KIS
Kiepenheuer-Institut
für Sonnenphysik
Stiftung des öffentlichen Rechts des Landes Baden-Württemberg
Mitglied der Leibniz-Gemeinschaft

Astrophysical Lab Course

Introduction to Python

1 Introduction

Python is a widely used open source programming language. It's structure is easy to read and by the use of modules it provides an easy way to integrate useful code from others into your programs. Since it is free to use you can install it on your private computer and do the data handling and analysis for the lab course by yourself.
This tutorial is intended to show how to write and run simple functional code. In order to learn try running the code examples yourself and modifying some of the code to really understand what it does.
If you already have programming experience we recommend that you still do the exercise at the end of this tutorial in order to get an idea of how it can be used in astrophysics.
Some useful scripts and functions are also provided.

2 Setting up Python

Go to `https://www.python.org/` to download Python or on Linux you may use the package manager that comes with your operating system. Make sure to download and install Python 3.

2.1 Useful modules

After installing Python you should install some modules. They are packages of code that provide a lot of useful functions, so that you don't have to write the code yourself. You can find countless modules on the internet. In the lab course we will be using the following.

- numpy
- scipy

- matplotlib
- astropy
- photutils
- easygui

When you are missing modules needed for the code you are trying to run Python will tell you which ones.

In order to install these modules you can use the package manager pip. Use the command
`pip3 install SomePackage`
from the Terminal/CMD/...
On Debian-based Systems (e.g. Ubuntu) it is recommended to use the user flag when installing:

```
pip3 install -user SomePackage
```

You will also need the package **ipython**. iPython is a command shell for interactive Python computing. It is not a module but can still be installed via the package manager.

```
pip3 install ipython
```

3 Getting started

3.1 Starting Python

In the lab course we will be using iPython with Python 3. To start simply type `ipython` (or `ipython3` if you also have Python 2 versions installed) into your Terminal/CMD/... .

3.2 Python Basics

For the basic knowledge of variables, operations and loops we would like to point to the Scipy Lecture Notes chapter 1.2.1. to 1.2.3.4.:
`http://www.scipy-lectures.org/intro/language/python_language.html`

3.3 Functions

In Python you can define Functions which you can call from the iPython shell. These can depend on input variables and can return output values. Two short examples are given below. The `#`-symbol defines the beginning of a comment over the rest of the line.

```
def greeting():       #definition of the function, it does not depend on variables
   print('HELLO!')    #the code that belongs to the function is indented by a tab

def div(a, b = 12):   #this function depends on variables; b is predefined
   c = a / b
   return c           #returns the value of c; without return nothing is stored
```

Make sure that you always use a **colon** and **indentation** to define which part belongs to the function.

Note that the function `div` depends on two variables, `a` and `b`, but one of them has a preset value. If you only enter one value when calling the function this will be the value for `a` and `b = 12` will be used. If you enter two values `b` will take the value of the second variable.

```
div(a = 12)          #b is predefined, so we only need one input value

div(12,1)            #12 is assigned to the first-named variable (a) in the
                     #definition of div
div(b = 1, a = 12)   #if the variables are named the order does not matter

a = 1                #this variable is different from the variable a inside the
b = 12               #definition of div even though they use the same name
div(a=b, b=a)        #the value of b (12) is assigned to the variable a of div
div(b, a)            #the same as above, but shorter
```

All these inputs will result in the same output. Note that the variables `a` and `b` outside the function are unrelated to `a` and `b` inside the definition of `div`. Since there is no variable given in which to store the output, it will simply be printed, but not saved. In order to store the output use

```
d = div(12)  #again c is only defined inside div, only its value is returned
```

A function can also have multiple return values. The example below shows how to store three values into separate variables.

```
a, b, c = otherfunction()
```

3.4 Scripts

A script is a chunk of code that is saved as a `.py` file. It can be run from inside iPython with:

```
run path/file.py
```

Note that when starting iPython you are working from the directory that you were currently at in the console. Paths can be relative or absolute.
Scripts can contain plain code and/or functions. If you run a script it is interpreted as if you typed every line into the console. Functions defined inside a script become callable outside after running it. To use a function from a script you can type

```
run path/file.py
function()
```

It is also possible to call a function inside a script as long as it is defined in the same one.
In order to use modules, e.g. Numpy, we have to import them at the beginning. Below is an example of a script that can be run from the console.

```
import numpy as np        #imports numpy; to use a numpy function type np.'function'
def mult12(input):        #an arbitrary function
   out = input * 12
   return out
def mylog(input):
   b = np.log2(input/4)   #uses the function log2 from numpy
   return b
a = 7                     #define some variables; these are stored when running the
b = 14                    #script since they are not defined inside of a function
c = mult12(a)
d = mylog(16)
```

4 Python in the lab course

4.1 Working with arrays

We will be working with arrays a lot. The module Numpy provides many useful functions in order to manipulate these. Some are shown below as well as some basic steps.

```
import numpy as np                  #import module numpy and call it np now;
                                    #calling functions from numpy can be done with
                                    #np.function
a = np.zeros(shape=3, dtype = int)  #a is a one-dimensional array consisting of
                                    #floating point numbers; all values are 0
b = np.ones(shape=(5,3))            #b is a two dimensional array of ones
b[0,0] = 3.2                        #the value of b at index (0,0) is set to 3.2
b[1:3,1] = 2                        #the values at indexes (1,1), (2,1) are set to 2
```

```
b[:,2] = 12                     #the values at index(i,2) for all i in the range
                                #of the defined dimensions of b are set to 12
print(b.shape)                  #prints the lengths of all dimensions of b
print(b.shape[0])               #prints the length of the first dimension of b
np.savetxt('filename.txt', b)   #saves the array b into a .txt file; the
                                #filename must be a string variable, make sure
                                #to use quotation marks
c = np.loadtxt('filename.txt')  #opens the .txt file and stores it into c
```

4.2 Working with FITS files

FITS files can contain multiple images and observation data stored in a header. In order to open and manipulate FITS files we need to import the fits module from Astropy.

```
from astropy.io import fits
```

Now we can read and write image data as well as headers.

```
def func(data='path_to_file'):
   #first read a file and its header
   image = fits.getdata(data)
   header1 = fits.getheader(data)

   #now create our own file and write it keeping the header
   image2 = np.ones(shape=(image.shape), dtype=int)
   hdu = fits.primaryHDU(image2)
   hdu.writeto('output_path', header=header1, overwrite=True)
   #a header is not required to save a fits file
```

Remember to always define the `primaryHDU` before you write your data into a FITS-file.

When opening a 2D-FITS-file (one image) the **first** axis of the array is the **y-axis**, the **second** one is the **x-axis**.
If you are working with a file containing multiple images the first axis represents the number of the image, the second one is the y-axis and the third one is the x-axis.

4.3 Plotting

To plot arrays you can use the module PyPlot from Matplotlib:

Matplotlib is a very powerful tool that can be used in many different ways. In order to use it properly you have to define the correct back end for your use. Instructions for the usage in the

lab course are below. We will not explain back ends in this tutorial, you can read more about them in the matplotlib documentation.
`https://matplotlib.org/faq/usage_faq.html#what-is-a-backend`

For using matplotlib in iPython simply type

```
%matplotlib
```

directly after starting. If you forget it and run a script iPython might crash.

Now we can import the tools we need:

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

With Matplotlib we are able to display graphics. For two-dimensional x,y-plots use `plot`, for 2D-images `imshow`. To have different windows and plots open at once define the plots with `figure`

```
plt.figure(1)                    #open window '1' to plot into (arbitrary number)
plt.imshow(2d_array)             #load the image data
plt.show()                       #display it
plt.figure(2)                    #open window '2' and work in it
plt.plot(x_array, y_array, '-')  #plot y over x (shape(x)=shape(y)); - defines
                                 #the style, it is a line connecting all data points
plt.plot(x2, y2, 'r.')           #another plot in the same coordinate system;
                                 #style is red dots
plt.show()                       #display everything in figure '2'
plt.figure(1)                    #back to working in figure '1'
plt.xlabel('x-axis')             #label the x-axis
```

PyPlot has a graphical UI in which you can zoom, return to the original view, or save your image as a png. You can also set your axes and save manually from iPython:

```
plt.axis([xlow, xhigh, ylow, yhigh])  #set the range of the axes
plt.xlabel(x_axisname)                #label the axes
plt.ylabel(y_axisname)
plt.savefig('path/file.png')          #save image as a png
```

4.4 Polynomial fit

Numpy provides an easy way to fit polynomials to our data and we can use matplotlib in order to visualize. Here we provide an example of an parabolic fit (second order polynomial fit).

```
x = [-0.6, 1.2, 1.5, 2.3, 2.4]       #x-axis of data
y = [-0.5, 2.80, 4.6, 10.3, 11.7]    #y-axis
plt.figure(1)
plt.plot(x,y, '.')                   #plot it
plt.show()
coefficients = np.polyfit(x,y,2)     #returns the coefficients as an array
polyfunc = np.poly1d(coefficients)   #creates a function that is callable
xd = np.linspace(-1,3,50)            #creates an array of 50 evenly spaced values
                                     #between -1 and 3.
plt.plot(xd, polyfunc(xd))           #use xd to get more data points;
                                     #using x would only return shape(x) points
plt.show()
```

5 Optional tasks

Below is an exercise to learn the programming methods that will be used in the lab course.
Please do not work on this exercise during the lab course if you already have actual data to analyze.

Correcting Data with flat-field and dark images
If you have image data taken with a camera through a telescope there are errors induced by both the camera and the telescope. Some of them can be corrected by the method that you will learn in this exercise.

A flat-field image is an equally illuminated image. It is taken with the exact same setting as the experiment except for the illumination. All structure seen in the flat-field image corresponds to contamination in the optical system. Most prominent is the visible dirt on the camera, it also contains vignetting and other unwanted errors.

A dark image is taken to correct readout noise and dark current of the camera. One is taken with the same exposure time as the flat-field and one with the same as the science data (unless they are the same).

To correct image Data one can create a so-called gain-table G. The Formula is below:

$$G = \frac{1}{F - D_F} \cdot \langle F - D_F \rangle$$

Where F is the flat-field image, D_F the dark for the flat-field and $\langle F - D_F \rangle$ the spacial average of $F - D_F$. Usually F and D are created as averages of multiple dark and flat-field images.
With G we can correct our Data:

$$I_{Corr} = (I - D_I) \cdot G$$

Where I is the image we want to correct.

You can find all files needed for the following tasks in the `'first steps'` directory. In our case the exposure times are the same so we do not need different dark images.

Tasks

1. Load all the fits data into variables with Python
2. Average the dark images so that you receive one 2D-image (Hint: Create an array containing the dark files and use np.mean(array, axis=x); what is the right value for x?)
3. Save the average dark as a FITS file
4. Average the flat-field images into one 2D-image

5. Create a gain-table (Hint: np.mean() can take multiple axes at once: axis=(x,y,z); what are the right values?)
6. Save the gain-table
7. Correct your Image with the average dark and the gain-table
8. Display the original image, the gain-table and the corrected image with matplotlib

(Version: 15. Juli 2018, S. Haulitschke)