**Proteomics Informatics (Spring 2014): Week 10**

**Introductory Matplotlib hands-on**

1. > import matplotlib.pyplot as plt
2. > plt.plot(range(1,11), range(11,21))

> plt.show() ## Non-interactive mode; Shell blocks; No more changes possible.

1. > plt.isinteractive?; plt.isinteractive()

> plt.interactive(True) ## pros and cons; other ways to do it (ipython --pylab)

## In interactive mode every pyplot command triggers a draw operation… for large, complex objects this can be expensive. So, a good strategy is to turn interactive off, make changes, and then force draw to update the plots…

> plt.isinteractive()

1. > import numpy as np

> from numpy import **arange**

> x = arange(0., 5., 0.2) ## Evenly spaced; increment by 0.2

## range and xrange only allow integer increments

> plt.plot(x, x) ## The shell is not blocked now

> plt.hold?; plt.hold(True) ## By default hold is set to True

> plt.plot(x, x\*\*2)

1. > plot(x, x\*\*4, color='r', marker='o') ## **keyword arguments**

> plt.plot(x, x\*\*3, 'ko') ## **shortcut;** plt.colors?; plt.plot? (for line markers/styles)

1. **Other more flexible ways to set properties of a line**
   1. Think of it as a “real” object with properties that can be modified (color, style, thickness etc.)
   2. First clear the axes and figure

> plt.cla(); plt.clf()

* 1. > lineEx, = plt.plot(x, x, 'bD') ## Notice the ‘,’ … list/tuple unpacking concept

## plot() returns a list of lines

## multiple lines can be made in one shot

* 1. Clear the fig/axes and create multiple lines in one shot:

> plt.clf()

> lines = plt.plot(x, x, 'b-', x, x\*\*2, 'bo', x, x\*\*3, 'bD')

> lines (enter)

> first = lines[0]; second = lines[1]

* 1. > first? ; first.<tab>
  2. > first.markers

> first.get\_marker(); second.get\_marker()

> first.get\_markersize(); second.get\_markersize()

> first.get\_xdata(); first.get\_ydata(); first.get\_xydata()

* 1. > plt.setp(first); plt.setp(second) **## to see configurable properties of the obj**
  2. > plt.getp(first) **## current values for all the configurable properties**

> plt.getp(first, 'marker') ## current value of ‘marker’ property

> first? ## will give possible values of configurable properties

* 1. **Methods for updating properties once the line (obj) has been created:**
     1. **setp() function:** allows setting multiple properties at once:

> plt.setp(**first**, ls='-', marker=None) ## object followed by key-value pairs

> plt.setp(**second**, marker='s', color='r')

## Note that the figure is automatically updated;

## pyplot/plt automatically calls the draw() function

* + 1. Directly use the object and call its “**set\_\*” methods/functions**

> first.get\_<tab> (methods to get current property values)

> first.get\_color(); first.get\_marker()

> first.set\_<tab> (methods to methods to modify these values)

**Issues:**

1. Only one property can be changed at a time
2. draw() function is not automatically called. Must be explicitly called to update the properties.

(**draw() forces a figure re-draw of the current figure** with current values of all configurable properties of all objects in the figure)

1. plt.setp() is very flexible. Use that for now.
2. **Adding text to figures (Text objects)**
   1. **Main functions**: plt.xlabel(), plt.ylabel(),plt.title(), plt.suptitle(), plt.text(), plt.annotate(), and legends (which are actually composite objects)
   2. **Just like line objects above, the properties of text objects can be controlled using exactly same 3 methods:**
   3. **X/Y-Labels**
      1. **Save the label object as a new variable**

> x\_label = plt.xlabel('X ---->')

> y\_label = plt.ylabel('Y ---->')

* + 1. **Use plt.getp() and plt.setp() functions**

> plt.getp(x\_label) ## Current values of all configurable properties

> plt.getp(x\_label, 'fontsize') ## current values of ‘fontsize’ property

> x\_label? ## will give possible values of configurable properties

> plt.setp(x\_label, fontsize=20, fontstyle='italic')

> plt.setp(y\_label, fontsize=20, fontstyle='italic ', rotation='horizontal')

**Note:** you can set these properties as keyword arguments to the plt.xlabel() and plt.ylabel() function calls…

* + 1. Directly use the object and call its “**set\_\*” methods/functions**

**> x\_label.set\_<tab>; x\_label.get\_<tab>**

🡪 Use various functions to check current or set property values

🡪 As above, if you follow this method, you’ll have to explicitly call the plt.draw() function to force a redraw for changes to take effect

* 1. Similarly properties of ‘**title’** and ‘**suptitle**’ text objects can be configured as per taste.

> plt.title('Y = X\*\*2, X\*\*3') ## **Axis/Subplot title**

> plt.suptitle(' Plotting Various functions', fontsize=25) **## Fig title**

* 1. **Random text on the plot:**

> plt.text? ## to check the documentation

> textObj = plt.text(2, 60, "Playing with the Identity Function", fontsize=30)

> textObj.<tab> to see the functionality

> plt.setp(textObj) ## Configurable properties

> plt.getp(textObj) ## to check current property values

> textObj.remove() ## to remove

* 1. **Legend objects:**

> first.set\_label('Y=X\*\*2') or plt.setp(first, label='Y=X\*\*2')

> second.set\_label('Y=X\*\*3')

> legendObj = plt.legend()

* 1. **Getting a reference for any object on the plot:**
     1. Lets say we made a mistake on one of the plot objects… say added text that we didn’t want… and forgot to store the reference to it… How would we get access back so we could manipulate its properties (in the current context, remove it from the plot)

> plt.text(1, 70, "Playing with the Identity Function", fontsize=30)

* + 1. Use plt.gca() to get the current axes that you are plotting stuff on; then use the get\_children() method. This gives the list of all the objects that are children of this particular axes object… You can extract the relevant reference of the object to be manipulated from this list and then do whatever you wish to with it.

> axes = plt.gca()

> children = axes.get\_children()

* 1. **Annotation of a particular element on the plot:**

> plt.annotate?

> annotation = plt.annotate('Point x=2, y=8', (2,8))

## Doesn’t give an arrow; annotation text is right on the point

> annotation.remove(); plt.draw()

> annotation = plt.annotate('Point x=2, y=8', (2,8), xytext=(1.5, 20), arrowprops=dict(facecolor='black', shrink=0.1))

1. We have seen at various objects, but there are **many more types of objects**… Figures, which contain axes, which contain all other elements… Ticks, Tick labels, random shapes etc… Plus various functions to manipulate them all.
2. **We have only seen the very flexible plot function**. But there are many other specialized functions available to make specific kinds of plots, for ex. plt.hist() for plotting histograms, plt.boxplot() for making boxplots, plt.scatter() for making scatter plots…

> plt.<tab> ## to see the entire list of functionality available…

> plt.<function>? ## to see documentation of the <function>

All these functions also provide various keyword arguments/parameters to set the properties of the various objects. These are typically complex objects. For example, a histogram is a collection of rectangles… For some of the properties, it is possible to collectively set them, like the color of the histogram… But it is also possible to get more granular access to the all the rectangle objects and other sub-objects that make up the entire histogram… Then you can configure their properties individually…

1. **Multiple ways to control properties of various objects**… at the time of creation give parameters… or set them post-creation…
   1. Since defaults are pretty good, so I would suggest using that to start with, and then interactively update your plots till they suit your taste… For that keep references of all objects in your plots as separate variables, and then use those to update properties, either directly using the object.set/get functions or plt.setp/getp functions
   2. Develop your own style… matplotlib is very flexible to accommodate various styles…
2. **Plot Navigation Buttons**
3. [**http://matplotlib.org/examples**](http://matplotlib.org/examples)

**## Making expression histograms for top-ranking differentially expressed genes in our dataset (introduced in Pandas-II hands-on)**

# Copy code up to but excluding the for loop used for plotting …

# Use **%paste** to paste it in ipython window (to retain proper formatting)

# All the modules/functions that we need are now available (matplotlib.pyplot, np.arange etc.)

> plt.interactive(True)

# Pull out top most gene’s expression values

> geneExpr = exprDF.ix[:, tTest\_pValues\_sortedAsc.index[0]]

# Extract separate sample values for each sample

> groups = groupLabels.unique()

> sample1 = geneExpr[groupLabels==groups[0]].tolist()

> sample2 = geneExpr[groupLabels==groups[1]].tolist()

# Now make plots

> plt.figure(1)

> plt.subplot(2, 2, 1, title = tTest\_pValues\_sortedAsc.index[0])

**## Note:** 2,2,1 => 1st plot of the 2\*2 matrix of plots

> plt.hist(sample1, normed=True, edgecolor='r', fill=False, label=groups[0])

**## Explain the keyword arguments**

> plt.hist(sample2, normed=True, edgecolor='b', fill=False, label=groups[1])

**## label is for the legend that we'll add later**

## Get the limit of the x-axis

> xlim = plt.xlim()

> xpoints = arange(xlim[0], xlim[1], step=0.01)

## Gaussian Curve

> mean1 = mean(sample1)

> std1 = std(sample1)

> density1 = stats.distributions.norm.pdf(xpoints, loc=mean1, scale=std1)

> plt.plot(xpoints, density1, color='r')

> mean2 = mean(sample2)

> std2 = std(sample2)

> density2 = stats.distributions.norm.pdf(xpoints, loc=mean2, scale=std2)

> plt.plot(xpoints, density2, color='b')

> plt.legend()

## Now repeat the process for another gene...

## plt.subplot(2,2,2, title=tTest\_pValues\_sortedAsc.index[1])

**## Note:** 2,2,2 => 2nd plot of the 2\*2 matrix of plots

## etc...

## This process is tedious for making many plots...

## Plus there is a lot of repetition of code... We'll have to repeat all these steps to make more plots... only the gene changes

## To make it cleaner/shorter and to make our lives easier, we can use for loops and functions...