Python for X-Informatics

(Introduction)

Python Packages and Tools

- Canopy IDE and IPython
- NumPy
- Matplotlib
- SciPy

Virtual Box Appliance

- Virtual Box Appliance with all packages set-up
- Tutorial to use the virtual box appliance
- *You can manually setup the necessary stuff or use the virtual box appliance*

Python for X-Informatics

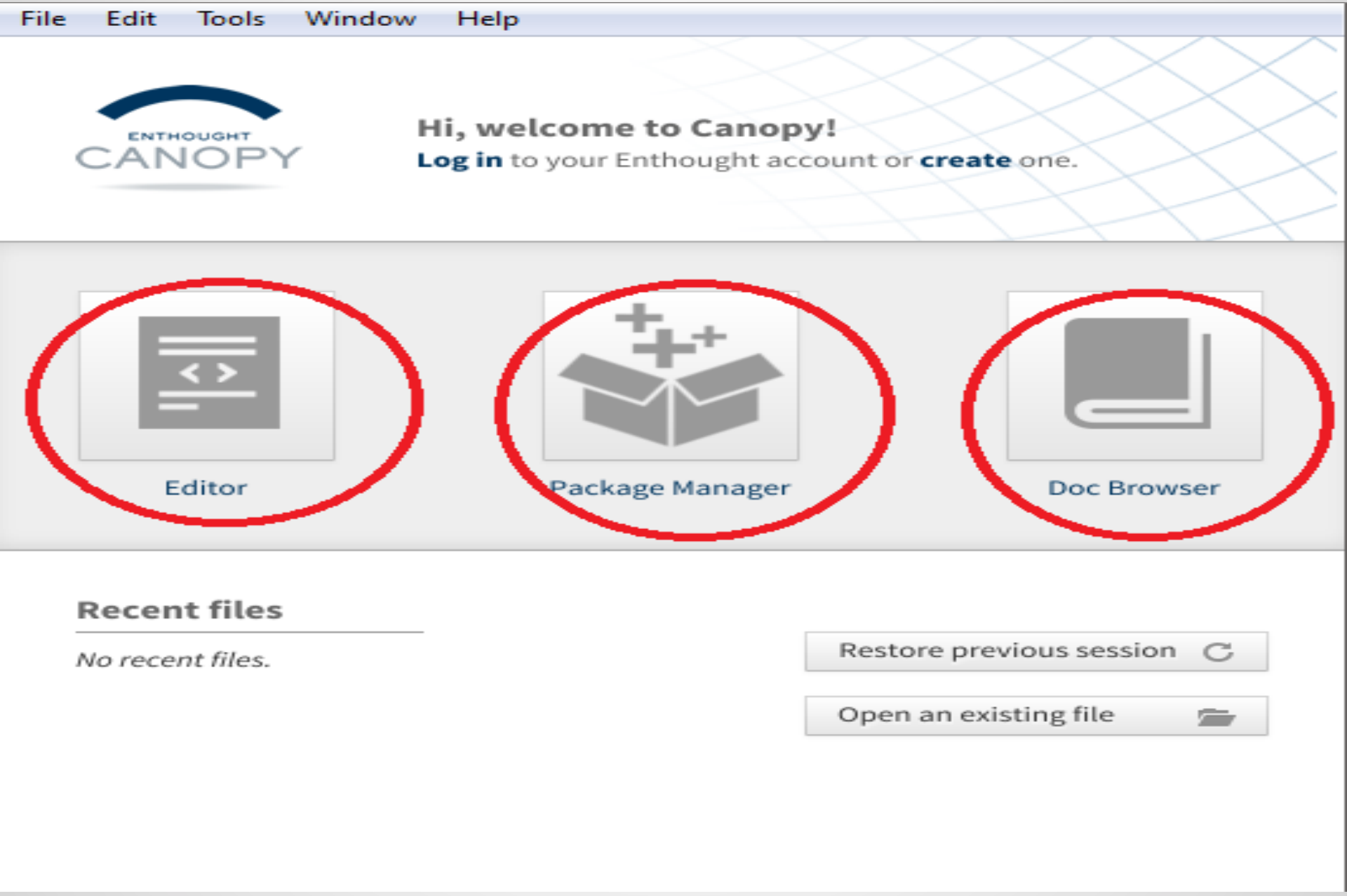
(Introduction to Canopy)

Download Canopy

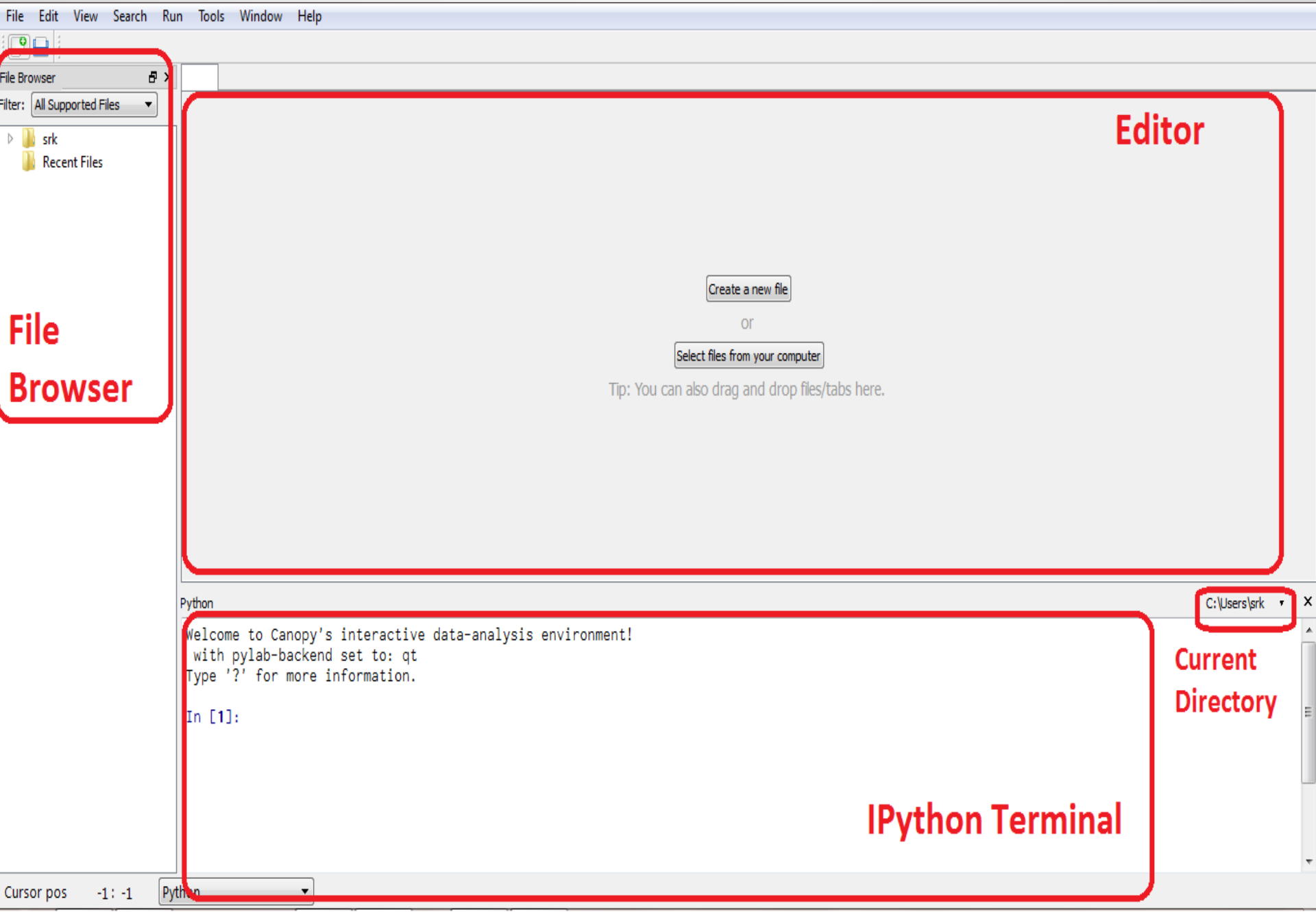
- https://www.enthought.com/store/

- Versions
  - Express
  - Basic
  - Professional

Welcome Screen

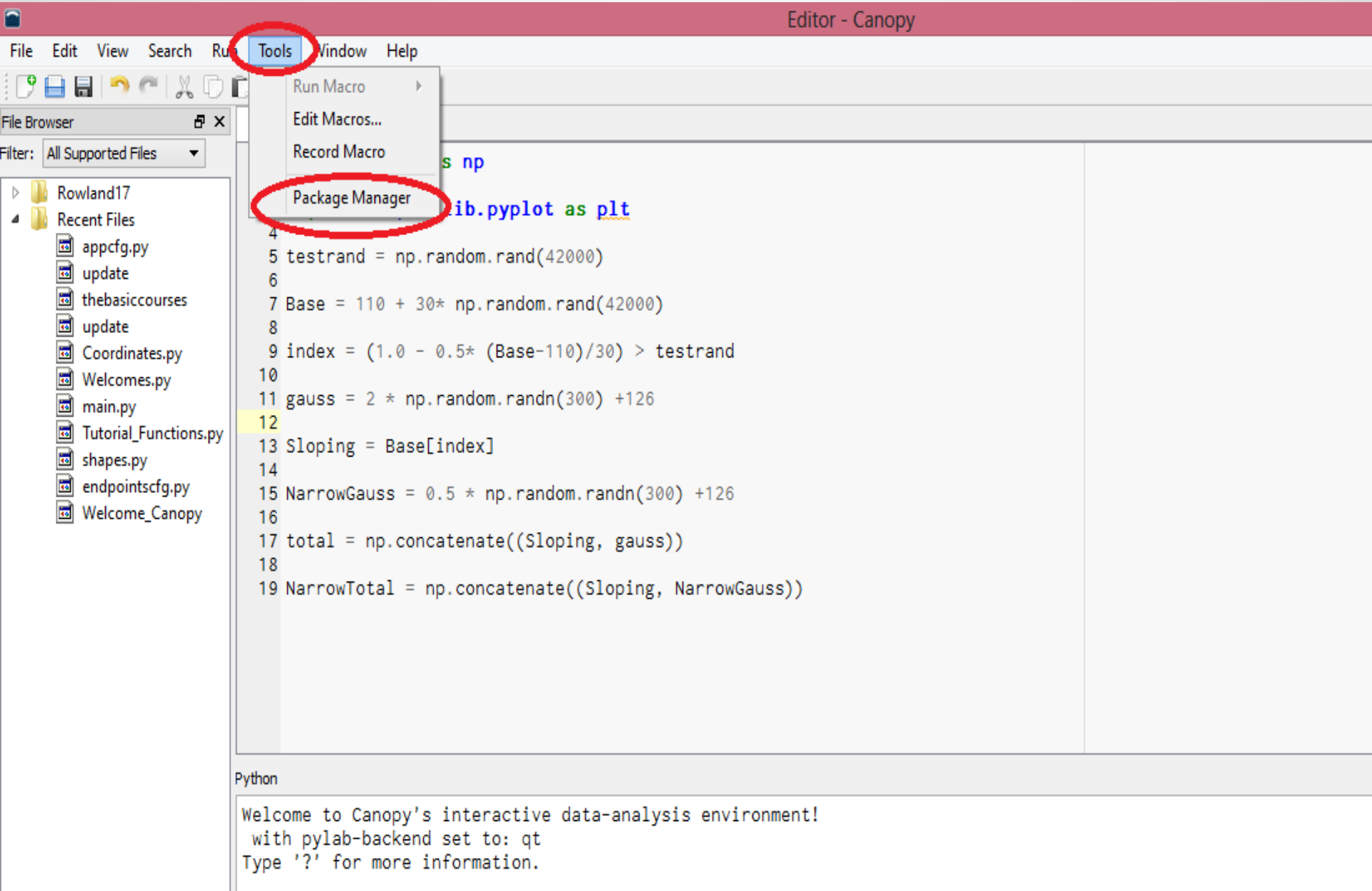


Editor

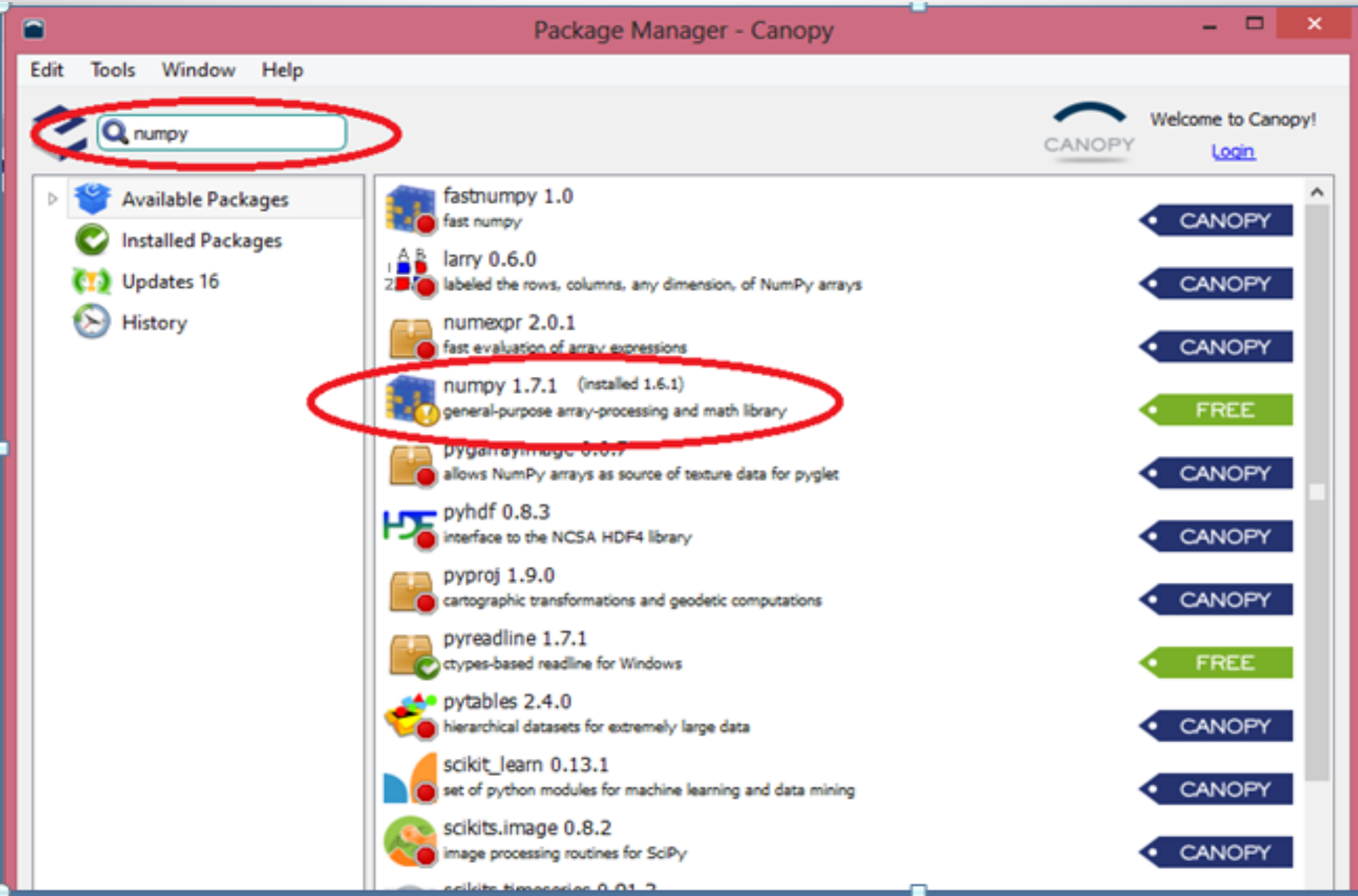

File Edit View Search Run Tools Window Help
File Browser
Filter: All Supported Files
srk
Recent Files
File Browser
Editor
Create a new file
or
Select files from your computer
Tip: You can also drag and drop files/tabs here.
Python
C:\Users\srk
Welcome to Canopy's interactive data-analysis environment!
with pylab-backend set to: qt
Type '?' for more information.
In [1]:
Current Directory
IPython Terminal
Cursor pos -1 : -1
Python

Getting Necessary Packages

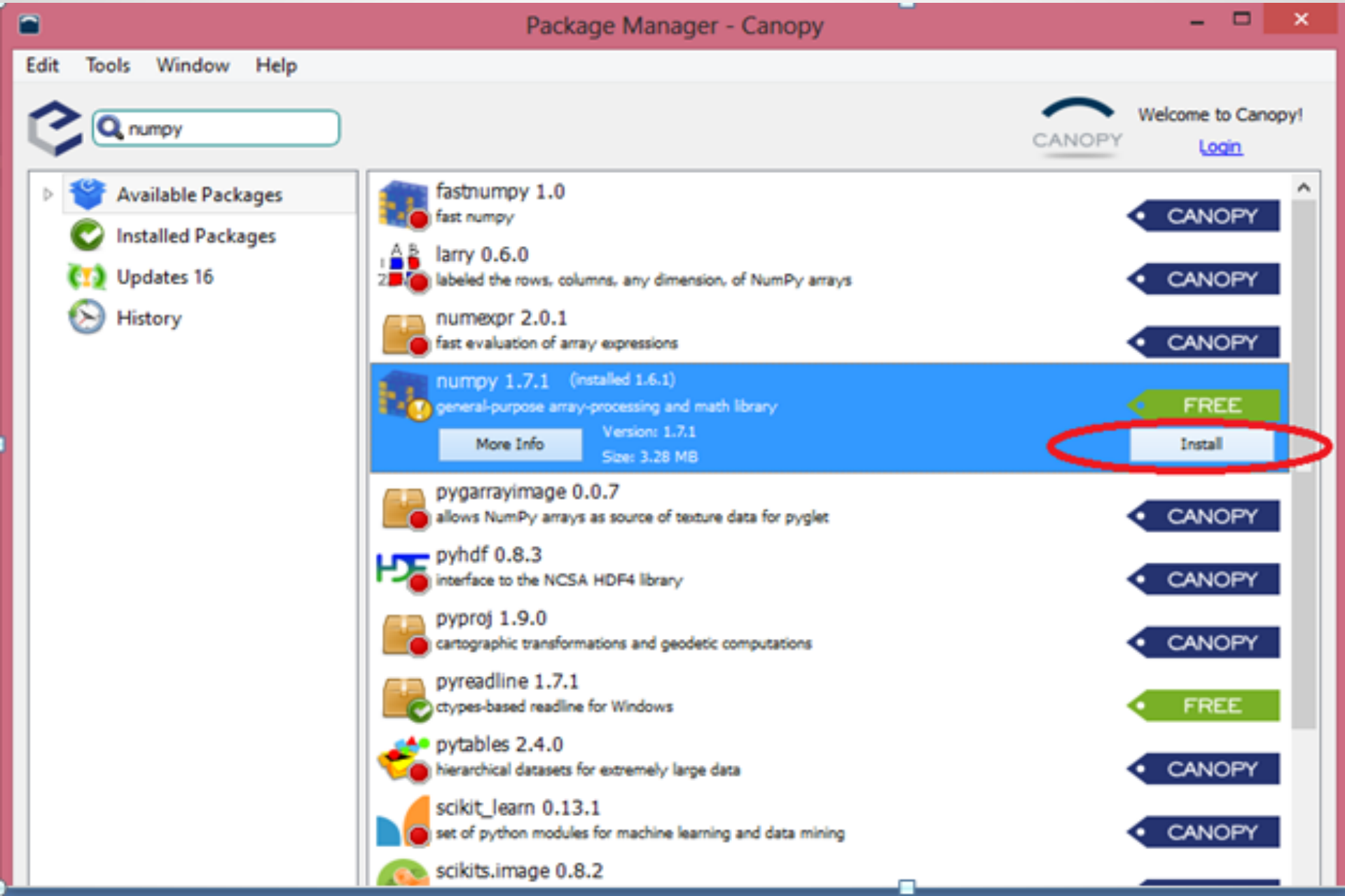
Click on Tools, then Package Manager



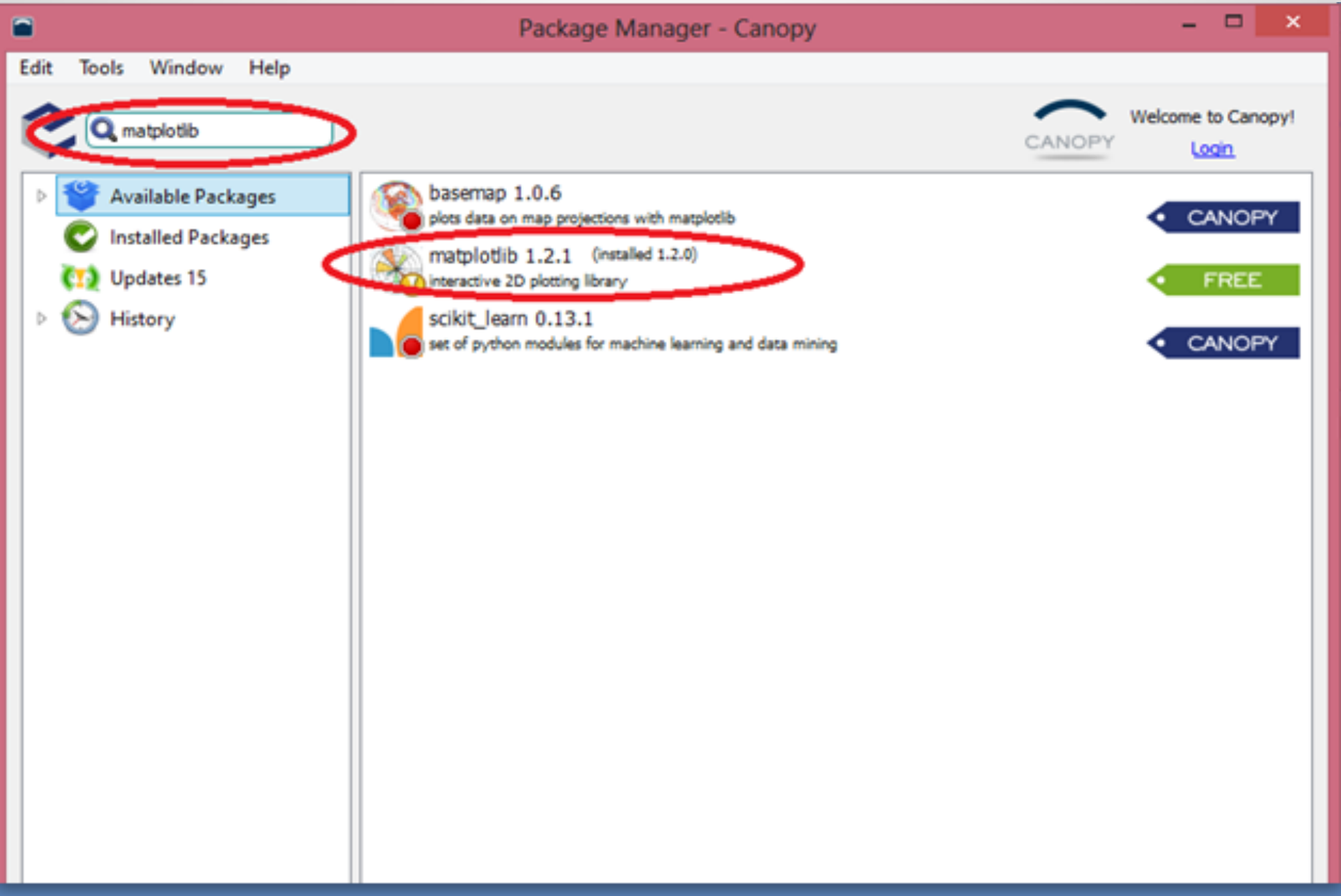
Search for NumPy



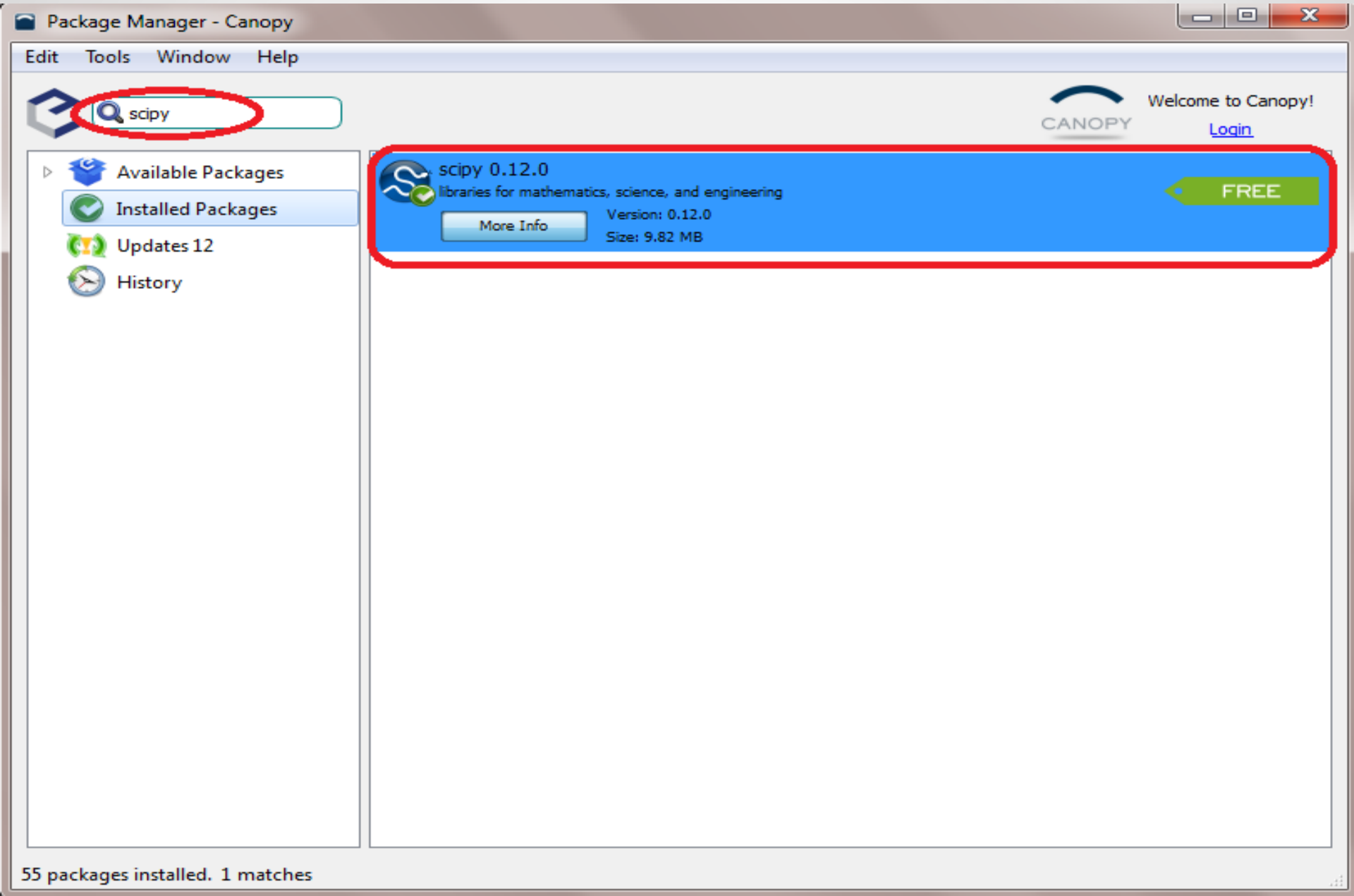
Then Click on the "install" button:



Simlarly Get Matplotlib



Simlarly Get Scipy



Python for X-Informatics – NumPy I

Overview

NumPy is a python package which can be used to represent data in a vectorized format. A number of other packages are built on top of numpy. These packages use the data structures and operations used in numpy.The express version of canopy (the one recommended for this class) comes with its own version of NumPy.

Download(If you are not using Canopy)

If you do not wish to use the canopy IDE numpy can be downloaded from - https://pypi.python.org/pypi/numpy. But for the purpose of the class we will be using the version of NumPy provided with Canopy.

Importing Numpy Package

```
import numpy as np
```

Importing numpy as np is a standard convention which will be used in the unit and the course. If this runs without any errors which indicates that numpy is present.

Introduction to Arrays using Numpy

N Dimensional Array(ndArray)

ndArray or an n-dimensional array is the most important data structure used in numpy. It is the data structure which gives all of the vectorization power to numpy.

Creating 1 dimensional ndArray

The **np.array()** method is used to create ndarrays. To it we pass a a sequence(like a list) of data elements.

```
s = np.array([1,2,3,4])
s
Out[100]: array([1, 2, 3, 4])
```

Creating 2 dimensional ndArray

For creating a 2-d array we need to pass a sequence of sequences (or a list of lists) where each sequence denotes a row.

```
import numpy as np
s = np.array([[2,3,4],[2,3,5],[1,4,5]])
s
```

Out[103]:
```
array([[2, 3, 4],
       [2, 3, 5],
       [1, 4, 5]])
```

We can extend this to create arrays of higher dimensions. Also it is essential that the number of elements in each of these inner lists be equal for it to be converted to a higher dimensional array.

Important Attributes and Methods of ndarray

Dimensionality of Array (ndim)

The ndim attribute of the ndarray can be used to find the dimensionality. The array s we created was a 2 dimensional array.

```
s.ndim
```
Out[104]: 2

Shape of Array(shape):

The shape attribute can be used to find the number of elements of the ndarray. s we created was a 3x3 array.

```
s.shape
```
Out[105]: (3, 3)

Size of the Array (size):

The size property tells the size of the array. The size is essentially the product of elements of the shape. Since our array s was a 3x3 array, the size is 9

```
s.size
```
Out[106]: 9

Data Type of Array Elements (dtype)

The dtype property stores the data type of the array elements. Our array was an int array and it used int32 datatype. A list of all data types can be found in the documentation - http://docs.scipy.org/doc/numpy/user/basics.types.html

```
s.dtype
```
Out[107]: dtype('int32')

Reshaping the array (reshape())

The reshape() method can be used to change the shape of the array. You can pass a tuple which indicates the new shape array and an array with the new shape is returned. The shape of the ***original array is however unaltered***. Here we change the reshape s as a 9x1 array and store it in d. However s is unaltered

```
d = s.reshape((9,1))
d
Out[109]:
array([[2],
       [3],
       [4],
       [2],
       [3],
       [5],
       [1],
       [4],
       [5]])
s
Out[110]:
array([[2, 3, 4],
       [2, 3, 5],
       [1, 4, 5]])
```

Few other methods of creating arrays

np.zeros():

To create an array with all elements set to zero use np.zeros() method. For higher dimensions pass the dimensions as a tuple.

```
np.zeros(5)
Out[111]: array([ 0.,  0.,  0.,  0.,  0.])
np.zeros((2,2))
Out[112]:
array([[ 0.,  0.],
       [ 0.,  0.]])
```

np.ones():

This method works similar to the np.zeros() method except that the array has all elements set to 1.

```
np.ones((2,2))
Out[113]:
array([[ 1.,  1.],
```

```
       [ 1.,  1.]])
```

np.arange()

It takes the input start, end, stepsize and creates an array accordingly. And as it always in python the ***end is not included***. So it starts from start and increments by the step size and stops when value becomes larger than or equal to end.

```
np.arange(1,11,2)
Out[114]: array([1, 3, 5, 7, 9])
```

If the step size is not specified a default step size of 1 is taken.

```
np.arange(1,11)
Out[116]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

If the start is not specified a default value of 0 is taken.

```
np.arange(11)
Out[118]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

linspace()

You can specify the start, end and the number of elements needed. It create an array with first element as start, last element as end***(here end is included),*** and the array has a total number of elements as specified.

```
np.linspace(1,10,10)
Out[119]: array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

The default value for the number of elements is 50.

```
np.linspace(1,10)
Out[120]:
array([ 1.        ,  1.18367347,  1.36734694,  1.55102041,
        1.73469388,  1.91836735,  2.10204082,  2.28571429,
        2.46938776,  2.65306122,  2.83673469,  3.02040816,
        3.20408163,  3.3877551 ,  3.57142857,  3.75510204,
        3.93877551,  4.12244898,  4.30612245,  4.48979592,
        4.67346939,  4.85714286,  5.04081633,  5.2244898 ,
        5.40816327,  5.59183673,  5.7755102 ,  5.95918367,
        6.14285714,  6.32653061,  6.51020408,  6.69387755,
        6.87755102,  7.06122449,  7.24489796,  7.42857143,
        7.6122449 ,  7.79591837,  7.97959184,  8.16326531,
        8.34693878,  8.53061224,  8.71428571,  8.89795918,
```

```
       9.08163265,  9.26530612,  9.44897959,  9.63265306,
       9.81632653,  10.        ])
```

Arithmetic operations(between 2 arrays)

When we perform an operation using +,-,*,/,%,**(power or exponent) between two arrays of the same shape, it results in an element-wise operation between the two arrays.

```
a = np.array([[2,3],[3,4]])
b = np.array([[1,2],[2,3]])
```

Addition

```
a+b
Out[123]:
array([[3, 5],
       [5, 7]])
```

Subtraction

```
a-b
Out[124]:
array([[1, 1],
       [1, 1]])
```

Product

```
a*b
Out[125]:
array([[ 2,  6],
       [ 6, 12]])
```

This is different from a dot product which can be obtained using the **dot()** method

```
a.dot(b)
Out[127]:
array([[ 8, 13],
       [11, 18]])
```

Division

```
a/b
Out[126]:
array([[2, 1],
       [1, 1]])
```

For division as in python, ***floating point division is performed only if atleast one array has a floating data type***

Arithmetic Operations with scalars

When a scalar is added(true for any other arithmetic operation), the scalar is added to each element in the array.

Addition

```
a + 2
Out[128]:
array([[4, 5],
       [5, 6]])
```

Subtraction

```
a - 2
Out[129]:
array([[0, 1],
       [1, 2]])
```

Product

```
a * 2
Out[130]:
array([[4, 6],
       [6, 8]])
```

Division

Here as all elements/scalars involved are int values the result is an int.

```
a / 2
Out[132]:
array([[1, 1],
       [1, 2]])
```

Here as 2.0 is float, we get float precision.

```
a / 2.0
Out[133]:
array([[ 1. ,  1.5],
       [ 1.5,  2. ]])
```

Exponent:

Here the "**" operator is used to indicate that an array a is raised in an element-wise fashion by a power of 2

```
a ** 2
Out[31]:
array([[ 4,  9],
       [ 9, 16]])
```

Python for X-Informatics – Numpy II

Basic Indexing

Indexing in One Dimensional Arrays

The indexing operation is used to get part of the array.
For one dimensional arrays, we can get an element like we do in a python list by using ***square brackets***.

```
b = np.arange(10)
print b
Out[111][0 1 2 3 4 5 6 7 8 9]
b[0]
Out[112]: 0
```

The indexing starts with 0 and the last index is "n-1" where "n" is the total number of elements. So in our case index starts from 0 and the last index is 9

```
b[9]
Out[113]: 9
```

Negative integers can also be used for index values

```
b[-1]
Out[114]: 9
b[-2]
Out[115]: 8
```

We can get the ***whole array*** by using "**:**" as the index

```
b[:]
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

We can get a ***range of elements*** as follows by having two index values separated by "**:**"

```
b[0:5]
Out[139]: array([0, 1, 2, 3, 4])
b[1:5]
Out[140]: array([1, 2, 3, 4])
```

By omitting the value before the ":", that value is set to a default of 0.

```
b[:5]
Out[141]: array([0, 1, 2, 3, 4])
```

By omitting the value after the ":" all elements till the last element are returned

```
b[5:]
Out[142]: array([5, 6, 7, 8, 9])
```

You can ***pass arrays as indices***. Then all elements which have indices equal to values in the array are returned.

```
b[[0,2,4]]
Out[144]: array([0, 2, 4])
```

Indexing For Multidimensional arrays

For multi-dimensional arrays we use one index per axis with indices separated by commas. For a 2 dimensional array, first value indicates the row number and second indicates column number.

```
b[0,0]
Out[147]: 0
```

Using the powerful methods to get range of elements – Multi Dimensional Arrays

Any of the techniques used to get elements for a 1 dimensional array. You can specify one index per axis and all the techniques for getting all the elements or a range of elements can be extended

```
b[0,:]
Out[148]: array([0, 1, 2, 3])
b[:,0]
Out[149]: array([ 0,  4,  8, 12])
b[0,2:]
Out[150]: array([2, 3])
```

If you pass an ***arrays as indices*** as in the example below you would expect a rectangular region containing all elements from the rows with indices 0 and 3 and cols with indices 1 and 2. However this method returns elements (0,1) and (3,2).

```
b[[0,1],[0,1]]
Out[151]: array([0, 5])
```

To get the rectangular result you expected use ***np.ix_()*** method

```
b[np.ix_([0,1],[0,1])]
Out[152]:
array([[0, 1],
       [4, 5]])
```

Relational Operations

Relational operators (like >, <, ==, !=, <=,>=) can be used to perform an element wise relational comparison. The return value of this relational operation is a boolean array which has the value True for all the elements that satisfy the condition and false for the others.

```
a = np.arange(4)
```

Greater Than(>)

```
a > 2
Out[154]: array([False, False, False,  True], dtype=bool)
```

Equal To(==)

```
a == 2
Out[155]: array([False, False,  True, False], dtype=bool)
```

Not Equal to (!=)

```
a != 2
Out[156]: array([ True,  True, False,  True], dtype=bool)
```

Less than(<)

```
a < 2
Out[157]: array([ True,  True, False, False], dtype=bool)
```

And(&) / Or(|) Operations

If you need to chain multiple comparisons, you should chain it using "|" for or and "&" for and. Also the conditions should be in brackets. Do not use the and/or operators in Python.

```
(a==2)|(a==3)
Out[158]: array([False, False,  True,  True], dtype=bool)
(a==2)&(a==3)
Out[159]: array([False, False, False, False], dtype=bool)
```

Boolean Indexing

You can pass boolean arrays as index parameters to arrays.

```
b = np.arange(4)
c = b>1
 c
Out[163]: array([False, False,  True,  True], dtype=bool)
b[c]
Out[164]: array([2, 3])
a = np.array([0,1,4,5])
a[c]
Out[165]: array([4, 5])
a[b>1]
Out[166]: array([4, 5])
```

Changing values using Boolean Indexing

You can change values by passing boolean indices. ***This is true for any kind of indexing*** but with boolean indexing it becomes hard to understand and it is really powerful.

```
a[b>1] = 100
a
```

Out[168]: array([0,   1, 100, 100])

Python for X-Informatics – Numpy III

Transposing and swapping axes

The ***"T" attribute*** of the array can be used to obtain the transpose of the array.

```
a = np.arange(4).reshape((2,2))
a
Out[162]:
array([[0, 1],
       [2, 3]])
a.T
Out[163]:
array([[0, 2],
       [1, 3]])
```

The ***transpose()*** method can also be used. The new axis order can be passed to transpose(). This comes in really handy for higher dimensional arrays when you want to change the order of the axes.

```
a.transpose()
Out[164]:
array([[0, 2],
       [1, 3]])
a.transpose(0,1)
Out[165]:
array([[0, 1],
       [2, 3]])
a.transpose(1,0)
Out[166]:
array([[0, 2],
       [1, 3]])
```

Universal Functions (ufuncs)

Unary ufuncs

These functions perform element wise operations on the arrays. Some of them are unary functions are sqrt, sin, cos, tan, log, etc.

```
a = np.arange(4)
a
Out[183]: array([0, 1, 2, 3])
np.sqrt(a)
Out[184]: array([ 0.        ,  1.        ,  1.41421356,  1.73205081])
```

Binary ufuncs

Some of them are binary i.e they take two arguments – for example add, subtract, divide, maximum, minimum, etc.

```
a = np.arange(4)
b = np.array([4,3,2,1])
a
Out[177]: array([0, 1, 2, 3])
b
Out[178]: array([4, 3, 2, 1])
np.maximum(a,b)
Out[179]: array([4, 3, 2, 3])
```

A list of all the possible universal functions can be found in the documentation - http://docs.scipy.org/doc/numpy/reference/ufuncs.html

Computing array Statistics

These are functions which can compute statistics of the entire array like sum, max, min, mean, variance, etc.

```
a
Out[181]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
a.sum()
Out[182]: 120
```

Computing statistics along the axes

When no parameter is provided these methods compute statistics for the whole array. However it is possible to pass axis numbers to the method to compute these statistics on an axis level. Passing 0 the sum(or any other statistic) can be computed along the row axis that is one sum per column and when you pass 1 you can compute sum along the column axis.

```
a.sum(0)
Out[183]: array([24, 28, 32, 36])
a.sum(1)
Out[184]: array([ 6, 22, 38, 54])
```

A list of all the array statistics methods can be found in the documentation - http://docs.scipy.org/doc/numpy/reference/routines.statistics.html

Linear Algebra

The linear algebra methods like inverse, decompositions, etc are found in the numpy.linalg package.

```
b = np.arange(4).reshape((2,2))
np.linalg.inv(b)
Out[258]:
array([[-1.5,  0.5],
       [ 1. ,  0. ]])
```

A list of linear algebra methods can be found in the documentation -
http://docs.scipy.org/doc/numpy/reference/routines.linalg.html

Random Number Generation

The **numpy.random** module has the necessary tools for generating random numbers. It also facilitates the use of distributions for drawing out random numbers.

Random numbers from a uniform distribution

The rand() used for drawing random numbers from uniform distribution between 0 and 1.

```
np.random.rand(2,2)
Out[185]:
array([[ 0.3164555 ,  0.22305861],
    [ 0.31224002,  0.83036833]])
```

Random Numbers from a Normal Distribution

The randn() method is used to get random numbers from a gaussian distribution with mean = 0 and standard deviation = 1.

```
np.random.randn(2,2)
Out[186]:
array([[-0.26117987, -0.9391148 ],
    [ 0.67581885, -0.17256601]])
```

The complete list of methods for random number generation can be found in the documentation -
http://docs.scipy.org/doc/numpy/reference/routines.random.html

Python For X-Informatics – Matplotlib I

Overview:
Matplotlib is a popular plotting package and is used for creating graphs and plots for data visualization. It can be used with ipython is an interactive manner and proves to be very useful. It has interactive features like zooming, panning. It also supports all the popular graphic formats like JPEG, GIF, TIFF, etc and can be used to plot, modify, images. If you are familiar with MATLAB, you can do almost all kinds of plotting possible with MATLAB using matplotlib.

Importing packages

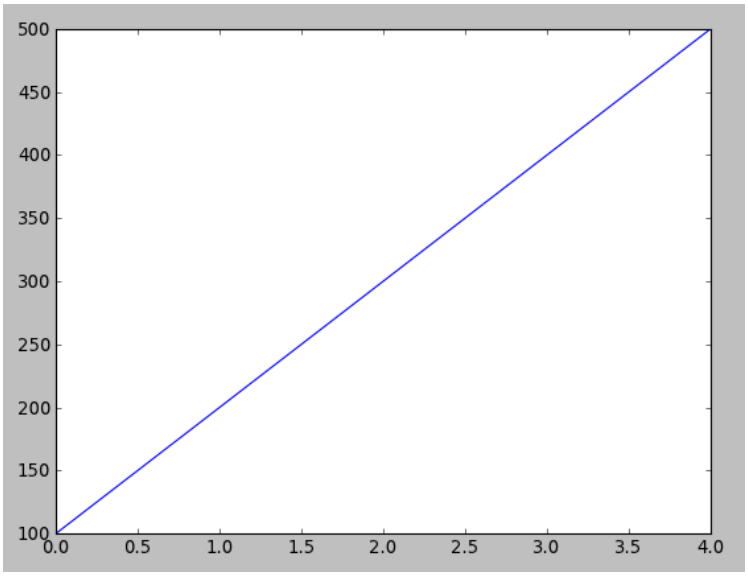
The matplotlib.pyplot package is one of the most important packages and provides a collection of methods that enable us to create graphs, figures, plots and modify them. The data science community generally imports matplotlib.pyplot as plt. For all further discussions plt would refer to matplotlib.pyplot. It allows you to make a lot of different kind of plots. The different kinds of plots you can use is listed in the documentation (http://matplotlib.org/api/pyplot_api.html).

Apart from matplotlib, since we will be using numpy in this tutorial, you will also have to import that.

```
import matplotlib.pyplot as plt
import numpy as np
```

Plot Method

```
plt.figure()
plt.plot(np.array([100,200,300,400,500]))
```

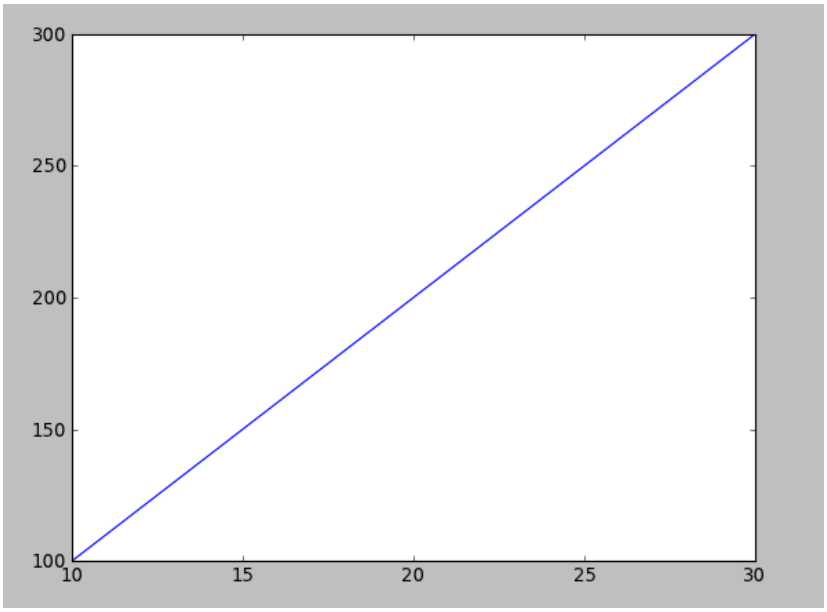


So here we plot the an array using the plt.plot() method. When only one array is passed to the method it is considered to be the Y-values and the X-values are self-generated. So you can see that the Y values go from 100 to 500 as in the array. However the X values start from 0 and go up to 4.

Plotting on Both Axes

We can pass X and Y values as separate parameters(X first) to the plot function.

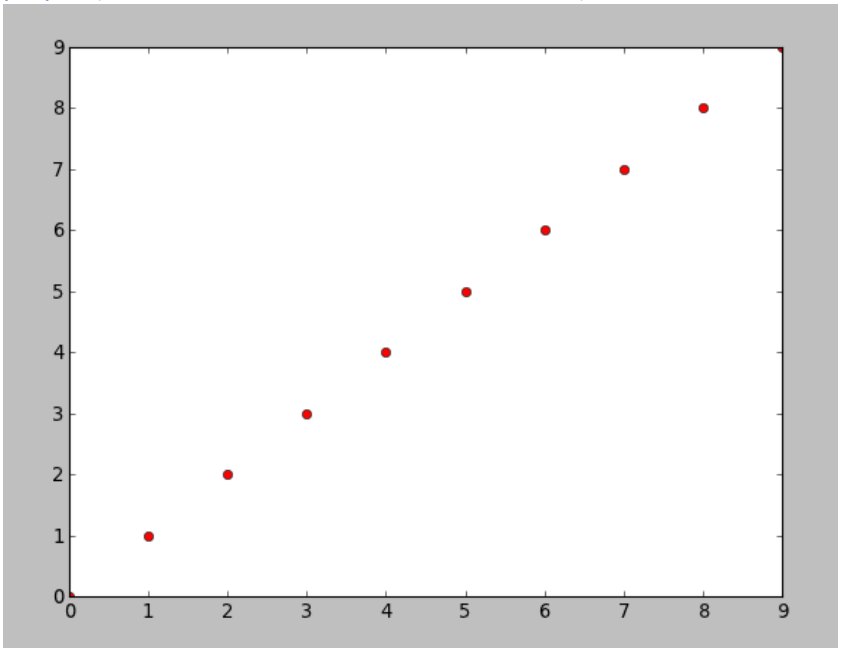
```
plt.figure()
plt.plot(np.array([10,20,30]), np.array([100,200,300]))
```



Formatting The Plots

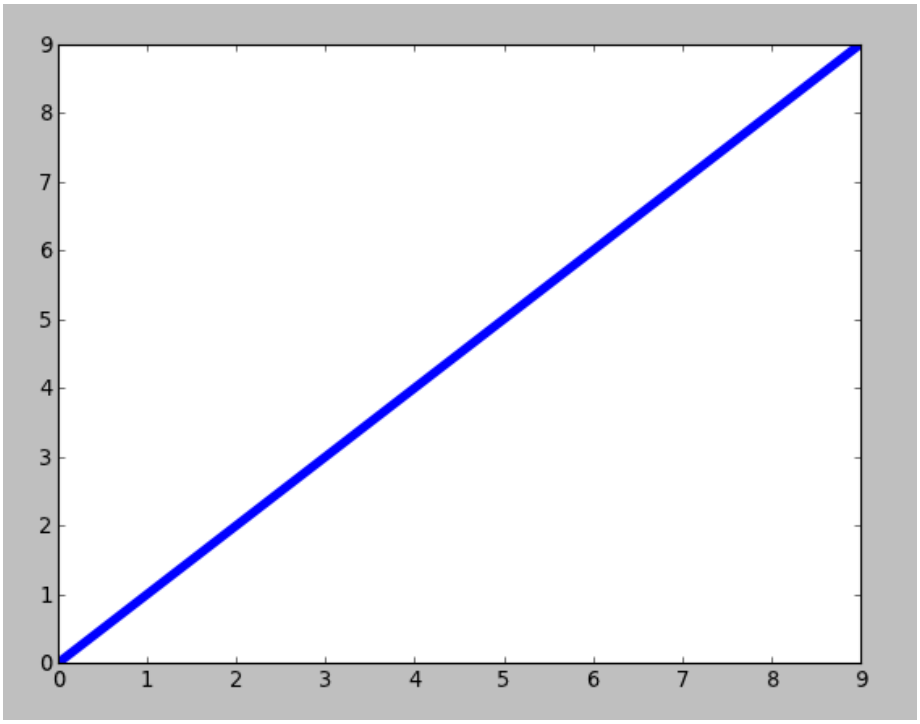
You can pass formatting parameters to the method. The default parameter is “b-“ where b indicates blue and – indicates line. You can change the color by changing the first character and the symbol by changing the second character

plt.plot([0,2,4,6,8,10,12],[1,3,5,7,9,11,13],"ro")



You can pass ***other parameters*** while creating a plot.

```
plt.figure()
plt.plot(np.arange(10),np.arange(10),linewidth=5.0)
```



List of the formatting and other parameters can be found in the pyplot documentation under the plot() method (http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot).

Axis, Labels, Title, Grids

Adding label to X-axis

```
plt.xlabel("x axis")
```

Adding label to Y-axis

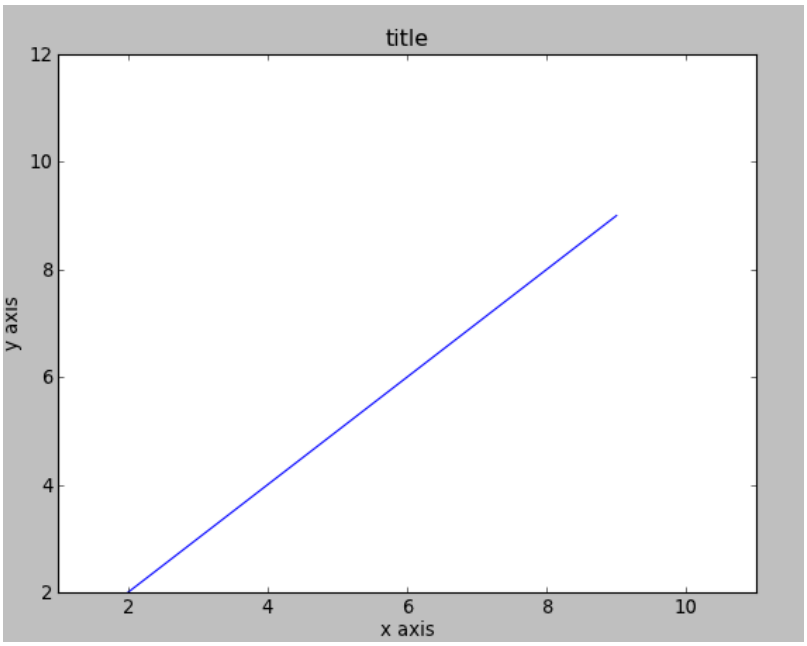
```
plt.ylabel("y axis")
```

Adding the title to the figure

```
plt.title("title")
```

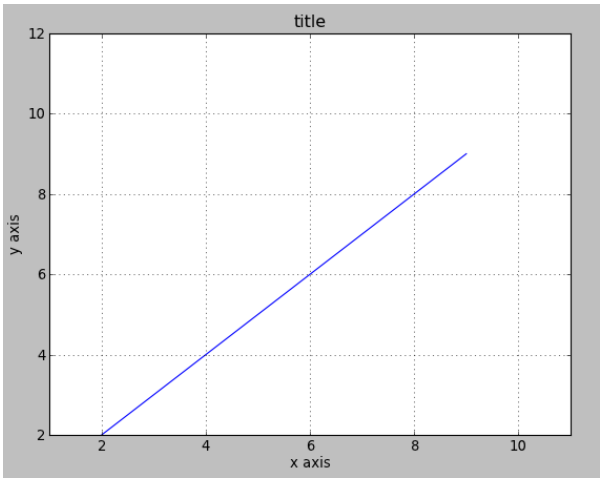
Changing the start and end of axis - plt.axis(x_min, x_max, y_min, y_max)

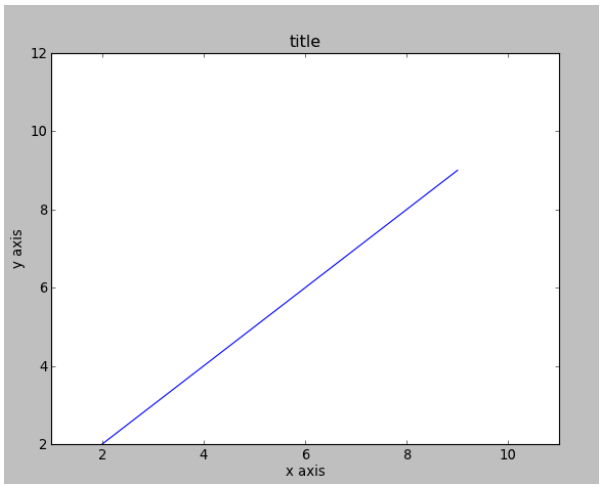
```
plt.axis([1,11,2,12])
```



Toggling Grid lines

```
plt.grid(True)
plt.grid(False)
```





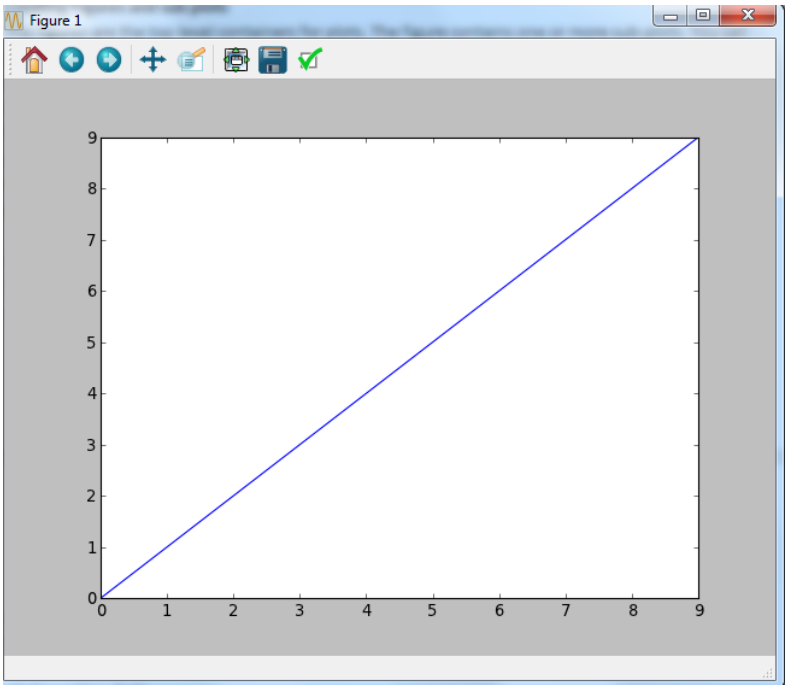
Python For X-Informatics – Matplotlib II

Creating Figures and sub plots

Figures

The figures are the top level containers for plots. The figure contains one or more sub-plots. You can use the plt.figure() method to create figures. You can pass the name of the figure you want to create as argument. If you don not pass anything it runs an auto-incrementing count. The name displayed if the word “Figure” followed by that number. Each figure has a unique key which is either given by the user or system generated. Now if you call the plt.figure() method and pass the figure key which is already present, it sets that figure as the current figure. All plots are made to that figure. Here in the example we create 2 figures which will be assigned numbers 1 and 2. Now when we call figure() and pass one to it a new figure is not created instaed the plots are made on the figure 1 created before.

```
plt.figure()
plt.figure()
plt.figure(1)
plt.plot(np.arange(10))
```



Creating figures with Name

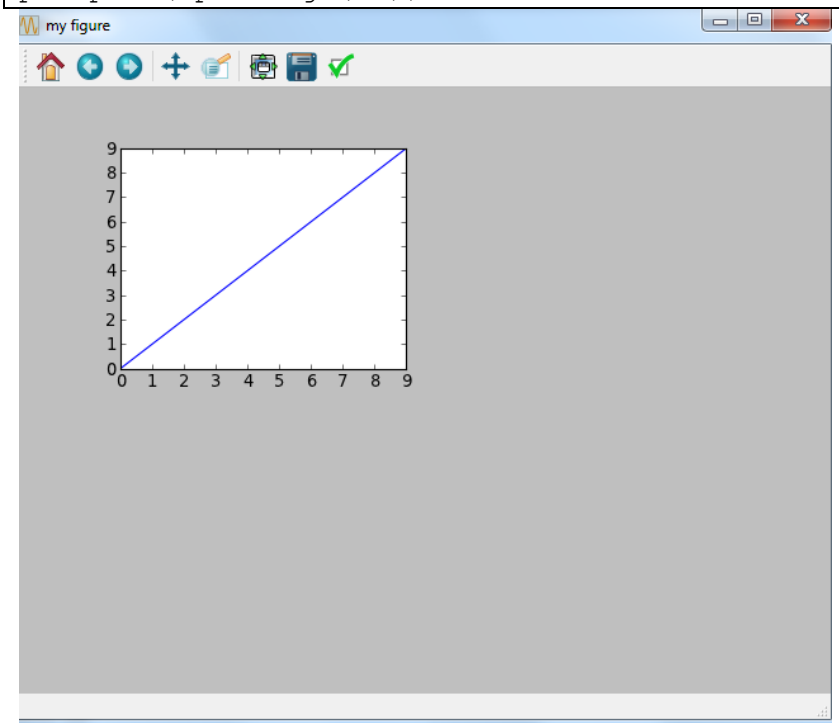
Pass the desired name to the figure method.

```
plt.figure("my figure")
```

Subplots

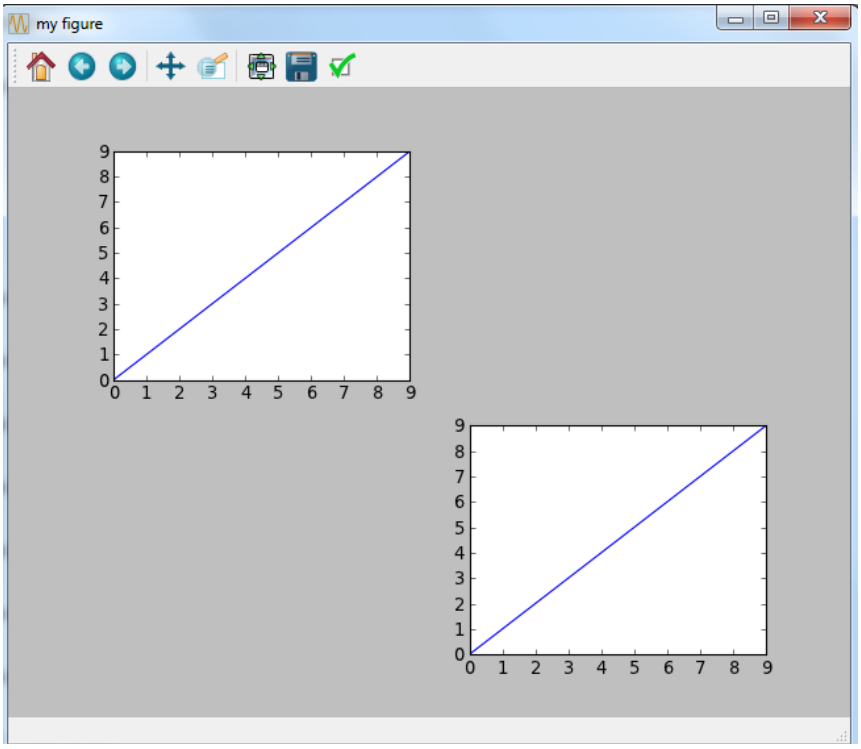
You can create **multiple plots** in a figure by using the plt.subplot() command. Here we create subplots in the figure "my figure" created before. The first two arguments indicate the number of vertical and horizontal plots you intend to create. The third argument indicates the subplot number you want to select. Like the figure() this will use the last called sub plot of the figure for plotting. Also if no subplot is mentioned for the figure, a single sub-plot is created and used.

```
plt.subplot(2,2,1)
plt.plot(np.arange(10))
```



Also you can specify the subplot index without the commas.

```
plt.subplot(224)
plt.plot(np.arange(10))
```



Interactive mode

The interactive mode allows you to interact with the plots. You can see the changes on the go. You may want to turn it off and plot all the figures at the end. So for this purpose you can switch off the interactive mode. The kind of interaction you want depends on what kind of task you are trying to achieve

Checking if interactive mode is on - plt.isinteractive():

The plt. Isinteractive() method can be used to check if the interactive mode is on or no. It returns true if the interactive mode is on False otherwise.

```
plt.isinteractive()
Out[72]: True
```

Turning off interactive mode:

plt.ioff() method can be used to turn off the interactive mode. Use the plt.show() method to show all the figures in this mode and the plt.draw() method to redraw the figures.

```
plt.ioff()
plt.isinteractive()
Out[74]: False
plt.figure()
plt.plot(arange(100))
plt.show()
```

```
plt.plot(arange(10000))
plt.draw()
```

Turn the interactive mode on:

The plt.ion() method can be used to turn on the interactive mode.

```
plt.ion()
plt.isinteractive()
Out[81]: True
```

Python for X-Informatics – SciPy I

Overview

Scipy is a powerful python package built on top of numpy. It has a number of algorithms pertaining to different fields like calculus, stats, signal processing, image processing, etc.

List of Scipy Packages

The documentation to the all the packages in scipy can be found online at
http://docs.scipy.org/doc/scipy/reference/

Clustering package (scipy.cluster)

This package provides the methods to perform clustering using the k-means algorithm.

Constants (scipy.constants)

This package provides all of the mathematical constants (eg: Pi(π))

Discrete Fourier transforms (scipy.fftpack)

Provides methods to perform fourier transforms. These are popularly used in the fields of Image Processing and Signal Processing.

Integration and ODEs (scipy.integrate)

This package provides methods for performing integration.

Interpolation (scipy.interpolate)

This package provides classes and methods to perform interpolation. Interpolation is finding values(missing) between a known range of values.

Input and output (scipy.io)

Provides Methods to save, load data and perform other I/O operations

Linear algebra (scipy.linalg)

This package provides methods to perform various linear algebra operations.

Miscellaneous routines (scipy.misc)

This package provides a number of general purpose tools. These include methods for reading images, finding factorials, etc

Multi-dimensional image processing (scipy.ndimage)

Provides a number of methods to perform image processing.

Orthogonal distance regression (scipy.odr)

Orthogonal distance regression is a technique used for prediction

Optimization and root finding (scipy.optimize)

This package provides methods to find optimum parameters(minimum or maximum) for a given function.
It also has methods find roots for a given function.

Signal processing (scipy.signal)

This package provides methods for signal processing.

Sparse matrices (scipy.sparse)

Sparse matrices are matrices where a large number of elements are 0. This package provides ways to deal with such matrices

Sparse linear algebra (scipy.sparse.linalg)

This package provides methods to perform linear algebra operations on matrices assuming them to be sparse matrices.

Compressed Sparse Graph Routines (scipy.sparse.csgraph)

These are graph based methods for processing sparse graphs.

Spatial algorithms and data structures (scipy.spatial)

This package contains methods to works with spatial data.

Special functions (scipy.special)

This package provides a number of special functions(eg: Bessel function). We will be looking at the ndtri() function int eh later stages of the course.

Statistical functions (scipy.stats)

These include functions from the field of statistics(example computing the probability density function, etc.)

Statistical functions for masked arrays (scipy.stats.mstats)

This is similar to the statistical functions except that it works on masked arrays. Masked arrays are a data structure provided in numpy to represent arrays with missing values.

C/C++ integration (scipy.weave)

It is used to write code so that it can interact with C or C++ languages

Python for X-Informatics – SciPy II

Clustering - K-Means

Clustering an unsupervised machine learning operation(training data is not labelled) where the data points are divided into clusters.

Imports for Clustering

```
from scipy.cluster.vq import kmeans, vq, whiten
```

Clustering Data

Clustering data can be an nd-array from numpy. This arrays represents points one described by each row.

```
a = np.array([[1.0,1.0],[1.0,2.0],[4.0,1.0],[4.0,2.0]])
a
Out[31]:
array([[ 1.,  1.],
       [ 1.,  2.],
       [ 4.,  1.],
       [ 4.,  2.]])
```

Whiten

The **whiten()** method can be used to rescale the array as per the standard deviation. Each element in the array is divided by the corresponding standard deviation

```
b = whiten(a)
b
Out[33]:
array([[ 0.66666667,  2.        ],
       [ 0.66666667,  4.        ],
       [ 2.66666667,  2.        ],
       [ 2.66666667,  4.        ]])
```

Finding Cluster Centers

The kmeans method finds the centroids and errors if the data(a in this case) and number of clusters(2 in this case) is provided. The return value is a tuple where the first element is an element codebook which tells the position of the cluster centers and the second element is a value which relates to the error.

```
k = kmeans(a,2)
k
Out[35]:
(array([[ 1. ,  1.5],
       [ 4. ,  1.5]]), 0.5)
```

Assigning Centroids

The vq() method is used to assign centroids to data points, given a set of data points and the centroid position(code book). The centroid number and the distance to the centroid are returned in separate lists.

```
vq(a,k[0])
```
Out[36]: (array([0, 0, 1, 1]), array([0.5, 0.5, 0.5, 0.5]))

Performing similar clustering operation on whitened data

```
k1 = kmeans(b,2)
k1
```
Out[39]:
(array([[1.33333333, 3.33333333],
[2.66666667, 2.]]),
0.98105825289544568)
```
vq(b,k1[0])
```
Out[40]:
(array([0, 0, 1, 0]),
array([1.49071198, 0.94280904, 0. , 1.49071198]))

Special functions(ndtri)

ndtri() is a special function. For a given value y, ndtri(y) gives the value x for which area under the standard gaussian is y. it accepts both scalars and vectors

```
from scipy.special import ndtri
ndtri(0.4)
```
Out[42]: -0.25334710313579972
```
ndtri(np.array([0.1,0.2,0.7,0.8]))
```
Out[43]: array([-1.28155157, -0.84162123, 0.52440051, 0.84162123])