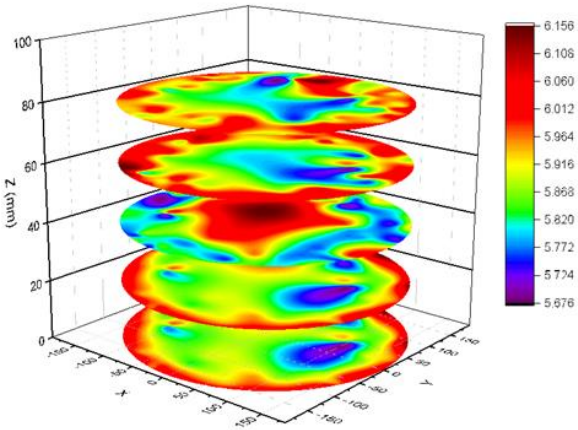
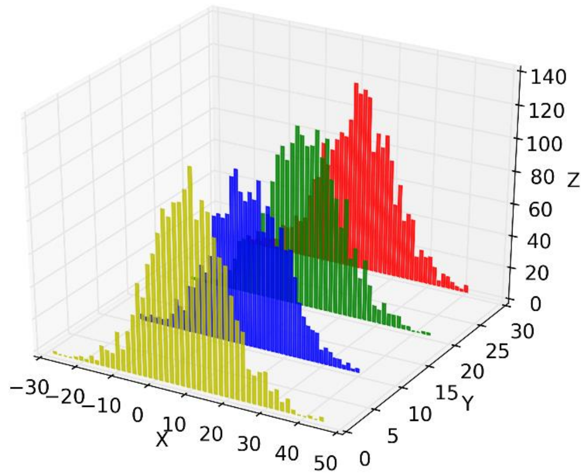
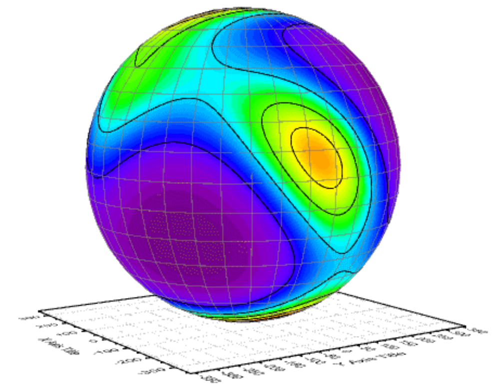


# Programming for BIG Data

## 11. Scientific Python

# 11. Scientific Python

Ipython
NumPy
SciPy
ScikitLearn
MatploitLib
Pandas

# The ideal programming environment for computational mathematics enjoys the following characteristics:

- It must be based on a computer language that allows the user to work quickly and integrate systems effectively. Ideally, the computer language should be portable to all platforms: Windows, Mac OS X, Linux, Unix, Android, and so on.
- Besides running the compiled code, the programming environment should allow the possibility of interactive sessions as well as scripting capabilities for quick experimentation.
- Different coding paradigms should be supported—imperative, object-oriented, and/or functional coding styles.
- It should be an open source software, that allows user access to the raw data code, and allows the user to modify basic algorithms if so desired.

# What's out there …

Among the best-known environments for numerical computations used by the scientific community is **MATLAB**, which is commercial, expensive, and which does not allow any tampering with the code.

**Maple** and **Mathematica** are more geared towards symbolic computation, although they can match many of the numerical computations from MATLAB. These are, however, also commercial, expensive, and closed to modifications.

One environment that combines the best of all worlds is Python with the open source libraries **NumPy** and **SciPy** for numerical operations.

# Python

The first property that attracts users to Python is, without a doubt, its code readability. The syntax is extremely clear and expressive.

It has the advantage of supporting code written in different paradigms: object oriented, functional, or old school imperative.

Python comes equipped with a large library of packages for machine learning tasks:

## Python comes equipped with a large library of packages for machine learning tasks:

The **IPython** console.

**NumPy**, which is an extension that adds support for multi-dimensional arrays, matrices, and high-level mathematical functions.

**SciPy**, which is a library of scientific formulae, constants, and mathematical functions.

**Scikit-learn**, which is a library for machine learning tasks Such as classification, regression, and clustering

# Python comes equipped with a large library of packages for machine learning tasks:

**Matplotlib**, which is for creating plots

**Pandas** provides fast, flexible and expressive data structures designed to make working with structured (tabular, multidimensional, potentially heterogeneous) and time series data both easy and intuitive.  It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.

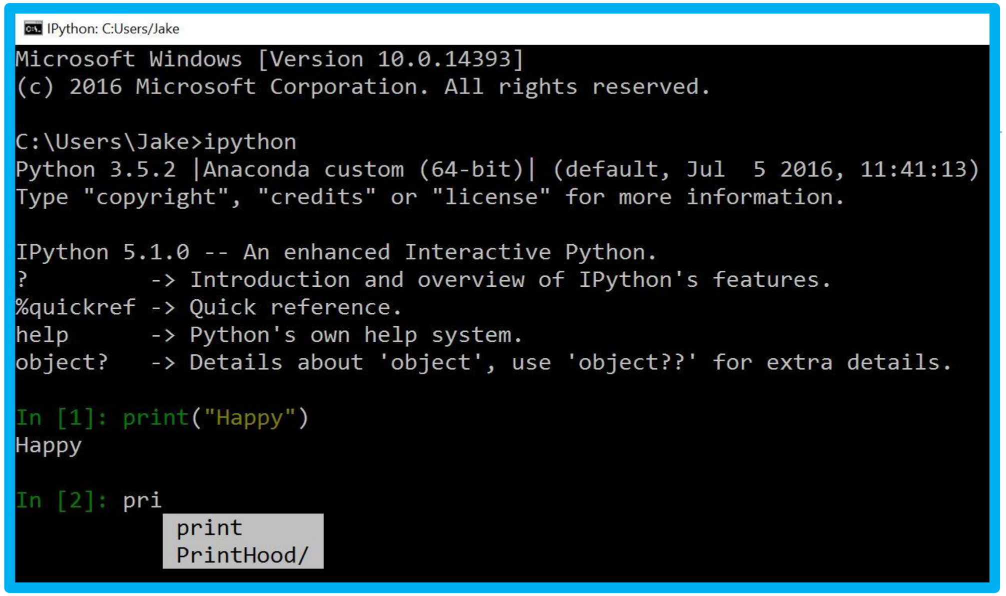
# The **IPython** console.

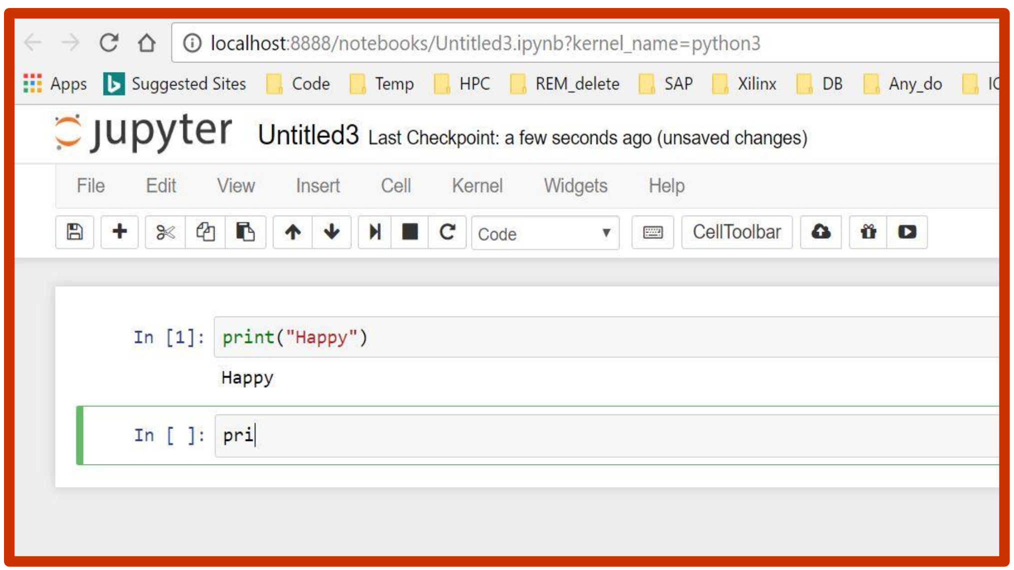
At the core of IPython is the IPython console: a powerful interactive interpreter that allows you to test your ideas in a very fast and intuitive way. Instead of having to create, save, and run a file every time you want to test a code snippet, you can simply type it into a console.

A powerful feature of IPython is that it decouples the traditional read-evaluate-print loop that most computing platforms are based on.

IPython puts the evaluate phase into its own process: a kernel (not to be confused with the kernel function used in machine learning algorithms).

More than one client can access the kernel. This means you can run code in a number of files and access them, for example, running a method from the console. Also, the kernel and the client do not need to be on the same machine. This has powerful implications for distributed and networked computing.




The IPython notebook has merged into another project known as Jupyter (jupyter.org).

This web application is a powerful platform for numerical computing in over 40 languages.

The notebook allows you to share and collaborate on live code and publish rich graphics and text.

# Data types in python

There is a hierarchy of types for representing data in Python.

At the root are **immutable** objects such as integers, floats, and Boolean.

Built on this, we have sequence types. These are ordered sets of objects indexed by non-negative integers. They are iterative objects that include strings, lists, and tuples. Sequence types have a common set of operations such as returning an element ( s[i] ) or a slice ( s[ i : j ] ), and finding the length ( len(s) ) or the sum (sum(s)).

Finally, we have mapping types. These are collections of objects indexed by another collection of key objects. Mapping objects are unordered and are indexed by numbers, strings, or other objects. The built-in Python mapping type is the dictionary.

# NumPy

NumPy builds on these data objects by providing two further objects: an N-dimensional array object (**ndarray**) and a universal function object (**ufunc**).

The ufunc object provides element-by-element operations on ndarray objects, allowing typecasting and array broadcasting. Typecasting is the process of changing one data type into another, and broadcasting describes how arrays of different sizes are treated during arithmetic operations.

There are sub-packages for:

- Linear algebra (linalg)
- Random number generation (random)
- Discrete Fourier transforms (fft)
- Unit testing (testing)

Indexing and slicing in NumPy builds on the slicing and indexing techniques used in sequences.

We are already familiar with slicing sequences, such as lists and tuples, in Python using the [ i :  j : k] syntax, where:

i is the start index,
j is the end,
k is the step.

NumPy extends this concept of the selection tuple to N-dimensions.

# Lets try some code

numpy.arange([start, ]stop, [step, ]dtype=None)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop) (in other words, the interval including start but excluding stop). For integer arguments the function is equivalent to the Python built-in range function, but returns an ndarray rather than a list.

```
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

# Lets try some code

Lets create an array of 60 integers

```
In [1]: import numpy as np

In [2]: a = np.arange(60)

In [3]: print(a)
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
 50 51 52 53 54 55 56 57 58 59]
```

# Lets slice it… a[???]

```
In [23]: a
Out[23]:
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59])

In [24]: a[6]
Out[24]: 6

In [25]: a[2:5]
Out[25]: array([2, 3, 4])

In [26]: a[2:12:2]
Out[26]: array([ 2,  4,  6,  8, 10])

In [27]: a[-1]
Out[27]: 59

In [28]: a[-10:-1]
Out[28]: array([50, 51, 52, 53, 54, 55, 56, 57, 58])
```

# Lets try some code

numpy.ndarray.shape

May be used to "reshape" the array, as long as this would not require a change in the total number of elements

```
In [4]: a.reshape(5,12)
Out[4]:
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],
       [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47],
       [48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

## Lets try some code

numpy.ndarray.shape

May be used to “reshape” the array, as long as this would not require a change in the total number of elements

```
In [4]: a.reshape(5,12)
Out[4]:
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],
       [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47],
       [48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

## Lets try some code

numpy.ndarray.shape

May be used to "reshape" the array, as long as this would not require a change in the total number of elements.

Lets re shape out array into a 5 by 12 structure ➔ a.reshape(?,?)

```
In [4]: a.reshape(5,12)
Out[4]:
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35],
       [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47],
       [48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
```

```
In [11]: a = a.reshape(3,4,5)

In [12]: a
Out[12]:
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]],

       [[40, 41, 42, 43, 44],
        [45, 46, 47, 48, 49],
        [50, 51, 52, 53, 54],
        [55, 56, 57, 58, 59]]])
```

```
In [14]: a[2,1,3]
Out[14]: 48
```

## Lets slice it… a[???]

Using the ellipse (…), we can select any remaining unspecified dimensions. For example, a[...,1] is equivalent to a[:,:,1]

What will a.[:,:,1] give us ?

# Lets slice it… a[???]

With slicing, we are creating views; the original array remains untouched, and the view retains a reference to the original array. This means that when we create a slice, even though we assign it to a new variable, if we change the original array, these changes are also reflected in the new array.

```
In [40]: b = a[2,2,0:2]

In [41]: b
Out[41]: array([50, 51])

In [42]: print(b)
[50 51]

In [43]: a[2] = 0
```

```
In [44]: a
Out[44]:
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]],

       [[ 0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0]]])
```

## Copies deep / shallow

Here, a and b are referring to the same array. When we assign values in a, this is also reflected in b. To copy an array rather than simply make a reference to it, we use the deep copy() function from the copy package in the standard library:

*import copy*
*c = copy.deepcopy(a)*

Here, we have created a new independent array, c. Any changes made in array a will **not** be reflected in array c.

# Lets slice it…  a[???]

```
In [45]: b
Out[45]: array([0, 0])

In [46]:

In [46]: import copy

In [47]: c = copy.deepcopy(a)

In [48]: c
Out[48]:
array([[[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]],

       [[20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29],
        [30, 31, 32, 33, 34],
        [35, 36, 37, 38, 39]],

       [[ 0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0],
        [ 0,  0,  0,  0,  0]]])
```

```
In [49]: c[1]
Out[49]:
array([[20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34],
       [35, 36, 37, 38, 39]])

In [50]: a[1] = 0

In [51]: a[1]
Out[51]:
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])

In [52]: c[1]
Out[52]:
array([[20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34],
       [35, 36, 37, 38, 39]])
```

# Numpy support vectorized operations

```
In [54]: a = np.arange(5)

In [55]: a
Out[55]: array([0, 1, 2, 3, 4])

In [56]: a * 2
Out[56]: array([0, 2, 4, 6, 8])

In [57]:
```

## Mathematical operations

As you would expect, you can perform mathematical operations such as addition, subtraction, multiplication, as well as the trigonometric functions on NumPy arrays.

Arithmetic operations on different shaped arrays can be carried out by a process known as **broadcasting**. When operating on two arrays, NumPy compares their shapes element-wise from the trailing dimension. Two dimensions are compatible if they are the same size, or if one of them is 1. If these conditions are not met, then a ValueError exception is thrown.

This is all done in the background using the ufunc object. This object operates on ndarrays on a element-by-element basis. They are essentially wrappers that provide a consistent interface to scalar functions to allow them to work with NumPy arrays. There are over 60 ufunc objects covering a wide variety of operations and types. The ufunc objects are called automatically when you perform operations such as adding two arrays using the + operator

# Matplotlib

Matplotlib is an essential tool for visualizing data in Python.

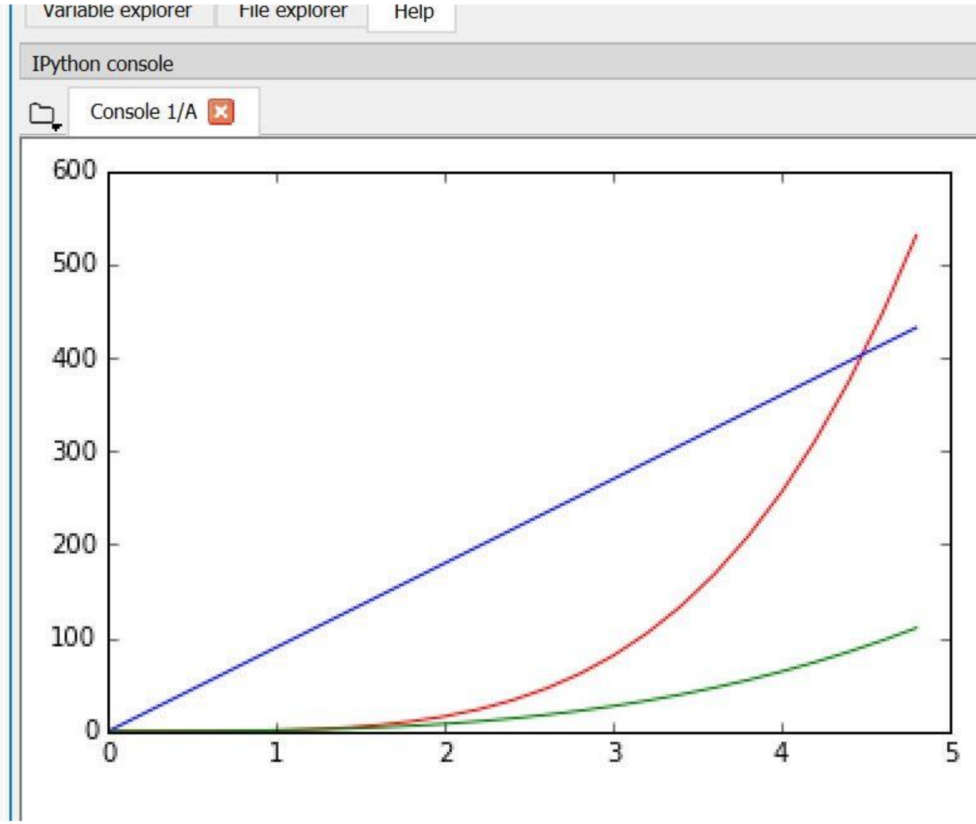
It has a sub package PyPlot with a range of graphical functions that can be invoked on a PyPlot instance. At the core of PyPlot is the plot method. The simplest implementation is to pass plot a list or a 1D array.

If only one argument is passed to plot, it assumes it is a sequence of y values, and it will automatically generate the x values.

More commonly, we pass plot two 1D arrays or lists for the co-ordinates x and y. The plot method can also accept an argument to indicate line properties such as line width, color, and style.

# Matplotlib

```python
@author: Jake
"""

import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0.,5., 0.2)

plt.plot(x, x**4, 'red', x, x*90, \
         'blue', x, x**3, 'green')

plt.show()
```

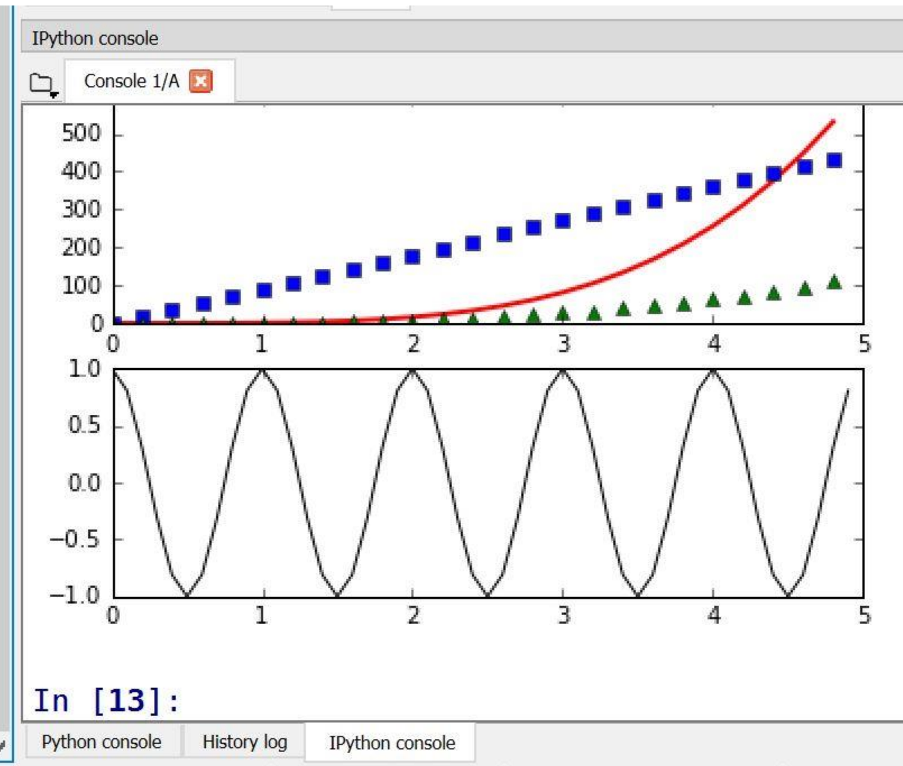



This code prints three lines in different styles: a red line, blue squares, and green triangles. Notice that we can pass more than one pair of coordinate arrays to plot multiple lines.  For a full list of line styles, type the help(plt.plot) function.

Pyplot applies plotting commands to the current axes. Multiple axes can be created using the subplot command.

The subplot() command specifies numrows, numcols, fignum where fignum ranges from 1 to numrows*numcols.

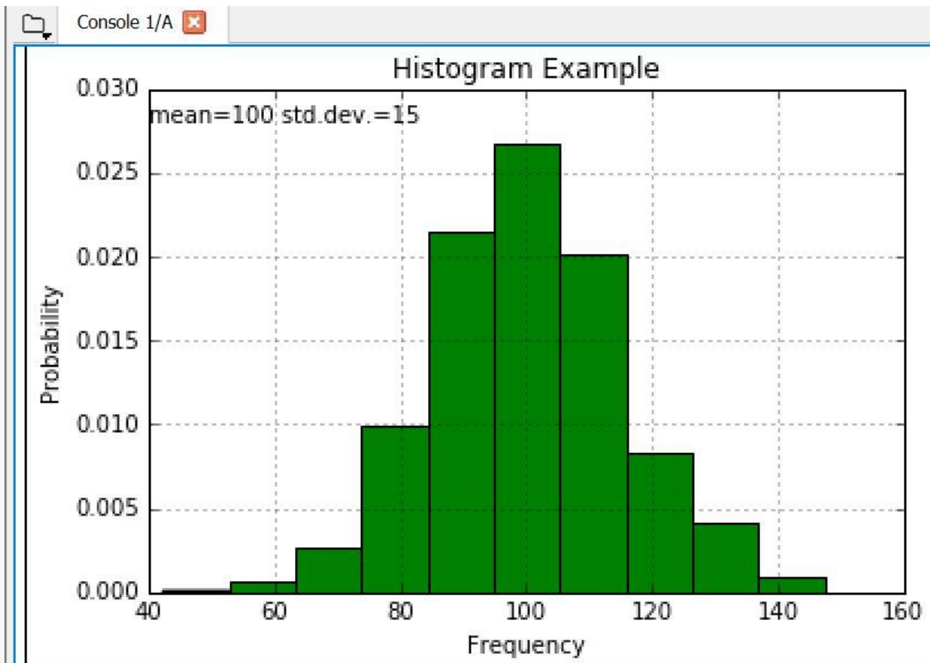
```
"""

import numpy as np
import matplotlib.pyplot as plt

x1 = np.arange(0., 5., 0.2)
x2 = np.arange(0., 5., 0.1)

plt.figure(1)
plt.subplot(211)

plt.plot(x1, x1**4, 'r', x1, x1*90, \
         'bs', x1, x1**3, 'g^',      \
         linewidth=2.0)

plt.subplot(212)

plt.plot(x2,np.cos(2*np.pi*x2), 'k')
plt.show()
```



Another useful plot is the histogram. The hist() object takes an array, or a sequence of arrays, of input values. The second parameter is the number of bins.

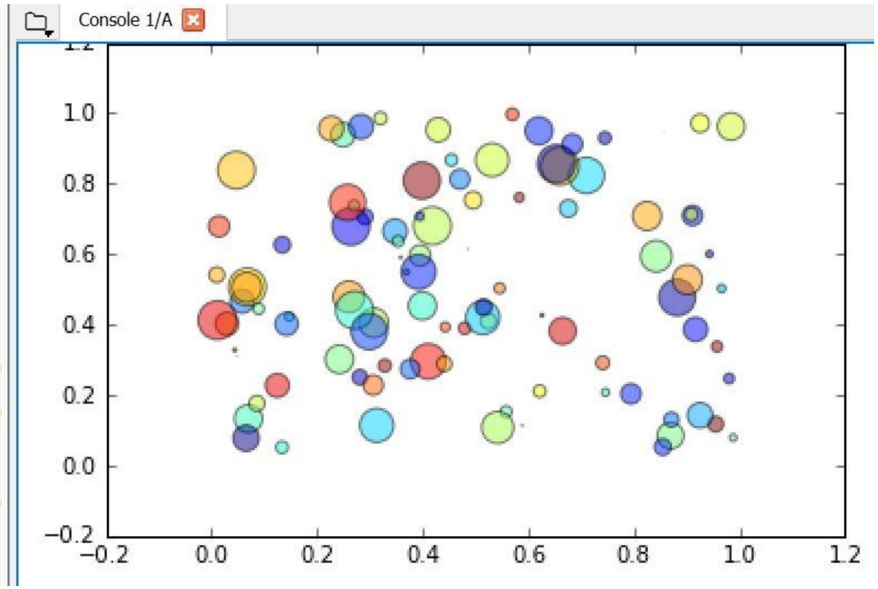
Lets create a histogram with labelled axes, and use 10 bins. Also by setting the normed parameter to 1 the counts are normalized to form a probability density.

```

import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(1000)
n, bins, patches = plt.hist(x, 10, \
                            normed=1,\
                            facecolor='g')
plt.xlabel('Frequency')
plt.ylabel('Probability')
plt.title('Histogram Example')
plt.text(40,.028, 'mean=100 std.dev.=15')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```



Next lets look at the scatter plot. The scatter object takes two sequence objects, such as arrays, of the same length and optional parameters to denote color and style attributes.

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

N = 100
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
#colors=('r','b','g')
area = np.pi * (10 * np.random.rand(N))**2 # 0
plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



# Pandas

The Pandas library builds on NumPy by introducing several useful data structures and functionalities to read and process data. Pandas is a great tool for general data munging. It easily handles common tasks such as dealing with missing data, manipulating shapes and sizes, converting between data formats and structures, and importing data from different sources.

The main data structures introduced by Pandas are:

- Series
- The DataFrame
- Panel

The DataFrame is probably the most widely used. It is a two-dimensional structure that is effectively a table created from either a NumPy array, lists, dicts, or series. You can also create a DataFrame by reading from a file.

## Pandas

The Pandas library builds on NumPy by introducing several useful data structures and functionalities to read and process data. Pandas is a great tool for general data munging. It easily handles common tasks such as dealing with missing data, manipulating shapes and sizes, converting between data formats and structures, and importing data from different sources.

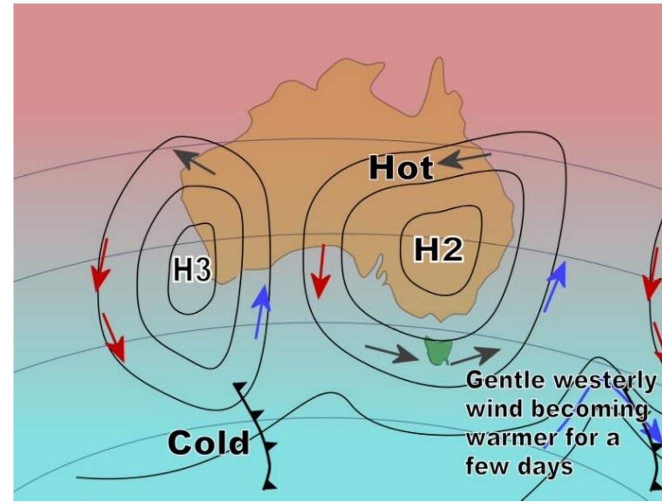
The main data structures introduced by Pandas are:

- Series
- The DataFrame
- Panel

The DataFrame is probably the most widely used. It is a two-dimensional structure that is effectively a table created from either a NumPy array, lists, dicts, or series. You can also create a DataFrame by reading from a file.

## Pandas

Lets consider Pandas by working through some data.




An interesting example is the historical weather observations from the Hobart weather station in Tasmania and consider the task of "*discovering how the daily maximum temperature has changed over time*".

First lets create a DataFrame from the data
And display the first few lines of it

```
import pandas as pd

df = pd.read_csv('C:/Users/Jake/sampleData.csv')

df.head()
```

The product code and the station number are the same for each row and that this information is superfluous. Also, the days of accumulated maximum temperature are not needed for our purpose, so we will delete them as well:

```
In [47]: runfile('C:/Users/Jake/lane.py', wdir='C:/Users/Jake')
  Product code  Bureau of Meteorology station number  Year  Month  Day  \
0    IDCJAC0010                                 94029  1882      1    1
1    IDCJAC0010                                 94029  1882      1    2
2    IDCJAC0010                                 94029  1882      1    3
3    IDCJAC0010                                 94029  1882      1    4
4    IDCJAC0010                                 94029  1882      1    5

   Maximum temperature (Degree C)  \
0                             NaN
1                             NaN
2                             NaN
3                             NaN
4                             NaN

   Days of accumulation of maximum temperature Quality
```

The product code and the station number are the same for each row and that this information is superfluous. Also, the days of accumulated maximum temperature are not needed for our purpose, so we will delete them as well:

```python
import pandas as pd

df = pd.read_csv('C:/Users/Jake/sampleData.csv')

del df['Bureau of Meteorology station number']
del df['Product code']
del df['Days of accumulation of maximum temperature']

df.head()
```

```
Console 1/A

Out[54]:
   Year  Month  Day  Maximum temperature
(Degree C) Quality
0  1882      1    1
NaN      NaN
1  1882      1    2
NaN      NaN
2  1882      1    3
NaN      NaN
3  1882      1    4
NaN      NaN
4  1882      1    5
NaN      NaN
```

## Pandas

pandas

Let's make our data a little easier to read by shorting the column labels:

```
df=df.rename(columns={'Maximum temperature (Degree C)':'maxtemp'})
```

```
del df['Bureau of Meteorology station number']
del df['Product code']
del df['Days of accumulation of maximum temperature']

df=df.rename(columns={'Maximum temperature (Degree C)

df.head()
```

```
In [2]: df.head()
Out[2]:
   Year  Month  Day  maxtemp Quality
0  1882      1    1      NaN     NaN
1  1882      1    2      NaN     NaN
2  1882      1    3      NaN     NaN
3  1882      1    4      NaN     NaN
4  1882      1    5      NaN     NaN
```

We are only interested in data that is of high quality, so we include only records that have a Y in the quality column:

```
df=df.rename(columns={'Maximum temperature (Degree C)

df=df[(df.Quality=='Y')]
```

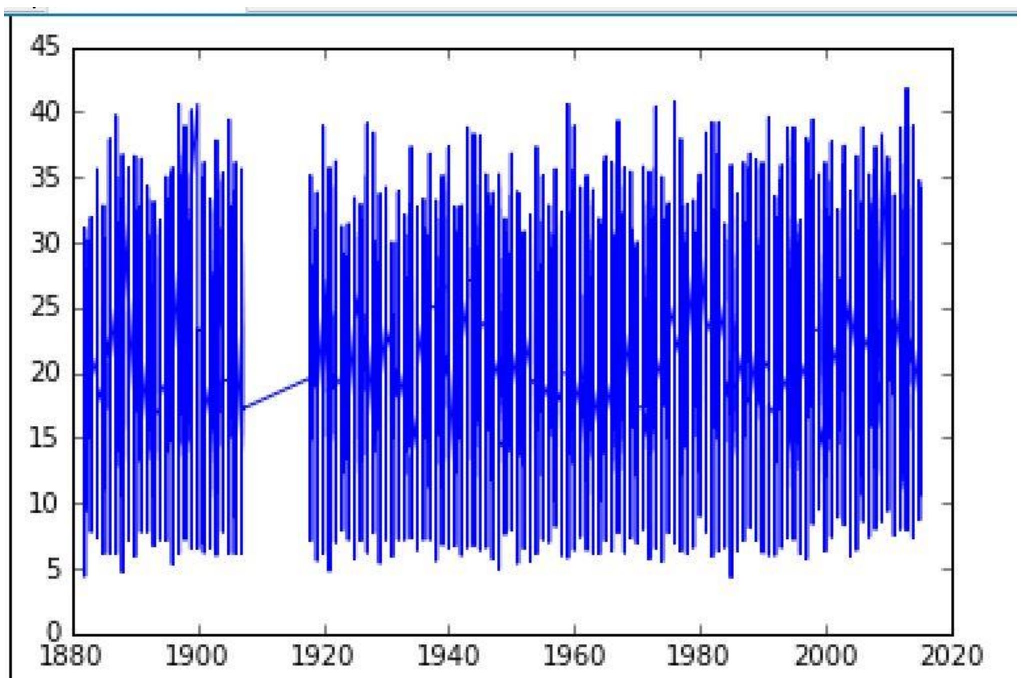
```
Out[5]:
    Year  Month  Day  maxtemp Quality
91  1882      4    2     15.6       Y
92  1882      4    3     14.4       Y
93  1882      4    4     12.8       Y
94  1882      4    5     10.0       Y
95  1882      4    6     10.6       Y
```

We can get a statistical summary of our data using the describe() command

```
In [6]: df.describe()
Out[6]:
               Year         Month           Day      maxtemp
count  44250.000000  44250.000000  44250.000000  44250.000000
mean    1952.207503      6.536339     15.734870     16.929941
std       38.212270      3.446311      8.802089      5.030362
min     1882.000000      1.000000      1.000000      4.300000
25%     1924.000000      4.000000      8.000000     13.300000
50%     1954.000000      7.000000     16.000000     16.400000
75%     1985.000000     10.000000     23.000000     20.000000
max     2015.000000     12.000000     31.000000     41.800000
```

Lets look at our data in MatplotLib

```
import matplotlib.pyplot as plt
plt.plot(df.Year, df.maxtemp)
```



# SciPy

SciPy (pronounced sigh pi) adds a layer to NumPy that wraps common scientific and statistical applications on top of the more purely mathematical constructs of NumPy.

SciPy provides higher-level functions for manipulating and visualizing data, and it is especially useful when using Python interactively. SciPy is organized into sub-packages covering different scientific computing applications.

A list of the packages most relevant to ML and their functions appear as follows:

| Package | Description |
|---|---|
| `cluster` | This contains two sub-packages:<br>`cluster.vq` for K-means clustering and vector quantization.<br>`cluster.hierachy` for hierarchical and agglomerative clustering, which is useful for distance matrices, calculating statistics on clusters, as well as visualizing clusters with dendrograms. |
| `constants` | These are physical and mathematical constants such as *pi* and *e*. |
| `integrate` | These are differential equation solvers |
| `interpolate` | These are interpolation functions for creating new data points within a range of known points. |
| `io` | This refers to input and output functions for creating string, binary, or raw data streams, and reading and writing to and from files. |
| `optimize` | This refers to optimizing and finding roots. |
| `linalg` | This refers to linear algebra routines such as basic matrix calculations, solving linear systems, finding determinants and norms, and decomposition. |
| `ndimage` | This is N-dimensional image processing. |
| `odr` | This is orthogonal distance regression. |
| `stats` | This refers to statistical distributions and functions. |

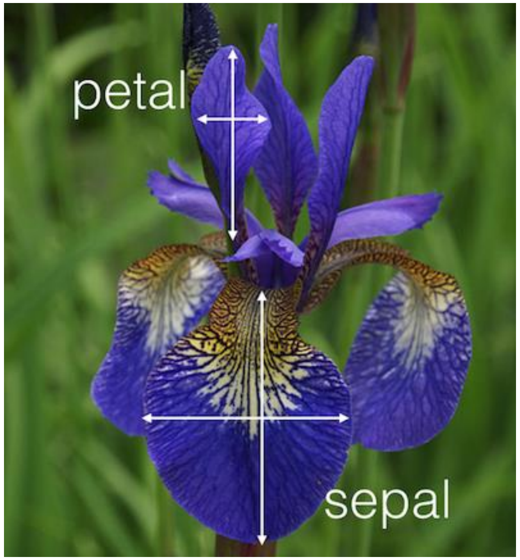
## ScikitLearn

This includes algorithms for the most common machine learning tasks, such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

Scikit-learn comes with several real-world data sets for us to practice with. Let's take a look at one of these—the Iris data set:

```
from sklearn import datasets

iris = datasets.load_iris()

iris_X = iris.data
iris_y = iris.target

print(iris_X.shape)

```

```
)
(150, 4)
>>>
```



## ScikitLearn

The data set contains 150 samples of three types of irises (Setosa, Versicolor, and Virginica), each with four features.

We can see that the four attributes, or features, are sepal width, sepal length, petal length, and petal width in centimeters. Each sample is associated with one of three classes. **Setosa**, **Versicolor**, and **Virginica**. These are represented by 0, 1, and 2 respectively.

Let's look at a simple classification problem using this data. We want to predict the type of iris based on its features: the length and width of its sepal and petals. Typically, scikit-learn uses estimators to implement a fit(X, y) method and for training a classifier, and a predict(X) method that if given unlabeled observations, X, returns the predicted labels, y. The fit() and predict() methods usually take a 2D array-like object. {more on this next semester}