Numpy, get rid of the Mathesaurus

author: Alessandro Ferrari

email: alessandroferrari87@gmail.com

I. INTRODUCTION

Many of us are scientists or former scientists. Any kind of scientist you are or you will be, you will need to develop, at least, basic data scientists skills. You will need to effectively read, handle, manipulate and visualize data for your professional activities, either you want to monitor vibrations of a mechanical component or to show trends of an advertising campaign of the company that you work for. In order to do so, nobody of us wants to re-invent the wheel, luckily there are easy to use and efficiently written software libraries that provides us the basic bricks for analysing our data.

Many of you are probably more acquainted with technologies for data analysis such as Matlab. Numpy and Python frameworks are definitely not just the cheaper alternative to the Matlab environment. While offering an outstanding variety of different scientific tools, that beats competitors in many different applications, *i.e.* image analysis or machine learning, they mantain the same easiness for prototyping as Matlab, but freeing the user to the burden of a heavy development environment (the Mathesaurus).

Python is a language created by hackers for hackers, its power is comparable to its easiness. By developing solutions with it, you can create solution that are ready to go to market in the same time that you were spending while doing your prototypes. What are you waiting for? Join the makers movement!

This lab will guide you step by step through the use of numpy to manipulate scientific data, and matplotlib to visualize them.

II. INSTRUCTION TO THE USE OF THIS LAB

By this lab, you can learn many precious stuffs that can make the difference in your professional career as a technology professional. When you do things actively you can memorize also ten times more, so make yourself a gift, be active! There are few rules that you need to follow:

- Do you have numpy documentation website open? If you do not, I will not help you.
- Did your lab mate ask you a question without having opened the numpy website documentation? Do not help him.
- Are you having troubles on finding numpy documentation online? Click here.
- You will find the code of the guided track in the folder of the lab, so you can avoid to spend time while copy/pasting functions. Try to understand the different topics and then get involved in the exercises so that you can learn how to apply what you have learnt.
- Be active.
- Enjoy!
- Please, send me feedback to improve the laboratory for next years.

Are you ready? Let's go!

III. NUMPY AND MATPLOTLIB

A. Array objects

NumPy provides an N-dimensional array type, the *ndarray*, which describes a collection of items of the same type. The items can be indexed using for example N integers.

All ndarrays are homogenous: every item takes up the same size block of memory, and all blocks are interpreted in exactly the same way. The type of the items in the array is specified by an attribute called *dtype* that is in every *ndarray*. Thus, ndarrays are not as python lists, that are heterogeneous, so that can contains items of different types.

An item extracted from an array, *e.g.*, by indexing, is represented by a Python object whose type is one of the array scalar types built in Numpy. The array scalars allow easy manipulation of also more complicated arrangements of data.

First of all, you need to import the numpy module:

```
import numpy as np
```

It is renamed as np for a convenient convention.

Then, you can create your array, by specifying its elements in a python list:

```
#### NDARRAY CREATION ####
print "Create a 1D ndarray:"
a = np.array([0, 1, 2, 3])
print "a = %(a)s.\n" % {"a":a}
```

A numpy array can contain an audio signal, an image, an hyperspectral hypercube or any other kind of digital signal. The ndarray object contains an attributes called *ndim* that indicates the number of dimensions of the array, and an attribute *shape* that contains the cardinality of the array in each dimension:

```
### NDARRAY NUMBER OF DIMENSIONS AND SHAPE ###
print "Access to the ndim and shape attributes of a ndarray:"
print "a.ndim = %(ndim)s" % { "ndim" : a.ndim }
print "a.shape = %(shape)s" % { "shape" : a.shape }
print "len(a) = %(len)s.\n" % { "len" : len(a) } #len return the length of the first dimension
```

Note the syntax used for string formatting, it can be useful when you need to print more complicated strings. For creating arrays biggers than 1 dimension, you can use the following syntax:

```
### 2D NDARRAY ###
print "Creating a 2D ndarray:"
b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
print "b = %(b)s" % {"b":b}
print "b.ndim = %(ndim)s" % {"ndim":b.ndim}
print "b.shape = %(shape)s" % {"shape":b.shape}
print "len(b) = %(len)s.\n" % {"len":len(b)}

### 3D NDARRAY ###
print "Creating a 3D ndarray:"
c = np.array([[[1], [2]], [[3], [4]]])    # 2 x 2 x 1 array
print "c = %(c)s" % {"c":c}
print "c.ndim = %(ndim)s" % {"ndim":c.ndim}
print "c.shape = %(shape)s" % {"shape":c.shape}
print "len(c) = %(len)s.\n" % {"len":len(c)}
```

What about create a 5D array? Try on your own!

To enter the items of the array one by one is rather unpractical in many cases. Some functions to generate numpy arrays without specifying the items will be introduced. The function *arange* creates a numpy array containing a sequence of evenly spaced items. The number of items is specified as a parameter. By default, the starting item of the sequence is 0 and the step between each item is 1.

```
### NDARRAY WITH np.arange ###
print "Construct a ndarray with np.arange:"
a = np.arange(10)
print "np.arange(10) = %{a}s" % {"a":a}
```

What about if you want a sequence that do not start from 0 and has step different than 1?

```
### NDARRAY WITH np.arange WITH MORE PARAMETERS ###
print "Use np.arange with more parameters:"
b = np.arange(1,9,2) # start = 1 ; end = 9 ; step = 2
print "np.arange(1,9,2) = %(b)s.\n" % {"b":b}
```

The function *linspace* creates a numpy array containing a sequence with start, end and number of points in between specified as parameters.

```
### NDARRAY WITH np.linspace ###
print "Construct a ndarray with np.linspace:"
a = np.linspace(0,1,6) # start = 0 ; end = 1 ; number of points in between = 6
print "np.linspace(0,1,6) = %(linspace)s" % {"linspace":a}
print "    ... and if you want to omit the endpoint in the sequence:"
b = np.linspace(0,1,6,endpoint=False)
print "np.linspace(0,1,6,endpoint=False) = %(linspace)s.\n" % {"linspace":b}
```

Then, there are some functions to create common arrays, such as the ones that you are familiar with in matlab. The function *zeros* creates an array of 0 with size specified as a parameter, the function ones does the same, but with 1 instead of 0. The function *eye* creates an identity matrix with diagonal of length specified as a parameter, while the function *diag* creates a diagonal matrix with values of the items on the diagonal specified in a list passed in input as a parameter.

```
### NDARRAY WITH np.zeros ###
print "Construct a ndarray with np.zeros:"
a = np.zeros( (3,3) )  #the size of the zeros array is specified with a tuple
print "np.zeros( (3,3) ) = %(zeros)s.\n" % {"zeros":a}

### NDARRAY WITH np.ones ###
print "Construct a ndarray with np.ones:"
b = np.ones( (3,3) )  #the size of the ones array is specified with a tuple
print "np.ones( (3,3) ) = %(ones)s.\n" % {"ones":b}

### NDARRAY WITH np.eye ###
print "Construct a ndarray with np.eye:"
c = np.eye(3)  #the number of ones in the diagonal in the identity matrix is specified as a parameter
print "np.eye(3) = %(eye)s.\n" % {"eye":c}

### NDARRAY WITH np.diag ###
print "Construct a ndarray with np.diag:"
d = np.diag([1,2,3,4]) #the items in the diagonal are specified as a list
print "np.diag([1,2,3,4]) = %(diag)s.\n" % {"diag":d}
```

When performing simulations, the creation of randomly generated arrays can be useful. The function *seed* initialize the random numbers pseudo generator, if the seed is not specified, by default it is None and the pseudo random generator is initialized by looking at the system time. The function *rand* generates a vector containing items distributed according a uniform probability distribution between 0 and 1. The function *randn* generates a vector containing items distributed according a gaussian probability distribution.

```
### INITIALIZE THE PSEUDO RANDOM NUMBER GENERATOR WITH np.random.seed ###
print "Initialize the pseudo random number generator with np.random.seed ..."
np.random.seed()

### RANDOM NDARRAY WITH np.random.rand ###
print "Construct a ndarray with np.rand:"
a = np.random.rand(4) #vector of 4 items chosen according to a U[0,1] prob. distribution
print "np.random.rand(4) = %(rand)s.\n" % {"rand":a}

### RANDOM NDARRAY WITH np.random.randn ###
print "Construct a ndarray with np.random.randn:"
b = np.random.randn(4) #vector of 4 items chosen according to gaussian dist., mean = 0, sigma = 1
print "np.random.randn(4) = %(randn)s.\n" % {"randn":b}
```

B. Basic numpy data types

Numpy arrays require uniformly typed items and they supports only certain defined types. In this subsection you will learn about the *dtype* attribute of the ndarray and you will be introduced to the most common numpy data types.

```
### dtype ATTRIBUTE IN ndarray ###
print "Check the type of the items by means of dtype attribute:"
a = np.array([0,1,2,3])
print "a.dtype of a = np.array([0,1,2,3]) is %(dtype)s." % {"dtype":a.dtype}
b = np.array([0.,1.,2.,3.])
print "b.dtype of b = np.array([0.,1.,2.,3.]) is %(dtype)s." % {"dtype":b.dtype}
print "You can also explicitely specify the type of the array that you are creating:"
c = np.array([0,1,2,3],dtype=float)
print "c.dtype of c = np.array([0,1,2,3],dtype=float) is %(dtype)s." % {"dtype":c.dtype}
print "The default data type is float:"
d = np.ones((3,3))
print "d.dtype of d = np.ones((3,3)) is %(dtype)s." % {"dtype":d.dtype}
print "Following some other types:"
e = np.array([1+2j, 3+4j, 5+6*1j])
print "e.dtype of e = np.array([1+2j, 3+4j, 5+6*1j]) is %(dtype)s." % {"dtype":e.dtype}
f = np.array([True, False, False, True])
print "f.dtype of f = np.array([True, False, False, True]) is %(dtype)s." % {"dtype":f.dtype}
g = np.array(['Bonjour', 'Hello', 'Hallo',])
print "g.dtype of g = np.array(['Bonjour', 'Hello', 'Hallo',]) is %(dtype)s." % {"dtype":g.dtype}
print "It is a string of maximum 7 characters!"
print "Some others useful types are int8, int16, int32, int64, uint8, uint16, uint32, uint34, float32 and many
    others. uint8 and float32 are especially useful if you use OpenCV library python bindings.\n"
```

C. Visualizing data

In this subsection, matplotlib will be briefly introduced. Matplotlib is probably the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats. Let me say that matplotlib is really a jewell for visualizing data, you will love it when you will start to get use to it. Furthermore, the potential of the library goes further beyond respect what it will be shown today. It has also some basic functions for 3d plotting, however if you want to plot 3d stuffs, you should evaluate more specialized modules as mayavi. Some examples of 1D plots (fig.1) follow:

```
#IMPORT THE MATPLOTLIB MODULE
import matplotlib.pyplot as plt  # the tidy way

print "Basic usage of matplotlib for 1D plotting:"

#POPULATE THE X AND Y ARRAY FOR PLOTTING THE GRAPH
x = np.linspace(0, 3, 20)
y = np.linspace(0, 9, 20)

#1D LINE PLOT OF THE GRAPH
plt.plot(x, y)       # line plot
plt.show()           # <-- shows the plot

#1D POINTS PLOT OF THE GRAPH
plt.plot(x, y, 'o')  # dot plot
plt.show()           # <-- shows the plot
```

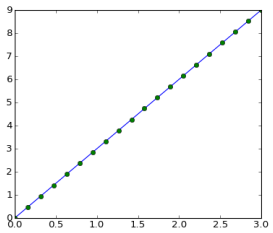

FIG. 1: Example of 1D plots.

An example of 2D plotting, in particular a function for showing images (fig.2), is shown in the code below:

```
print "Basic usage of matplotlib for 2D plotting:"
image = np.random.rand(30, 30)
plt.imshow(image, cmap=plt.cm.hot)
plt.colorbar()
plt.show()
```

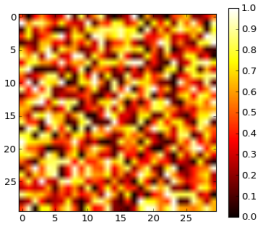

FIG. 2: Example of 2D plots.

These are just the most basic stuffs, however, in order to give you a glimpse of the stuffs that you can do with matplotlib, take a look to fig.3, that is possible to find at the end of the lab track.

D. Indexing and Slicing

Numpy arrays are zero-based (differently from Matlab array, that are 1 based). The items of an array can be accessed using indexing at the same way as other Python sequences (*e.g.* lists):

```
### INDEXING OF NUMPY ARRAYS ###
print "Indexing of numpy array's items:"
a = np.arange(10)
a_items = (a[0], a[3], a[-1]) #the index equals to -1 points to the last item
print "(a[0], a[3], a[-1]) = %(tuple)s where a = %(arange)s.\n" % {"tuple":a_items,"arange":a}
```

For multidimensional arrays, indexes are tuples of integers:

```
### MULTIMENSIONAL INDEXING OF NUMPY ARRAYS ###
print "Multidimensional indexing of numpy array's items:"
a = np.diag(np.arange(3))
print "a[1,1] = %(a11)s where a = %(a)s" % {"a11":a[1,1],"a":a}
print "Assigning of multidimensional numpy arrays:"
a[2,1]=10
print "if a[2,1] is assigned equal to 10, where a was %(a)s, then a = %(a_new)s.\n" % {"a":np.diag(np.arange(3)),"
    a_new":a}
```

Numpy arrays, like other Python sequences can also be sliced:

```
### SLICING OF NUMPY ARRAYS ###
print "Slicing of numpy arrays:"
a = np.arange(10)
a_sliced = a[2:9:3] # [start:end:step]
print "a[2:9:3] = %(a_sliced)s where a = %(a)s. The order of the indices used to slice the array is relatively [
    start:end:step]. If the step index is omitted, by default is equal to 1." % {"a_sliced":a_sliced,"a":a}
a_sliced_2 = a[:4] # [0:end:1]
print "a[:4] = %(a_sliced_2)s where a = %(a)s. The start index is missing, so the slicing start by default from the
    index 0. Also the step index is missing, so by default the step is 1." % {"a_sliced_2":a_sliced_2,"a":a}
a_sliced_3 = a[1:3]
print "a[1:3] = %(a_sliced_3)s where a = %(a)s. The step index is missing, so by default is missing." % {"
    a_sliced_3":a_sliced_3,"a":a}
a_sliced_4 = a[::2]
print "a[::2] = %(a_sliced_4)s where a = %(a)s. The start is set by default at the beginning of the vector, the end
    to the end of the vector, while the step is set to 2." % {"a_sliced_4":a_sliced_4,"a":a}
a_sliced_5 = a[3:]
print "a[3:] = %(a_sliced_5)s where a = %(a)s. In this case, the vector is sliced starting from the 4th item till
    its end, with step 1." % {"a_sliced_5":a_sliced_5,"a":a}
print "What about if you want to reverse the access order of the sequence?"
print "a[::-1] = %(a_reversed)s where a = %(a)s. The step index set to -1 reverse the accessing order of the
    sequence.\n" % {"a_reversed":a[::-1],"a":a}
```

It is possible to combine also slicing and assigning:

```
print "How to combine slicing and assigning:"
a[5:]=10
print "a = %(a)s, if the assignement is performed a[5:]=10.\n" % {"a":a}
```

Look at fig.4 for having a graphical idea of the slicing.

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```

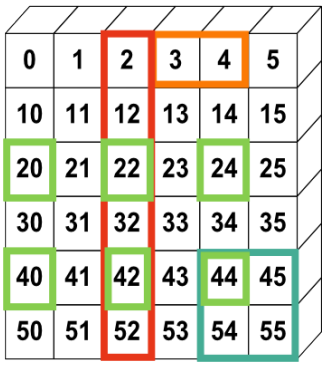

FIG. 3: Graphical examples of numpy slicing.

This slicing does look really useful for cropping images. Doesn't it?

E. Copies and Views

A slicing operation creates a view on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. You can use np.may_share_memory() to check if two arrays share the same memory block. Note however, that this uses heuristics and may give you false positives. When modifying the view, the original array is modified as well:

```
print "Explicit copy of a numpy array:"
a = np.arange(10)
b = a[::2].copy()
print "a = np.arange(10) and b = a[::2].copy().Do they share memory? %(bool)s.\n" % {"bool" : np.may{\_}share{\_}
    memory(a, b) }
```

If you want to force the copy of an array:

```
print "Explicit copy of a numpy array:"
a = np.arange(10)
b = a[::2].copy()
print "a = np.arange(10) and b = a[::2].copy().Do they share memory? %(bool)s.\n" % {"bool" : np.may{\_}share{\_}
    memory(a, b) }
```

F. Fancy indexing

Numpy arrays can be indexed with slices, but also with boolean or integer arrays (masks). This method is called fancy indexing. Let me say that fancy indexing is really fancy. Differently from slicing it creates copies not views.

```
print "Fancy indexing:"
np.random.seed(3)
a = np.random.random_integers(0, 20, 15)
mask = (a % 3 == 0)
print "a = %(a)s" % {"a":a}
print "The mask contains whether the value of an item is multiple of 3 or not. mask = (a % 3 == 0) = %(mask)s" % {"
    mask":mask}
print "a[mask] = %(a_masked)s.\n" % {"a_masked":a[mask]}
```

This looks real fun if you have to mask an image or if you want to extract values from a ROI (Region of Interest). Doesn't it?

You can perform fancy indexing also using python lists of integer indeces:

```
print "Fancy indexing with integers arrays:"
a = np.arange(0, 100, 10)
print "a = np.arange(0,100,10) and a[[2,3,2,4,2]] = %(a_indexed)s" % {"a_indexed" : a[[2, 3, 2, 4, 2]]}
```

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:,[0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45]])
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)
>>> a[mask,2]
array([2,22,52])
```

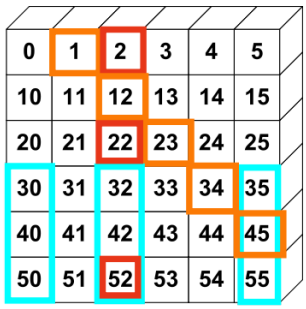

FIG. 4: Graphical examples of numpy fancy indexing.

G. Numpy for Matlab users

You have been guided through the first steps of numpy and matplotlib libraries, however the things that you can do with those library are much more. In fact, in this tutorial important aspects such as

array arithmetical operations, array broadcasting, reshaping and resizing are omitted. However, starting from this basic knowledge and the online documentations I am sure you can get really far very fast. For those of you that are use to Matlab and feel lazy about study new technologies, I would suggest to take a look at this address: http://mathesaurus.sourceforge.net/matlab-numpy.html. It contains all the most common Matlab functions translated to the correspondent numpy method. At this web address http://conference.scipy.org/scipy2013/tutorials.php you can find some really interesting and high quality tutorials, where you can improve your knowledge of the libraries.

IV. EXERCISES

A. Exercise 1

Do you like pop art? What about to make your facebook profile picture Andy Warhol style? In this exercise you will have to take a RGB color picture, preferably a your profile picture, and create a Andy Warhol collage as in fig.7.

You may find useful OpenCV python bindings module (cv2), the OpenCV functions imread, cvtColor and merge. Then, you may find useful also the numpy functions hstack, vstack and astype.

Have fun!

FIG. 5: My profile picture.

FIG. 6: My profile picture Andy Warhol style.

B. Exercise 2

Threshold the coins image and set to black the foreground, as in fig.9.
Enjoy!

FIG. 7: Coins pictures.

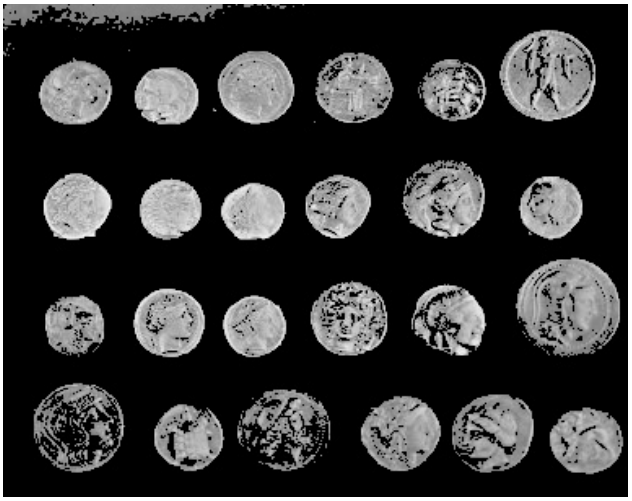

FIG. 8: Coins pictures with foreground to black.

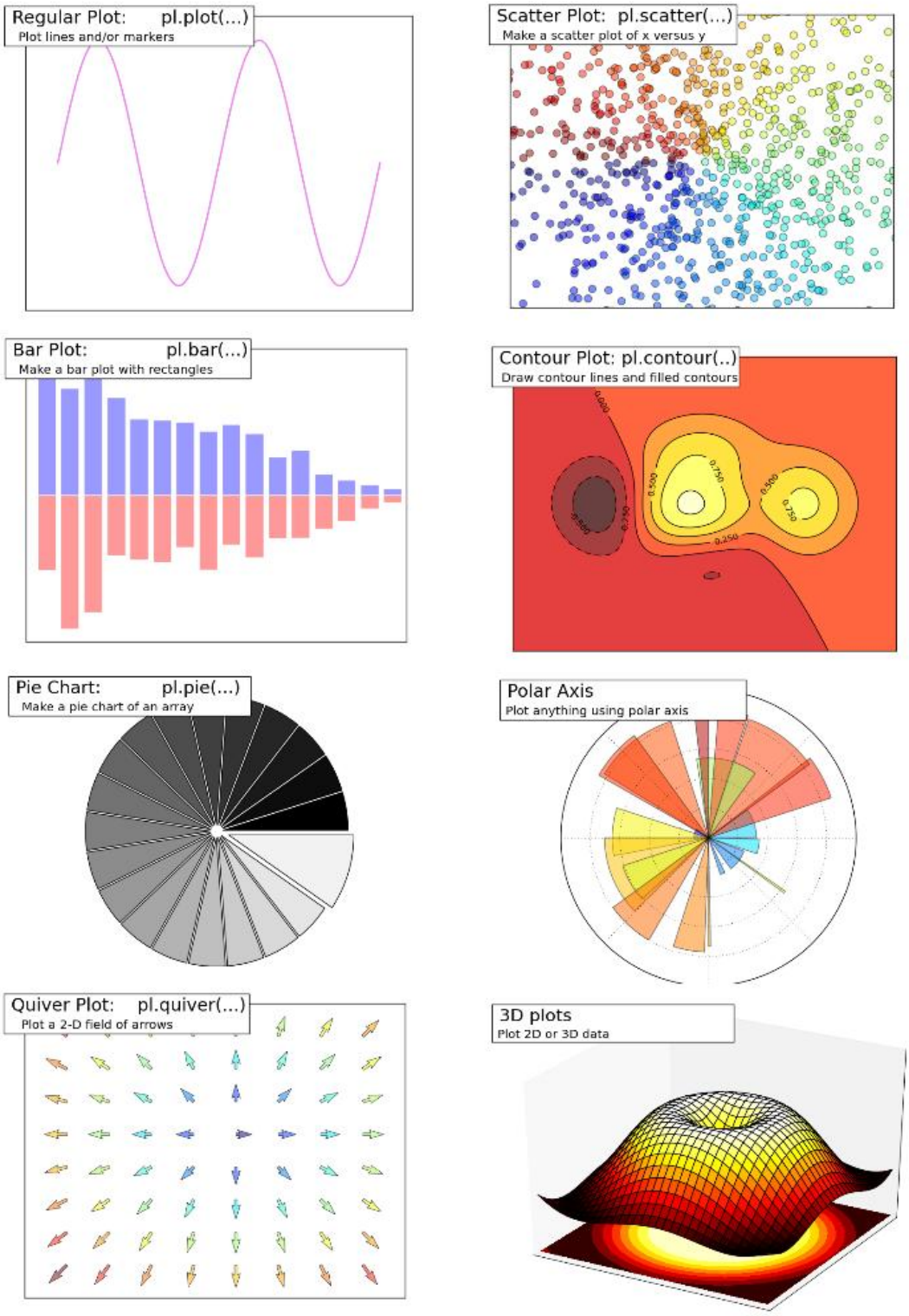

FIG. 9: Matplotlib plotting methods.