Using Python like Matlab and Mathematica

Adam Watts

This notebook is a beginning tutorial of how to use Python in a way very similar to Matlab and Mathematica using some of the Scientific Python libraries. This tutorial is using Python 2.6. The most convenient way to install everything is to use the Anaconda distribution: https://www.continuum.io/downloads (https://www.continuum.io/downloads).

To run this notebook, click "Cell", then "Run All". This ensures that all cells are run in the correct sequential order, and the first command "%reset -f" is executed, which clears all variables. Not doing this can cause some headaches later on if you have stale variables floating around in memory. This is only necessary in the Python Notebook (i.e. Jupyter) environment.

In [1]:
```
# Clear memory and all variables
% reset -f

# Import libraries, call them something shorthand
import numpy as np
import matplotlib.pyplot as plt

# Tell the compiler to put plots directly in the notebook
% matplotlib inline
```

Lists

First I'll introduce Python lists, which are the built-in array type in Python. These are useful because you can append values to the end of them without having to specify where that value goes, and you don't even have to know how big the list will end up being. Lists can contain numbers, characters, strings, or even other lists.

In [2]:
```
# Make a list of integers
list1 = [1,2,3,4,5]
print list1
# Append a number to the end of the list
list1.append(6)
print list1
# Delete the first element of the list
del list1[0]
list1
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6]
```

Out[2]:
```
[2, 3, 4, 5, 6]
```

Lists are useful, but problematic if you want to do any element-wise math. Here's an example of trying to do element-wise multiplication of a list. Notice that trying to multiply an integer with a list just repeats the list. This can be useful, but it's not exactly what we're going for. Also note that if you try multiplying a float (i.e. "2.1") and a list, the compiler throws an error.

```
In [3]: list1 = [1,2,3,4,5]
        print 2*list1

        [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Numpy Arrays

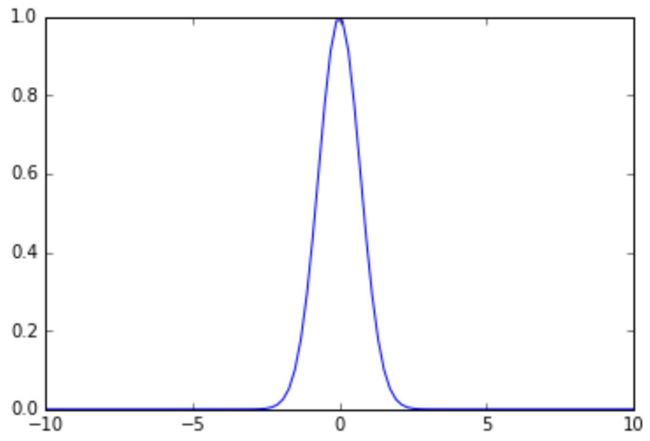
If you want your Python lists to behave like Matlab arrays, then you want to use Numpy arrays. Note that I have already imported the Numpy library above, and used an alias so I can just refer to it as "np" (see first code box).

```
In [4]: # Create a Numpy array
        array1 = np.asarray([1,2,3,4,5])
        print array1
        array1 = 2*array1
        print array1
        array1 = 1.2*array1 + 7
        print array1

        [1 2 3 4 5]
        [ 2  4  6  8 10]
        [  9.4  11.8  14.2  16.6  19. ]
```

Here's an example of when you want to use a Numpy array. If I want to plot a general function, like a Gaussian, then I need to first generate an array of x-axis values. Then I compute the y values as a function of x.

In [5]:
```
# First create Numpy array of x values using the "linspace" function,
# creating an array from -10 to 10 with 100 evenly-spaced points
x = np.linspace(-10,10,100)

# Next, compute the y values as a function of x using the formula for a Gaussian cu
rve
# Note that I am using the Numpy exponential function, so np.exp(stuff) is the same
 as e^(stuff)
# Also note that exponentiation in Python is "**", so x-squared is "x**2"
y = 1*np.exp(-(x**2))

# Now let's use the Matplotlib library to plot y vs. x
# We've already imported and aliased the library as "plt" in the first code box
# I've also given a compiler directive to show plots directly in this notebook: "%
matplotlib inline"
plt.plot(x,y)
```

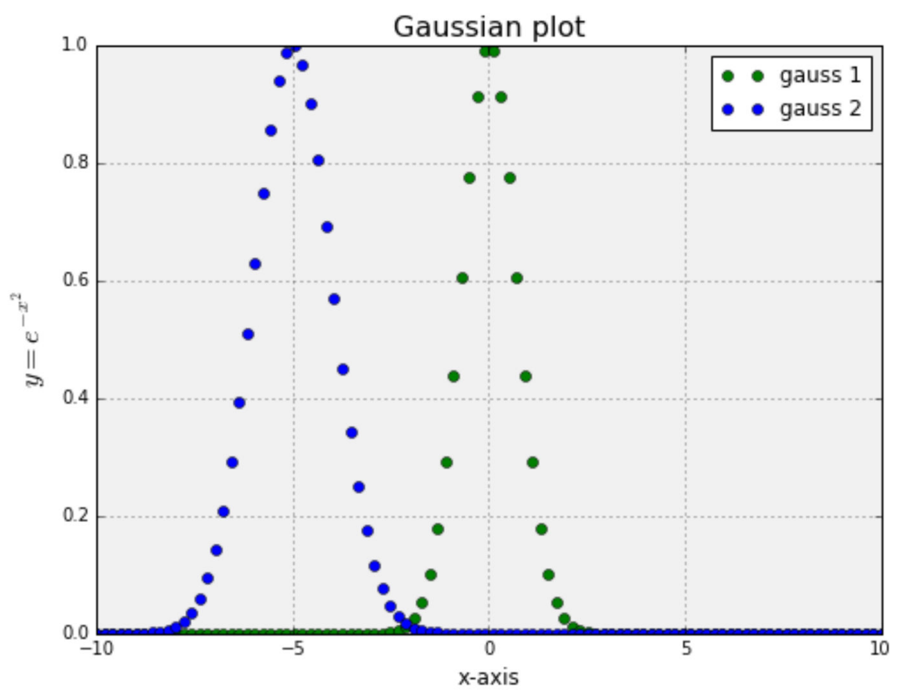
Out[5]:
```
[<matplotlib.lines.Line2D at 0x71d3780>]
```



Plotting with Matplotlib

Building on the previous example, let's use the same Gaussian function to go over some plotting options in the Matplotlib library's Pyplot code. An excellent introduction to Pyplot is here: http://matplotlib.org/users/pyplot_tutorial.html (http://matplotlib.org/users/pyplot_tutorial.html).

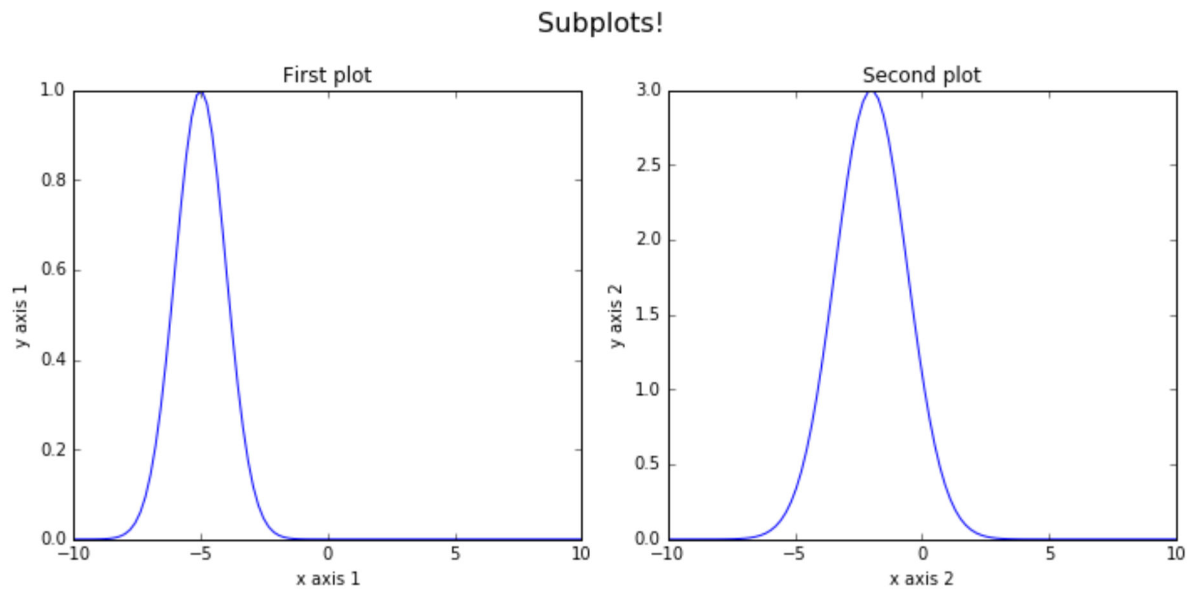
```
In [6]: # Create a "figure" and set its size; this is the container in which our plots sit
        plt.figure(figsize=(8,6))

        # Give the plot a title, with defined font size
        plt.title('Gaussian plot',fontsize=16)

        # Plot the data as before, but this time don't connect points with a line. We'll us
        e
        # circular green markers ('ro') and set the size to something fairly large.
        # I'm giving the plot a label so we can refer to it later with a legend
        plt.plot(x,y,'go',markersize=6.0,label='gauss 1')

        # To show how easy it is to plot multiple curves on the same plot, I'll plot a slig
        htly
        # different gaussian as well
        plt.plot(x,np.exp(-(x+5)**2/2),'bo',markersize=6.0,label='gauss 2')

        # Now we label the x and y axes. I'll get fancy here and label the y-axis with an a
        ctual
        # equation to show LaTeX printing. I do this by using a "raw string", i.e. putting
        an "r"
        # in front of the string, and denoting LaTeX math mode with "$$". I also make the f
        ont size
        # a bit bigger to make it easier to read.
        plt.xlabel('x-axis',fontsize=12)
        plt.ylabel(r'$y=e^{-x^2}$',fontsize=16)

        # Enable a legend if you've labeled your plots. Labelling allows the legend to tell
         the
        # difference between multiple curves on the same plot.
        # I'll specify the location manually, but you can try "loc='best'" and Pyplot will
        try
        # to place the legend so it blocks the least amount of your data points
        plt.legend(loc='upper right')

        # I don't really like the white background, so I typically change it to a light gre
        y to
        # improve contrast and be easier on the eyes. Also, I sometimes like to add in a gr
        id.
        plt.gca().set_axis_bgcolor('#F1F1F1')
        plt.grid()

        # If you want to save the plot as a file in the same directory as the notebook,
        # just use "plt.savefig"
        plt.savefig('plot.png')

        # Note that if you're running this as a standalone script, and not in a Jupyter not
        ebook,
        # you'll actually need to call "plt.show()" to see your plots pop up, or you can ju
        st save them
        # without seeing them first with "plt.savefig".
```



Subplots

Instead of plotting two curves on the same plot, I can split them up into multiple subplots on the same figure. This lets me independently control the axes scale of each plot. Note that while you can set the x and y scaling yourself using *"plt.xlim"* and *"plt.ylim"*, I'm letting Pyplot autoscale in this example.

In [7]:
```
# Create a figure again, this time making room for the second plot.
# First we'll make the plots side-by-side, so the length of the figure
# will be twice the height
plt.figure(figsize=(10,5))

# The "subplot" function takes a numerical argument "ABC" that says
# "this figure has A columns and B rows, and this is the Cth plot
plt.subplot(121)
plt.title('First plot')
plt.plot(x,np.exp(-(x+5)**2/2))
plt.xlabel('x axis 1')
plt.ylabel('y axis 1')

plt.subplot(122)
plt.title('Second plot')
plt.plot(x,3*np.exp(-(x+2)**2/4))
plt.xlabel('x axis 2')
plt.ylabel('y axis 2')

# I can set an overall title for the figure with "suptitle", but
# you'll want to make the fontsize bigger for the suptitle.
plt.suptitle('Subplots!',fontsize=16)
# Things can get cramped in subplots, so we can use the "tight_layout" function
# to work out the padding automatically. However, tight_layout doesn't
# take into account suptitle as of this writing, so we'll move the top of the plot
# up a bit to keep suptitle away from everything else.
plt.tight_layout()
plt.subplots_adjust(top=0.85)
```

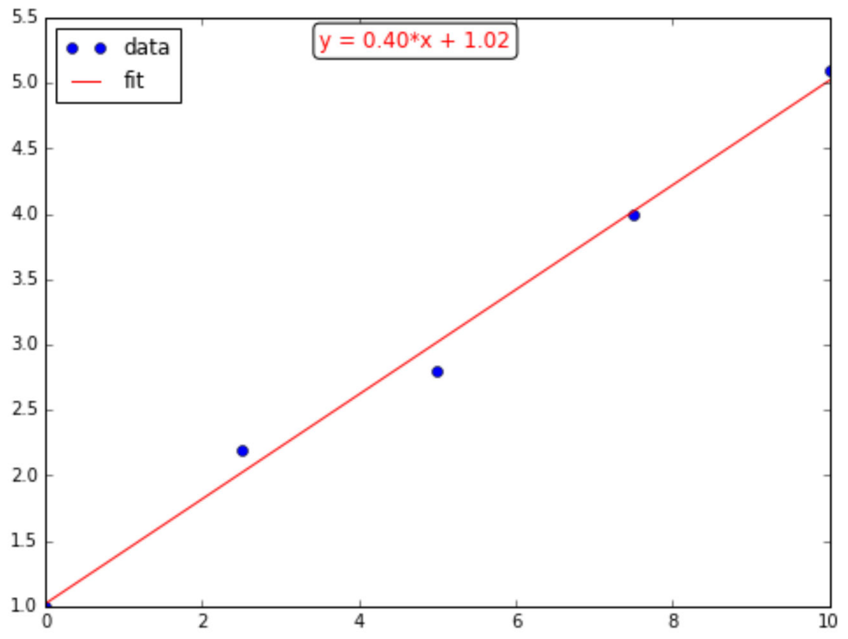


Fitting data

Here I'll give an example of fitting data to a general equation defined by the user. There are libraries specifically for linear fits, normal distribution fits, etc. but this general example is all you'll ever need.

In [8]:
```
# Define the function to which you're trying to fit.
# In this example, I'll start with a simple line parametrized as y=m*x+b,
# but any form can be used. Here, we're passing all fitted parameters into
# the variable "p" for simplicity.
def line(x, *p):
    m,b = p
    return m*x+b

# Import the curve_fit library from SciPy, then define the fitting function,
# which will reference our "line" function
from scipy.optimize import curve_fit
def fit(x,data):
    p0 = [1, 1] # initial guess
    coeff, var_matrix = curve_fit(line, x, data, p0=p0)
    y_fit = line(x, *coeff)
    m = coeff[0]
    b = coeff[1]
    return y_fit, m, b

# Now I'll make an array of x values, then make up some y values as our "data"
# we're trying to fit.
# Note that the length of x and data have to be equal!
x = np.linspace(0,10,5)
y = np.asarray([1.0,2.2,2.8,4.0,5.1])

# Now we run the fit function, which returns the fit y values, the slope, and the
# offset. Then I'll plot both on the same plot to see if it's a good fit.
y_fit, m, b = fit(x,y)
plt.figure(figsize=(8,6))
plt.plot(x,y,'bo',label='data')
plt.plot(x,y_fit,'r-',label='fit')
plt.legend(loc='best')

# Since we're usually interested in the fit parameters too, let's
# print them right on the plot in a fancy box.
# First I build the string to print, using compiler directives to
# round the float fit parameters to 2 decimal places (i.e. "%.2f")
textstr = 'y = %.2f*x + %.2f'%(m,b)
plt.text(0.35,
         0.95,
         textstr,
         transform=plt.gca().transAxes,
         fontsize=12,
         color='red',
         bbox=dict(facecolor='white', edgecolor='black', boxstyle='round,pad=0.3'))
```

```
Out[8]: <matplotlib.text.Text at 0x8a100f0>
```



Symbolic Manipulation

Using the "SymPy" library, we can perform algebraic symbolic manipulation using Python. This includes simplifying large algebraic expressions, solving systems, computing derivatives and integrals, etc. I'm only going to show a small glimpse at what SymPy can do, but it can save a ton of time just on tedious algebra.

Rather than introducing each feature of SymPy slowly, I'm going to dive into an example that derives how to measure beam Courant-Snyder parameters from scanning a quadrupole and measuring the profile after a drift.

For a better description of the physics behind this derivation, see this paper: http://beamdocs.fnal.gov/AD-public/DocDB/ShowDocument?docid=5323 (http://beamdocs.fnal.gov/AD-public/DocDB/ShowDocument?docid=5323).

In [9]:
```
# Import SymPy into the namespace...that means I don't have to
# say "sympy.symbols" to use the symbols function; I can just say "symbols"
from sympy import *

# I'll want to print equations in a readable format, so I'll initialize
# MathJax printing to make everything look good
init_printing(use_latex='mathjax')

# First define the variables we'll use as symbols. This lets SymPy know that
# these are special variables that can be algebraically manipulated, and also
# tells it how they should look when printed (using LaTeX notation if necessary).
L1, L2, k = symbols(['L_1','L_2','kappa'])
s11,s12,s22 = symbols(['\sigma_{11}','\sigma_{12}','\sigma_{22}'])
A,B,C = symbols(['A','B','C'])

# Define the transfer matrix through the quadrupole then the drift
R = Matrix([[1,L1],[0,1]])*Matrix([[1,0],[-k,1]])

# Define the initial beam matrix in terms of the statistical second moments
S = Matrix([[s11,s12],[s12,s22]])

# Compute the new beam matrix after the quad and drift, and simplify the expression
S2 = simplify(R*S*R.T)

# Take the upper-left element of the 2x2 "S2" beam matrix, expand terms out,
# then collect in power of the quadrupole strength "k". In other words, rewrite
# so we have terms in different powers of k: k^0, k, and k^2.
d = collect(expand(S2[0,0]),k,evaluate=False)

# Now set each coefficient for each power in k equal to a variable. The coefficient
# in front of k^2 is A, the one in front of k is B, and the constant term is C. The
n,
# make a system of equations out of the result.
System = [
    Eq(A,d[k**2]),
    Eq(B,d[k]),
    Eq(C,d[k**0])
]

# Finally, solve the system for the original beam second moments (s11, s12, s22)
# and print the solution
soln = solve(System,[s11,s12,s22])
soln
```

Out[9]:
$$\left\{\sigma_{11} : \frac{A}{L_{1}^{2}}, \quad \sigma_{12} : - \frac{1}{L_{1}^{3}} \left(A + \frac{B L_{1}}{2}\right), \quad \sigma_{22} : \frac{1}{L_{1}^{4}} \left(A + B L_{1} + C L_{1}^{2}\right)\right\}$$

The "solve" function returns a Python Dictionary, which is a type of relational variable. In the C programming language, this is called a "associative array". This is convenient, because instead of indexing by element number, I can directly ask for the solution for a particular variable. For example, in the code below I'll just print the solution for the beam divergence into the quadrupole, i.e. "s22".

In [10]:
```
# Ask for the s22 solution only:
soln[s22]
```

Out[10]:
$$\frac{1}{L_{1}^{4}} \left(A + B L_{1} + C L_{1}^{2}\right)$$