simple plots

kacper.topolnicki@uj.edu.pl

The `plots.py` script contains some basic examples of plots that use the `matplotlib` library. The examples below are taken from the official `pyplot` tutorial (link). Please have a look at this tutorial, what follows is an abridged version.

Comments

The script contains many comments. Any lines that begin with the `#` symbol are ignored by python and only contain additional information for the programmer. The script file contains many instances of `#@`, `#@ref`, ... These lines are used by an external programm to create a PDF file, these lines can also be ignored by the programmer.

Running the script

To run the script simply navigate to this directory in the terminal and run:

```
<user> $ python plots.py
```

Alternatively you can make the script executable and run:

```
<user> $ ./plots.py
```

You can also run `ipython` and execute the commands one by one.

Importing the necessary libraries

First we will import the `numpy` library in [plots.py line: 54]

```
import numpy
```

and the `pyplot` module from the `matplotlib` library in [plots.py line: 58]

```
import matplotlib.pyplot as plt
```

Notice that in the second case we will refer to the module using the shorthand `plt`.

Plotting

First we will define some functions to plot. In order to do this we will first create an array with function arguments. If our function is f and has values $f(z)$ and arguments z then the variable `x` [plots.py line: 77]

```
x = numpy.linspace(0 , 2.0 * numpy.math.pi , 100)
```

will contain all the values of z - all the function arguments. In this case the we take 100 numbers in the range from 0 to 2π (`numpy.math.pi`).

Next we need function values. First $f = sin$ [plots.py line: 93]

```
y1 = numpy.sin(x)
```

Notice that `x` is an array of function arguments and the function `numpy.sin` maps the sine function over each element of the array. The result pointed to by `y1` is also an array. We perform similar steps with the cosine function in [plots.py line: 96]

```
y2 = numpy.cos(x)
```

It is time for our first plot. We will use the `plt.plot` (this is an alias for `matplotlib.pyplot`) function [plots.py line: 114]

```
plt.plot(x , y1)
```

If you are running this through `ipython` you might have noticed that nothing happened. This is because we need to run the `plt.show` function [plots.py line: 118]

```
plt.show()
```

This should result in a new window with a plot of the sine function. Have a look at the controls at the top of the window, they allow you to save the plot, zoom, pan, ...

Now a slightly more advanced example. We will give labels to the horizontal (x) and vertical (y) axis. This can be done via the `plt.xlabel` and `plt.ylabel` functions [plots.py line: 129]

```
plt.plot(x , y1)
plt.xlabel("position")
plt.ylabel("amplitude")
plt.show()
```

In the next example we change the width of the line on the plot (`linewidth`) and the style of the line (`linestyle`) [plots.py line: 148]

```
plt.xlabel("position")
plt.ylabel('you can use LaTeX $\phi$')
plt.plot(x , y1 , linewidth = 5.0 , linestyle = "--")
plt.show()
```

Notice also that we can use LaTeX math notation in the labels of the axis. See the `matplotlib` library documentation for more plotting options (optional arguments in python are function arguments in the form `<option name> = <option value>`).

We can also plot two functions on one plot. In the example below [plots.py line: 163]

```
plt.xlabel("position")
plt.ylabel("amplitude")
plt.title("simple plot")
plt.plot(x , y1 , linestyle = "-." , color = "red")
plt.plot(x , y2 , linewidth = 4.0 , linestyle = "--")
plt.show()
```

we plot both the sine function (`y1`) and the cosine function (`y2`). The sine function will be drawn with a dashed dotted red line while the cosine function will be drawn with a thick dashed line.

There are other ways of changing the plot properties. You can use setter functions as in [plots.py line: 183]

```
plot = plt.plot(x , y1 , "-.")
plot[0].set_antialiased(False)
plt.show()
```

Notice that in this case you have to save the plot to a variable (`plot`) and apply the `set_antialiased` method (turns antialiasing on and off) directly to the plot (`plot[0].set_antialiased(False)`). We will talk about this in more detail some other time.

Another option is to use the `plt.setp` function [plots.py line: 199]

```
lines = plt.plot(x , y1 , x , y2)
plt.setp(lines , color = "r" , linewidth = 3.0)
plt.show()
```

Here we set the line color to red and change the linewidth. There are two ways to call this function the second method [plots.py line: 205]

```
lines = plt.plot(x , y1 , x , y2)
plt.setp(lines , "color" , "r" , "linewidth" , 3.0)
plt.show()
```

might be more familiar to users of other plotting software.

Plot legends can be added as in [plots.py line: 222]

```
fig , ax = plt.subplots()
ax.plot(x , y1 , label = "sin")
ax.plot(x , y2 , label = "cos")
```

```
ax.legend(loc = "upper center" , shadow = False , fontsize = "small")
plt.show()
```

In line [plots.py line: 223] the plot is "unpacked" to two variables (`fig` and `ax`). Next labels are added to the plot (`label = "sin"` and `label` = "cos"). Finally the legend is added and placed in the central upper location (`loc = "upper center"`).

Finally we can add annotations to the plot [plots.py line: 242]

```
plt.plot(x , y1)
plt.annotate('point where x = 1 and y = 1' , xy = (1 , 1))
plt.show()
```

The annotation will be inserted into coordinates $(1, 1)$ of the plot. There are other options that can set the coordinate system, you can for instance give the annotation coordinated in relative coordinates where the bottom left of the plot has coordinate $(0, 0)$ and the top right has coordinate $(1, 1)$.