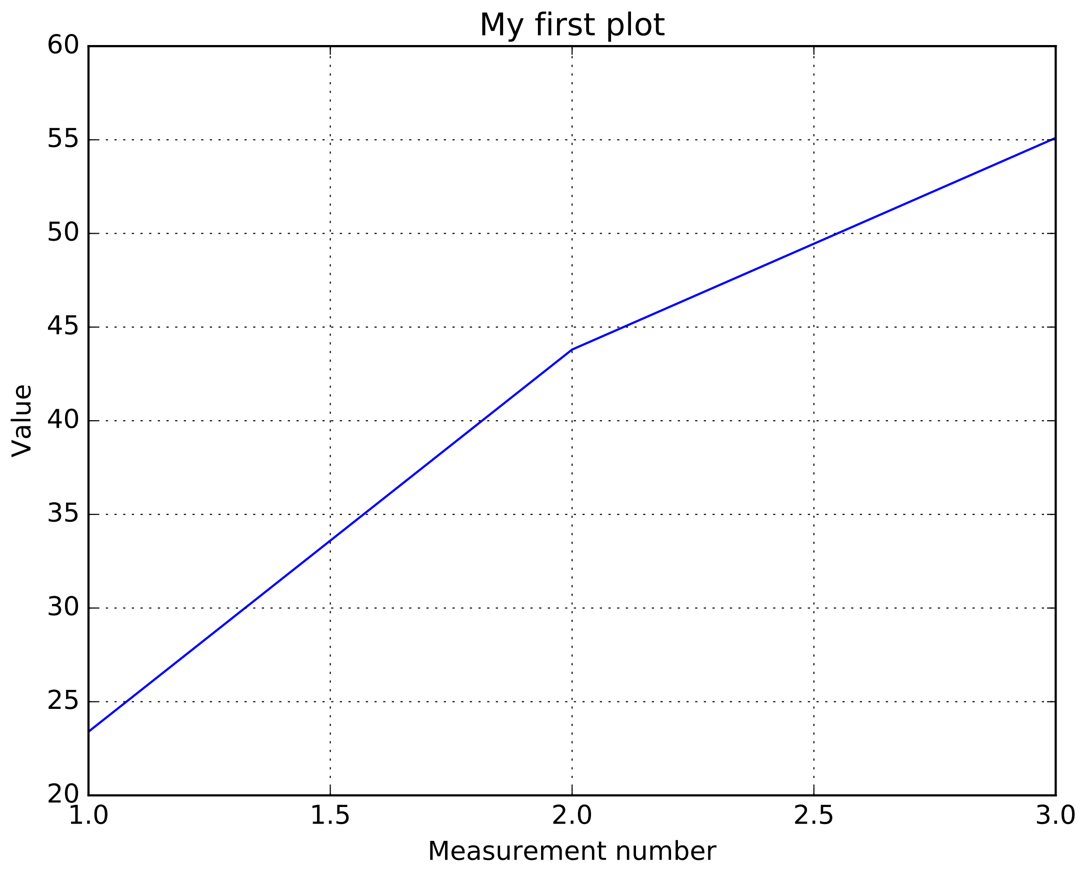
Now that we now how to access our data we can try to plot them:

```
>>> x = a[:,0]
>>> y = a[:,1]
>>> plot(x,y)
```



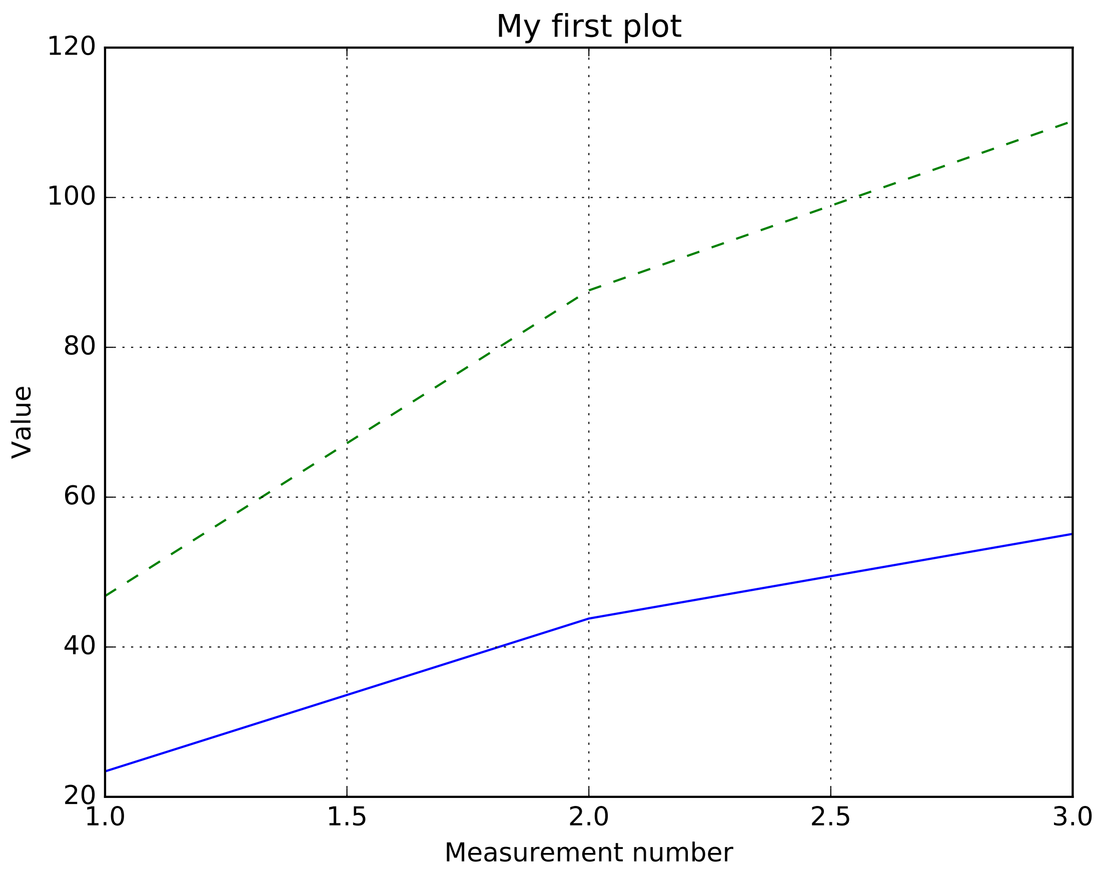
This needs some styling:

```
>>> xlabel("Measurement number") # add a label for the x axis
>>> ylabel("Value")              # add a label for the y axis
>>> title("My first plot")       # add also a title
>>> grid(True)                   # activate the grid
```



You can add another line in the same figure by calling the *plot* function again. Let's plot a second line representing the double of our values, i.e. 2*y. We will also change the style of this line in green dashes:

```
>>> plot(x,2*y, '--g')
```



4.4 Saving your results

Let's say now that you after you have read your measurements you have performed some (very useful) calculation and you want to save the result, so you can use it in the future:

```
>>> result = 1/*y**2 - sin(y)
```

You can save an array in a text file by using the savetxt command:

```
>>> savetxt('output.txt', result)
```

This command will save the array *result* in a file named *output.txt*.

4.5 Getting help

It is certain that soon (or maybe now) you will forget the option you need to use with *loadtxt*, or you will forget the order you need to provide the arguments for *savetxt* (was it the filename first or the array...).

A good way to remember is to use the help within ipython:

```
>>> loadtxt ?
Load data from a text file.

Each row in the text file must have the same number of values.

Parameters
----------
fname : file or str
    File, filename, or generator to read.  If the filename extension is
    ``.gz`` or ``.bz2``, the file is first decompressed. Note that
    generators should return byte strings for Python 3k.
dtype : data-type, optional
    Data-type of the resulting array; default: float.  If this is a
    record data-type, the resulting array will be 1-dimensional, and
    each row will be interpreted as an element of the array.  In this
```

In this way you can learn about a function options and details.

Note: You may need to press **q** in order to exit back to the normal ipython terminal.

An other useful way to learn about the python functions is (ofcourse) the Internet. Here are some links that can help you read more about the functions that you learned in this part of the quick start guide:

- Plotting: http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot
- Reading text files: http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html
- Saving text files: http://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html

CHAPTER 5

Calculating with arrays

Built-in python data types (lists, dictionaries, etc.) are fine for many applications. For mathematical operations, however, these types are not so flexible and fast. This is why the *numpy* module was created, which is now the base for most python scientific code.

The core of numpy is written in the low-level C programming language, so all computations are executed very fast. Moreover, computations with numpy arrays look very similar to the usual mathematical notations and this makes them very easy to read.

5.1 Creating a simple numpy array

To use *numpy* you first need to import it:

```
>>> import numpy as np
```

This means that in our code we will call the numpy module with the short name *np*.

We can now create a first array. One way to create the array is to define all its elements:

```
>>> a = np.array([1, 2, 3])
```

Several things happen in this one line:

- We define a elements of our array in a list i.e. [1, 2, 3];
- We convert this list to a numpy array using the *array* function of the numpy module i.e. np.array()
- We store the resulting array in a variable called *a*.

The variable *a* is now a numpy array, suitable for mathematical computations:

```
>>> print(a)
[1 2 3]

>>> print(2 * a)  # Multiply by 2
```

```
[2 4 6]

>>> print(a**2)    # Square, element by element
[1 4 9]
```

We can, of course, store our results in a new array for further use:

```
>>> b = a**2
>>> print(b)
[1 4 9]

>>> print(a + b)
[2, 6, 12]
```

5.2 Convenient ways to create numpy arrays

The numpy module provides some functions that can create standard numpy arrays easily.

np.ones(<number of elements>) Creates an array of ones:

```
>>> a = np.ones(5)
>>> print(a)
[1.  1.  1.  1.  1.]
```

Note the dots after the numpy 1. This stands for *1.0* i.e. the elements are of type *float* not *integers*.

np.zeros(<number of elements>) Similarly, creates an array of zeros:

```
>>> a = np.zeros(5)
>>> print(a)
[0.  0.  0.  0.  0.]
```

np.arange(<stop number>) Creates an array with a sequence of numbers:

```
>>> a = np.arange(5)
>>> print(a)
[0 1 2 3 4]
```

You can also call the *arange* function with two arguments, defining both the start and stop number e.g.:

```
>>> a = np.arange(5, 10)
>>> print(a)
[5 6 7 8 9]
```

As with any python function, you can read the documentation of *arange* in *ipython* using the question mark:

```
>>> np.arange?
Type:        builtin_function_or_method
String form: <built-in function arange>
Docstring:
arange([start,] stop[, step,], dtype=None)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval ``[start, stop)``
(in other words, the interval including `start` but excluding `stop`).
```

```
For integer arguments the function is equivalent to the Python built-in
`range <http://docs.python.org/lib/built-in-funcs.html>`_ function,
but returns an ndarray rather than a list.

[...]
```

Remember you need to type *q* to exit the documentation view.

5.3 Accessing specific elements of an array

You can access specific elements of an array by using brackets notation. Note, that the first element of an array is called the element *0*. For example:

```
>>> a = np.array([4, 5, 6, 7])
>>> print(a)
[4 5 6 7]

>>> print(a[0])
4

>>> print(a[2])
6
```

You can get a range of elements using the notation [<start index>:<stop index>]. The result is a new numpy array with the specified elements:

```
>>> print(a[0:3])
[4 5 6]

>>> print(a[1:3])
[5 6]
```

Skipping the start means to start from the first element:

```
>>> print(a[:2])   # Equivalent to a[0:2]
[4 5]
```

Similarly, skipping the end index means to get all the elements until the last:

```
>>> print(a[2:])
[6 7]
```

5.4 Process arrays using numpy functions

The numpy module provides many convenient arrays to perform usual mathematical calculations.

You can find the average value of an array using the *mean* function:

```
>>> a = np.array([1, 2, 3, 4])
>>> a_mean = np.mean(a)
>>> print(a_mean)
2.5
```

You can also calculate the minimum, maximum, and median value easily:

```
>>> a_min = np.min(a)
>>> print(a_min)
1

>>> a_max = np.max(a)
>>> print(a_max)
4

>>> a_median = np.max(a)
>>> print(a_median)
2.5
```

See also:

A more advanced introduction in numpy can be found in the tentative numpy tutorial.

Check the numpy list of array creating routines included in numpy.

Check the numpy list of statistical functions included in numpy.

If you are familiar with Matlab programming this comparison can help you understand the similarities and differences of Matlab matrices and numpy arrays.

CHAPTER 6

Writing modules

Up to now we have uses python in an interactive way, using *ipython*. It is useful, however, to put your code in a module so you can use it again.

A module in python is nothing more than a collection of simple text files that contain python code. In the simplest case a module is a single file.

6.1 Creating your first python file

We will create now your first module. First, create a folder that will contain all your future modules, e.g. *mymodules*. Then, open a text editor (e.g. notepad, notepad++, gedit, ...) and write inside a python commands:

```
print "This is my first file"
```

Now, save the file with a name you choose, giving the extension *.py*, e.g. *first_file.py*. Save the file in the folder you just created.

You can now open a command line (terminal) and run the file, using the command:

```
python first_file.py
```

All the command in the file will now be executed and you should see the message "This is my first file" appear in the command line.

6.2 Importing the module

You can now use the same from *ipython* or even from within other python modules. To import the module, start ipython and navigate to the folder where you stored the file for example:

```
cd /home/user/mymodules/
```

using the path to your own folder.

You can now import the module using its name, without the *.py* extension:

```
>>> import first_file
```

When you import a file, all the commands that are in the file are executed. In our case, you should see the text "This is my first file" appear. Admittedly, this is not yet very useful, as your file does not contain any code that you can reuse. Your modules will become more useful when you start using functions, as we will see later.

6.3 Import search path

In the above case we had to navigate to the folder were the file was located to use it. This is not very convenient, especially is you have modules located in different folders.

Every time you try to import a module Python will search at a number of locations to find a module with the required name:

- the current directory
- a list of directories that are calls the python paths.

You can see which directories are in your python path by running the following commands in ipython:

```
>>> import sys
>>> print sys.path
```

The built-in *sys* module gives you access to a number of system specific variables, like the current version of python and the system path. The second command should give you a list of directories that are currently in the search path of python.

You can add modify the above list and add the folder with your own modules. In this way you don't need to navigate every time in the folder before importing a module:

```
>>> sys.path.append['/home/user/mymodules/']
```

changing again the above string to the location of your folder.

See also:

Detailed description of python modules in the official documentation.

The procedure for adding a python path permanently depends on your operating system (e.g. windows or linux and Mac). You can find more information online.

CHAPTER 7

Controling execution

CHAPTER 8

Computing

CHAPTER 9

Writing functions

CHAPTER 10

Reading and writing files

CHAPTER 11

Data plotting

CHAPTER 12

Creating a command line tool

Most of the examples we have seen up to know we using python using the interactive *ipython* environment. Many times, however, it is useful to run your script form the command line. In this chapter we will see how you can write python code that can be used both from the command line and the interactive interpreters.

12.1 The difference with the command line

Let examine for a minute the following file *example.py*:

```python
# filename: example.py
def simple_function():
    print("This is inside the function.")

print("This is outside the function.")
```

Imagine you run the following script from the command line:

```
python example.py
```

The output will be:

```
This is outside the function.
```

When a python files are imported from the command line all the commands in the file are interpreted. In our case first the function is declared, and then the print command is executed.

Specifically, imagine that you now want to interpret the *simple_function* from the command line:

```
>>> import example
This is outside the function.

>>> example.simple_function()
This is inside the function.
```

What just happened? When we imported a file, all the code inside the file was executed, including the print function.

This is not perfect. Ideally, we should import and use a function from a file without executing any other command.

12.2 Keeping some command only for the command line

The way to keep some command only for the command line is to use the following code:

```
if __name__ == "__main__":
    <your commands>
```

Each python module has the built-in property called *__name__*. When the module is the main program running (i.e. it is run from the command line) the the variable __name__ gets the string value "__main__".

Instead, when the module is imported the variable __name__ holds the name of the imported module.

You can check this with this very simple file:

```
# filename: test_name.py
print "The __name__ variable is: " + __name__
```

Then try to run it both from the command line and importing it:

```
python test_name.py
The __name__ variable is: __main__
```

and:

```
>>> import test_name
The __name__ variable is test_name
```

We can now change the *example.py* file to print something only when run from the command line:

```
# filename: example.py
def simple_function():
    print("This is inside the function.")

if __name__ == "__main__":
    print("This is outside the function.")
```

When we import this file nothing is printed. The variable __name__ is not equal to __main__ so the print command is never run.

12.3 Reading user input

A way of making your script more flexible is to ask the user for some input parameters, to control the program execution. This is done by the *raw_input* built-in function:

```
if __name__ == "__main__":
    color = raw_input("What color is your car?")
    print("Your car is %s" % color)
```

The *raw_input* command will just return a string, so if you are expecting a number you should do the conversion yourself.

12.4 Providing arguments for the command line

A command line tool can be extra useful when provide arguments from the command line to control their behavior. Command line arguments can also be used to write batch scripts or when you use your script in a chain of commands, a thing not possible when interactive user input is required.

We will see here how we can read the command line arguments in our script and change the behavior of our program accordingly.

We will try to write a small program that will count the number of '.txt' files in a directory. We start by writing a function that will do just this and save it in a file. A simple implementation could be:

```
# filename: count_txt.py
import os
import glob  # Module to read the content of the directory

def count_txt(directory):
    """ Count the number of txt files in the provided directory. """

    # Find the search path in a system-independent way
    # this will give e.g. /my/dir/*.txt
    search_str = os.path.join(directory, '*.txt')

    # Get a list with all the files
    files = glob.glob(search_str)

    # Count the files in the list
    number_of_files = len(files)

    return number_of_files
```

The above function will get a list of all the files with the .txt ending in a directory and will return their number. We want now to provide the directory name in the command line e.g.:

```
python count_txt /my/dir/
```

One way to do it is to use the *sys* module. This module contains the sys.argv property that contains the command line arguments. The name of the module is stored in the first position sys.argv[0], the first argument is at position 1 etc. The simplest way to use this is:

```
# filename: count_txt.py (cont.)
import sys  # Add sys to read the command line arguments

if __name__ == "__main__":
    directory = sys.argv[1]  # Get the first command line argument
    number_of_files = count_txt(directory)

    print("The number of txt files are %s." % number_of_files)
```

You can now run the script from the command line (changing of course the path to a real directory):

```
python count_txt.py /home/user/mydir/
The number of txt files are 3.
```

CHAPTER 13

Learning resources

13.1 Online tutorials

- Python tutorial: http://docs.python.org/tutorial/
- Numpy tutorial: http://www.scipy.org/Tentative_NumPy_Tutorial
- Lecture notes for scientific python: http://scipy-lectures.github.com/index.html

13.2 Books

- A Primer on Scientific Programming with Python

http://www.amazon.com/Scientific-Programming-Computational-Science-Engineering/dp/3642183654/

- Python Scripting for Computational Science

http://www.amazon.com/Python-Scripting-Computational-Science-Engineering/dp/3642093159/

13.3 Modules

- Numpy: http://numpy.scipy.org/
- Scipy: http://www.scipy.org/
- Matplotlib: http://matplotlib.sourceforge.net/index.html

CHAPTER 14

Appendix 1: The Mercurial version control system

Soon after you start writing your first code, you will face the problem of keeping track of all the versions of the file you create. In the beginning, this will probably not be a major issue. For example, every time you want to reuse some old file you could make a copy with a new name. However, this does not work well in the long run. You soon end up with files name *myfile.py*, *myfile_new.py*, *myfile_new_new.py*, *myfile_test.py*, and so on. Does this sounds familiar? In worst case, you forget to make a copy of the old version, and some of your old code sounds! If this sounds familiar, you are not alone. This is a major problem of almost all most researchers.

Keeping track of your code is an important issue. Coding needs a lot of effort and patience, and loosing one of your scripts can easily cost you several days of work. Equally importantly, your code is the best documentation of all your processing steps. If you lose some of your old code, or if you make changes that you don't keep track, you will not be able to reproduce your research results in case a colleague, or reviewer, asks. Luckily, there are several established systems, called *version control systems*, that will help you to keep track of your ever-evolving code. In this section, we will briefly introduce one of the most popular such system call *Mercurial*.

Note: In this section we will focus on the *Mercurial* version control system, but this is by no means your only choice. **Git**, for example, is a very popular and powerful version control system used by many open source projects.

You can find more information about *git*, in the following links:

https://git-scm.com/ All the information you need about git.

https://github.com/ A web-based Git repository hosting service.

https://try.github.io/levels/1/challenges/6 A hands-on course on git.

https://www.codecademy.com/learn/learn-git Another hands-on course.

14.1 Mercurial overview

Mercurial, often abbreviated as **hg**, is a software tool that can help you manage the version of your code, and also collaborate with others. Mercurial can be used from the command line, but there are several graphical user interfaces that can get you started. One excellent choice for beginners is *EasyHg <http://easyhg.org/>*_, that is designed to be easy to use and understand.

[TBD]

14.2 Setting up a repository

14.3 Committing some changes

14.4 Update to the previous version

14.5 Working with a remote repository

See also:

Hg init An introduction to Mercurial.

EasyHg start video A short video showing you how to put your files in a repository using EasyHg.

Version control intro video A simple introduction to version control systems. Not for Mercurial, but highlight the main concepts.

CHAPTER 15

Indices and tables

- genindex
- search