Lecture 10 Numpy

IDS 400

Programming for Data Science in Business

Tentative schedule

Date	Lecture Number	Topics
08/24	Lecture 1	Introduction
08/31	Lecture 2	Basic
09/07	Lecture 3	Condition
09/14	Lecture 4	Loop
09/21	Lecture 5	String + **Quiz 1**
09/28	Lecture 6	Type
10/05	Lecture 7	Function
10/12	Lecture 8	File + **Quiz 2**
10/19	Lecture 9	Pandas
10/26	Lecture 10	Numpy
11/02	Lecture 11	Machine Learning
11/09	Lecture 12	Visualization
11/16	Lecture 13	Web Scraping & Deep Learning
11/23	***Thanksgiving***	***No lecture***
11/30	**Final presentation**	In class presentation
12/05	**Project submission due**	

Overview

Steps for preparing data:

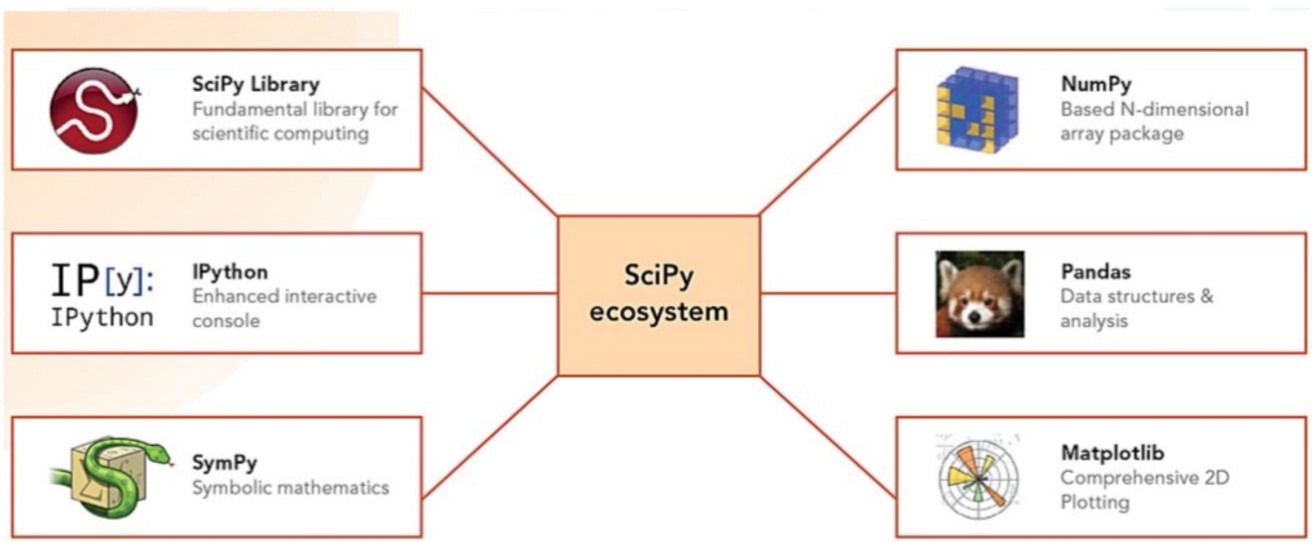
- Installing/Importing required packages
- Create a Dataframe using Pandas
- Dealing with missing data (remove/replace)
- Removing duplicate data (depends on what unique variables you would like to keep!)
- Filtering/Querying Data
- Using lambda function (rows/columns)
- Merging different datasets (left/right/outer/inner)

Scientific Applications

- There are a few packages available for scientific computing that extend Python's basic math module:
  - **NumPy** - numerical and scientific function libraries.
  - **Numba** - Python compiler that support JIT compilation.
  - **ALGLIB** - numerical analysis library.
  - **PyGSL** - Python interface for GNU Scientific Library.
  - **ScientificPython** - collection of scientific computing modules.

SciPy

- By far, the most commonly used packages are those in the SciPy stack. SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. This stack can let you do scientific computing in Python. The six most important packages found in the SciPy stack include:



SciPy

- By far, the most commonly used packages are those in the SciPy stack. SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. This stack can let you do scientific computing in Python. The six most important packages found in the SciPy stack include:
  - **NumPy** - fundamental package for scientific computing.
  - **SciPy** - efficient numerical routines.
  - **Matplotlib** - plotting library.
  - **IPython** – interactive computing.
  - **SymPy** – symbolic computation library.
  - **Pandas** – data analysis library.

NumPy

- The fundamental package for scientific computing with Python. It contains:
  - A powerful N-dimensional array (ndarray) object.
  - Sophisticated (broadcasting/universal) functions.
  - Tools for integrating C/C++ and Fortran code.
  - Useful linear algebra, Fourier transform, and random number capabilities.

- Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

Import NumPy

```
import numpy
import numpy as np
from numpy import *   # import all functions

dir(np)  # show all the functions contained in Numpy
```

```
['ALLOW_THREADS',
 'AxisError',
 'BUFSIZE',
 'CLIP',
 'ComplexWarning',
 'DataSource',
 'ERR_CALL',
 'ERR_DEFAULT',
 'ERR_IGNORE',
 'ERR_LOG',
 'ERR_PRINT',
 'ERR_RAISE',
 'ERR_WARN',
 'FLOATING_POINT_SUPPORT',
 'FPE_DIVIDEBYZERO',
 'FPE_INVALID',
 'FPE_OVERFLOW',
 'FPE_UNDERFLOW',
 'False_',
```

```
help(np.ndarray)  # Help on class ndarray in module numpy
```

```
Help on class ndarray in module numpy:

class ndarray(builtins.object)
 |  ndarray(shape, dtype=float, buffer=None, offset=0,
 |          strides=None, order=None)
 |
 |  An array object represents a multidimensional, homogeneous array
 |  of fixed-size items.  An associated data-type object describes the
 |  format of each element in the array (its byte-order, how many bytes it
 |  occupies in memory, whether it is an integer, a floating point number,
 |  or something else, etc.)
 |
 |  Arrays should be constructed using `array`, `zeros` or `empty` (refer
 |  to the See Also section below).  The parameters given here refer to
 |  a low-level method (`ndarray(...)`) for instantiating an array.
 |
 |  For more information, refer to the `numpy` module and examine the
 |  methods and attributes of an array.
 |
```

NumPy Data Types

- By default, Python have these data types:
  - **strings** - used to represent text data, the text is given under quote marks. eg. "ABCD"
  - **integer** - used to represent integer numbers. eg. -1, -2, -3
  - **float** - used to represent real numbers. eg. 1.2, 42.42
  - **boolean** - used to represent True or False.
  - **complex** - used to represent a number in complex plain. eg. 1.0 + 2.0j, 1.5 + 2.5j

NumPy Data Types

- NumPy has some extra data types:
  - bool_, int_, intc, intp, u/int8, u/int16, u/int32,u/int64,float_, float16, float32, float64 complex_, complex64, complex128, …

  https://numpy.org/devdocs/user/basics.types.html

- NumPy numerical types are instances (data-type) objects:

  numpy.dtype(object, align, copy)

NumPy Data Types

```
# np.float is an alias for python float type.
# np.float32 is numpy specific 32-bit float types.

x = np.float32(1.0)
x
```

```
1.0
```

```
y = np.int_([1,2,4])
y
```

```
array([1, 2, 4])
```

```
y.dtype
```

```
dtype('int32')
```

```
z = np.arange(3, dtype=np.uint8)  # uint8:Unsigned integer (0 to 255)
z
```

```
array([0, 1, 2], dtype=uint8)
```

```
z.dtype
```

```
dtype('uint8')
```

arange is a function in numpy which generates an array with evenly spaced values.

NumPy Arrays

- The main feature of NumPy is an array object:
    - Arrays can be N-dimensional.
    - Array elements have to be the same type.
    - Array elements can be accessed, sliced, and manipulated in the same way as the lists.
    - The number of elements in the array is fixed.
    - Shape of the array can be changed.
- Built-in NumPy array creation:

  array(), arange(), ones(), zeros(), …

NumPy Arrays

```
np.array([2,3,1,0])
```

```
array([2, 3, 1, 0])
```

```
np.zeros((2,3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
np.array([[1,2.0],[0,0],[1+1j,3.]])
```

```
array([[1.+0.j, 2.+0.j],
       [0.+0.j, 0.+0.j],
       [1.+1.j, 3.+0.j]])
```

```
l = np.array([[1,2.0],[0,0],[1+1j,3.]])
```

```
l.shape
```

```
(3, 2)
```

```
np.zeros((2,3,4))
```

```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

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

NumPy Arrays

```
np.arange(10)
```

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

```
np.arange(2,10,dtype=np.float)
```

```
array([2., 3., 4., 5., 6., 7., 8., 9.])
```

```
np.arange(2,3,0.1)
```

```
array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

NumPy Arrays

```
a = np.arange(3)
```

```
a
```

```
array([0, 1, 2])
```

```
print(a)
```

```
[0 1 2]
```

Reshape function change the dimension of a numpy array

```
np.arange(9).reshape(3,3)
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
np.arange(24).reshape(2,3,4)
```

```
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]]])
```

NumPy Arrays

- **linspace(start, stop[, num, endpoint, retstep, dtype])**

  creates arrays with a specified number of elements, and spaced equally between the specified beginning and end values.

- **random.random([size])**

  creates arrays with random floats over the interval [0.,1.).

- **random.randint(low[, high, size, dtype])**

  creates arrays with random integers from low (inclusive) to high (exclusive).

NumPy Arrays

```
np.linspace(1.,4.,6)
```

```
array([1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```

```
np.random.random((2,3))
```

```
array([[0.48724427, 0.87441327, 0.45243135],
       [0.0660517 , 0.32073882, 0.30583646]])
```

```
np.random.randint(1,7,(2,6))
```

```
array([[4, 5, 6, 2, 4, 2],
       [2, 2, 5, 6, 5, 4]])
```

Statistics in NumPy

```
a = np.random.randint(1,7,(2,3))
a
```

```
array([[6, 1, 4],
       [4, 3, 5]])
```

Column-wise: axis= 0

Row-wise: axis = 1

- Then try the following command:
  - **a.max()**
  - **a.min()**
  - **a.argmax()**
  - **a.argmin()**
  - **np.amax(a,0)**
  - **np.amax(a,1)**
  - **np.mean(x)**
  - **np.std(x)**
  - **np.var(x)**

```
# Returns the indices of the maximum values.
a.argmax()
# Returns the indices of the maximum values along an axis.
np.argmax(a, 0)
```

```
array([0, 1, 1], dtype=int64)
```

```
#Return the maximum of an array or maximum along an axis.
np.amax(a, 0)
```

```
array([6, 3, 5])
```

```
np.amax(a, 1)
```

```
array([6, 5])
```

Sorting in NumPy

```
a = np.random.randint(1,7,(2,3))
a
```

```
array([[6, 1, 4],
       [4, 3, 5]])
```

Column-wise: axis= 0

Row-wise: axis = 1

Then try the following command:

- np.sort(a,axis=1)
- np.sort(a,axis=0)

```
np.sort(a,axis=0)
```

```
array([[4, 1, 4],
       [6, 3, 5]])
```

```
np.sort(a,axis=1)
```

```
array([[1, 4, 6],
       [3, 4, 5]])
```

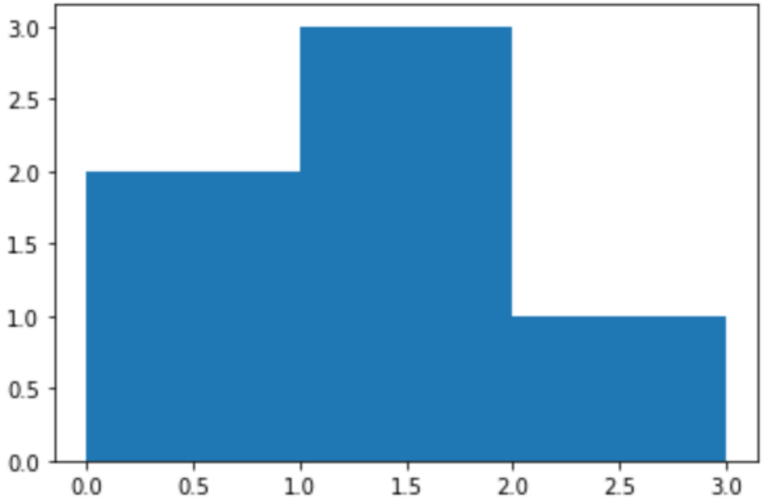
Histogram using matplotlib

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0.5, 0.7, 1.0, 1.2, 1.3, 2.1])
bins1 = np.array([0, 1, 2, 3])
print("ans=\n", np.histogram(x, bins1))
```

```
ans=
 (array([2, 3, 1], dtype=int64), array([0, 1, 2, 3]))
```

This part returns information about the histogram

```
plt.hist(x, bins=bins1)
plt.show()
```

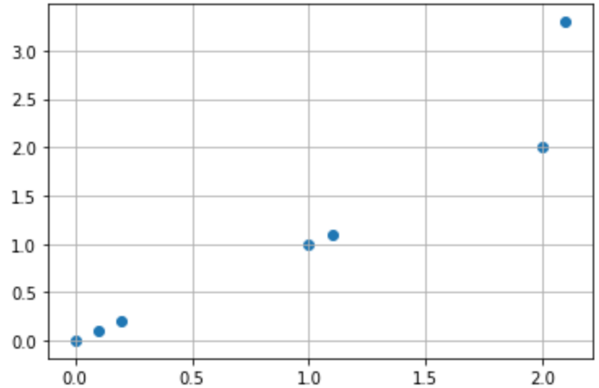


For histogram() and hist(), first parameter is data, second parameter is number of bins

2D Histogram using matplotlib

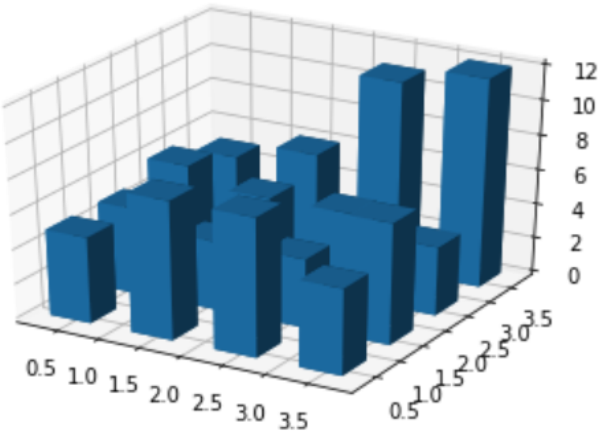
```
import matplotlib.pyplot as plt
import numpy as np

xedges= [0, 1, 2, 3]
yedges= [0, 1, 2, 3, 4]
x = np.array([0, 0.1, 0.2, 1., 1.1, 2., 2.1])
y = np.array([0, 0.1, 0.2, 1., 1.1, 2., 3.3])
H, xedges, yedges= np.histogram2d(x, y, bins=(xedges, yedges))
print("ans=\n", H)

plt.scatter(x, y)
plt.grid()
plt.show()
```

```
ans=
 [[3. 0. 0. 0.]
 [0. 2. 0. 0.]
 [0. 0. 1. 1.]]
```



If you want to plot a 3D histogram

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(19680801)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x, y = np.random.rand(2, 100) * 4
hist, xedges, yedges= np.histogram2d(x, y, bins=4, range=[[0, 4], [0, 4]])
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25, indexing="ij")
xpos = xpos.ravel()
ypos = ypos.ravel()
zpos = 0
dx = dy= 0.5 * np.ones_like(zpos)
dz = hist.ravel()
ax.bar3d(xpos, ypos, zpos, dx, dy, dz, zsort='average')
plt.show()
```



NumPy Linear Algebra

- All linear algebra routines expect an object that can be converted into a 2-dimensional array.
- The output is also a two-dimensional array.
    - **dot(a, b[, out])** - dot product of two arrays.
    - **trace(a[, offset, axis1, axis2, dtype, out])** - returns the sum along diagonals of the array.
    - **inv(a)** - computes the inverse of a matrix.
    - **eig(a)** - eigenvalues and right eigenvectors of a square array.
    - **solve(a, b)** - solves a linear matrix equation, or system of linear scalar equations.

NumPy LinAlg

```
from numpy import *
from numpy.linalg import *
a = array([[1.0,2.0],[3.0,4.0]])
print(a)
```

```
[[1. 2.]
 [3. 4.]]
```

```
a.transpose()
```

```
array([[1., 3.],
       [2., 4.]])
```

```
inv(a) #inverse
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{\underbrace{ad-bc}_{\text{determinant}}} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

NumPy LinAlg

```python
#unit 2x2 matrix; "eye"~"I”
u = eye(2)
u
```

```
array([[1., 0.],
       [0., 1.]])
```

```python
trace(u)
```

```
2.0
```

```python
j = array([[0.0,-1.0],[1.0,0.0]])
dot(j,j) # matrix product
```

```
array([[-1.,  0.],
       [ 0., -1.]])
```

```python
eig(j) # get eigenvalues & eigenvectors
```

```
(array([0.+1.j, 0.-1.j]),
 array([[0.70710678+0.j        , 0.70710678-0.j        ],
        [0.        -0.70710678j, 0.        +0.70710678j]]))
```

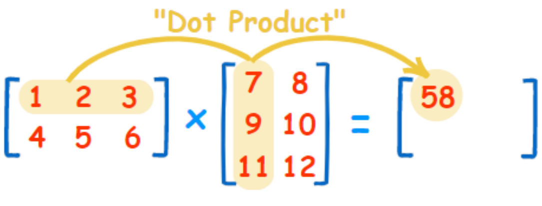
The trace of a square matrix A is defined to be the sum of elements on the main diagonal (from the upper left to the lower right) of A.

Let $\mathbf{A}$ be a matrix, with

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} -1 & 0 & 3 \\ 11 & 5 & 2 \\ 6 & 12 & -5 \end{pmatrix}$$

Then

$$\mathrm{tr}(\mathbf{A}) = \sum_{i=1}^{3} a_{ii} = a_{11} + a_{22} + a_{33} = -1 + 5 + (-5) = -1$$



About eigenvalue: https://mathworld.wolfram.com/Eigenvalue.html#:~:text=Entries%20%3E%20Interactive%20Demonstrations%20%3E-,Eigenvalue,144.

NumPy LinAlg

- There are two alcohol solutions: 50% & 90%.
- How many gallons of each solution to be mixed to get 10 gallons of 74% alcohol solution?
  - **$x1 + x2 = 10$**
  - **$0.5x1 + 0.9x2 = 0.74*10 = 7.4$**

AX = Y

```
A = array([[1.0,1.0],[0.5,0.9]])
Y = array([[10.0],[7.4]])
solve(A,Y) #solve linear equations
```

```
array([[4.],
       [6.]])
```

NumPy LinAlg

- A drone flying with the wind could cover in 2 hours.
- The return trip against the wind took 2.5 hours.
- How fast was the drone?
- What was the air speed?

2d + 2w = 60
2.5d – 2.w = 60

Trip	Rate		Time		Distance
With wind	d + w	×	2	=	60
Against wind	d – w	×	2.5	=	60

```
A = array([[2.0,2.0],[2.5,-2.5]])
Y = array([[60.0],[60.0]])
solve(A,Y) #solve linear equations
```

```
array([[27.],
       [ 3.]])
```

NumPy Matrix Versus Array

- NumPy matrices are strictly 2-dimensional, while NumPy arrays (ndarrays) are N-dimensional.
- Matrix objects are a subclass of ndarray, so they inherit all the attributes and methods of ndarrays.
- The main advantage of NumPy matrices is that they provide a convenient notation for matrix multiplication. e.g. If A and B are matrices, then A*B is their matrix product.

NumPy Matrices

```
A = matrix('1.0 2.0; 3.0 4.0')
A
```

```
matrix([[1., 2.],
        [3., 4.]])
```

```
type(A)
```

```
numpy.matrix
```

```
Y = matrix('5.0; 7.0')
```

```
print(A.I) #inverse
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

```
print(A.I*Y) #multiplication
```

```
[[-3.]
 [ 4.]]
```

```
solve(A,Y) #solve linear equations
```

```
matrix([[-3.],
        [ 4.]])
```

Note:

$$A.X = Y$$

Inv(A)*Y = X   or   A.I*Y = X

SciPy

- A collection of mathematical algorithms and convenient functions built on the NumPy extension of Python.
- An interactive Python session for manipulating and visualizing data.
- A data-processing and system-prototyping environment rivaling systems such as MATLAB, IDL, Octave, R-lab, and SciLab.

```
>>> import scipy
```

SciPy Sub -- Modules

- **cluster** -clustering algorithms
- **integrate** - integration and ordinary differential equation solvers.
- **interpolate** - interpolation and smoothing splines
- **io** - input and output
- **linalg** - linear algebra
- **optimize** - optimization and root-finding routines
- **stats** – statistical distributions and functions

```
>>> from scipy import linalg, optimize
>>> from scipy import *
```

SciPy Clustering

- **Clustering** - finds clusters and cluster centers in a set of unlabeled data.
- Intuitively, a cluster comprises a group of data points whose inter-point distances are small compared to the distances to points outside of the cluster.

SciPy *K-means Clustering*

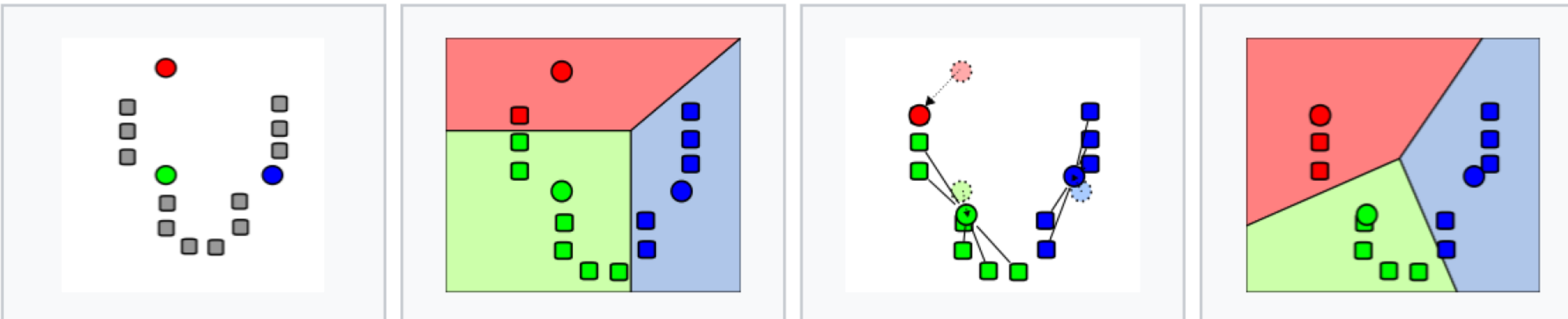
- scipy.cluster.vq
  - **kmeans(obs, k_or_guess[, iter, thresh, ...])**

    - perform k-means on a set of observation vectors forming k clusters.
  - **kmeans2(data, k[, iter, thresh, minit, ...])**

    -classify a set of observations into k clusters using the k-means algorithm.

- Given an initial set of k centers, the k-means algorithm alternates the two steps:
  1) For each center, we identify the subset of training points (its cluster) that is closer to it than any other center.
  2) The means of each feature for the data points in each cluster are computed, and this mean vector becomes the new center for that cluster.

K-means Clustering (k=3)



k initial "means" (in this case k=3) are randomly generated within the data domain (shown in color).

k clusters are created by associating every observation with the nearest mean.

The centroid of each of the k clusters becomes the new mean.

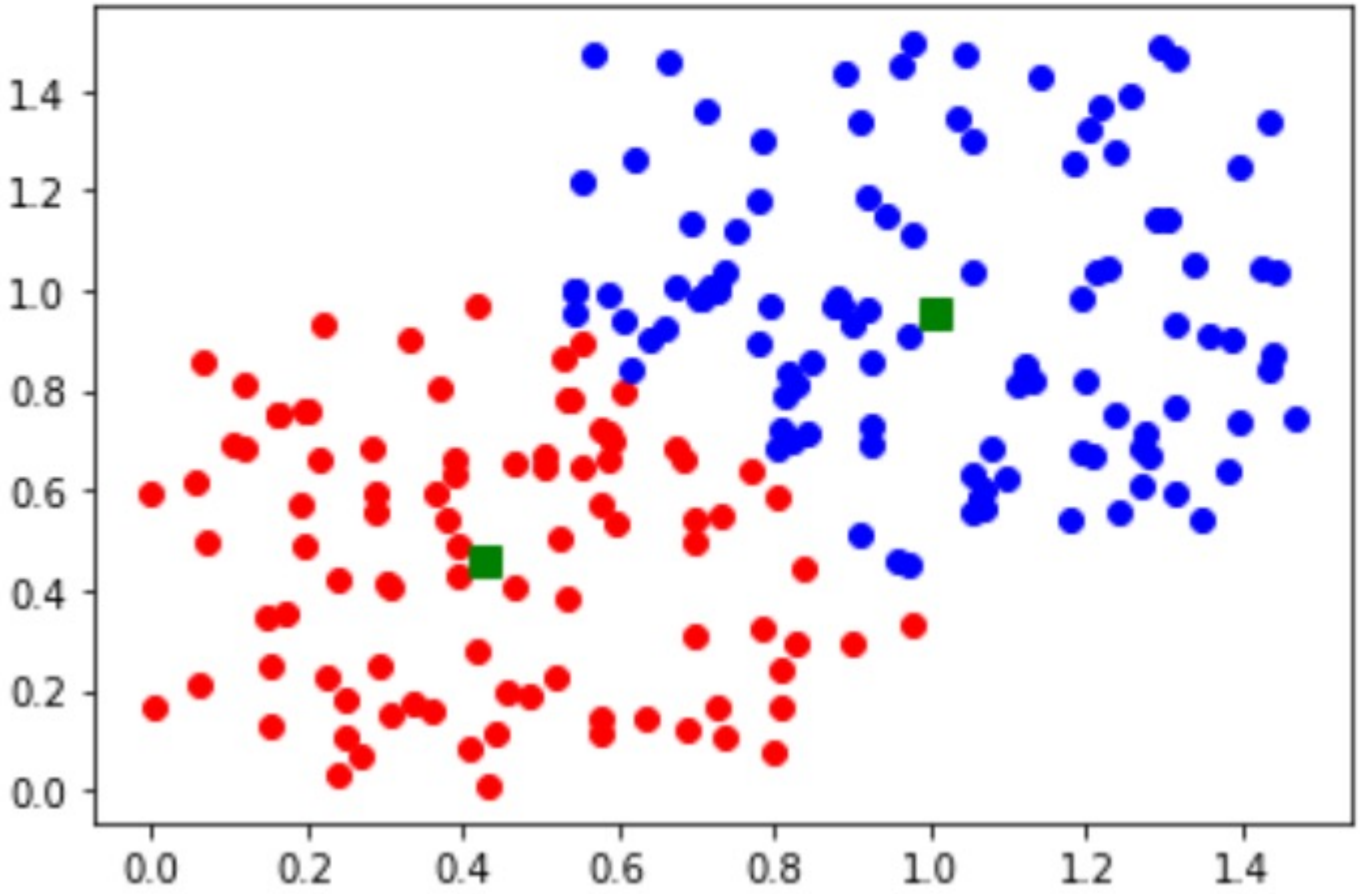
Steps 2 and 3 are repeated until convergence has been reached.

SciPy *2-means Clustering*

```
from pylab import *
from numpy import *
from numpy.random import *
from scipy.cluster.vq import*

# data generation
data = vstack((rand(100,2)+array([.5,.5]),rand(100,2)))
# computing k-means with k = 2 (2 clusters)
centroids,_ = kmeans(data,2)
# assign each sample to a cluster
index,_ = vq(data,centroids)
# some plotting using numpy's logical indexing
plot(data[index==0,0],data[index==0,1],'or',
data[index==1,0],data[index==1,1],'ob')
plot(centroids[:,0],centroids[:,1],'sg',markersize=8)
show()
```

‘o': Use circle markers.
'r': Use red color.
‘s’: Use square markers

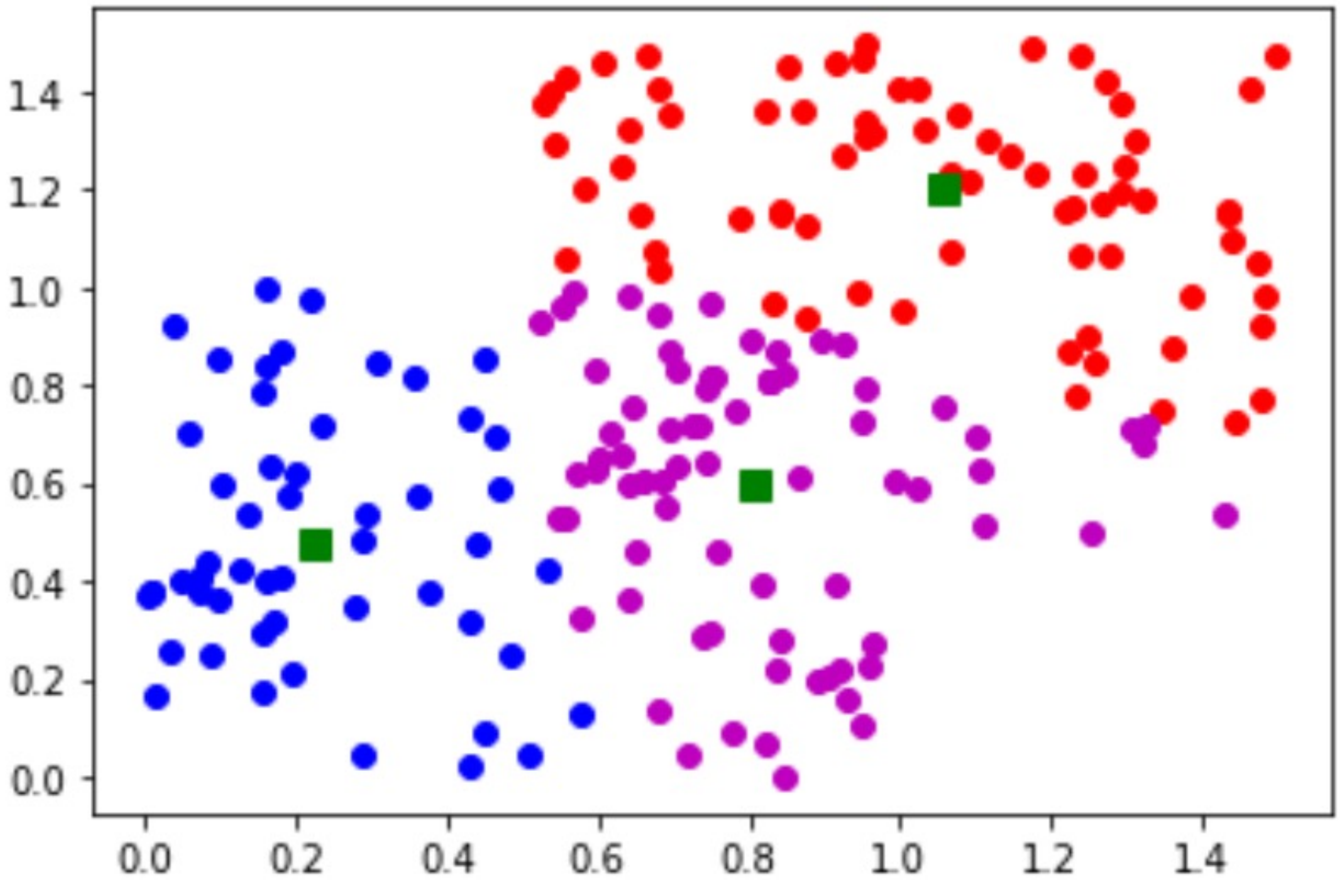
SciPy *2-means Clustering*



SciPy *3-means Clustering*

```
#data generation
data = vstack((rand(100,2)+array([.5,.5]),rand(100,2)))
# computing k-means with k = 3 (3 clusters)
centroids,_ = kmeans(data,3)
# assign each sample to a cluster
index,_ = vq(data,centroids)
# some plotting using numpy's logical indexing
plot(data[index==0,0],data[index==0,1],'or',
data[index==1,0],data[index==1,1],'ob',
data[index==2,0],data[index==2,1],'om')
plot(centroids[:,0],centroids[:,1],'sg',markersize=8)
show()
```

SciPy *3-means Clustering*



Exercise

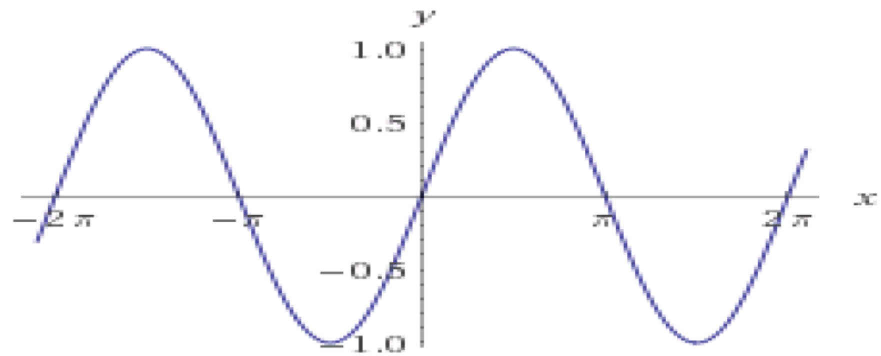
- Conduct a 4-means clustering and plot it using matplotlib

SciPy Integration

- Methods for Integrating Functions given a function object:
  - **quad** - general purpose integration
  - **dblquad** - general purpose double integration
  - **tplquad** - general purpose triple integration
  - **fixed_quad** - integrate f(x) using Gaussian quadrature
  - **quadrature** - integrate with tolerance using Gaussian quadrature
  - **romberg** - integrate f(x) using Romberg integration
- Methods for I.F. given a fixed set of samples:
  - **trapz** - use trapezoidal rule to compute integral
  - **cumtrapz** -use trapezoidal rule to cumulatively compute integral
  - **simps** - use Simpson's rule to compute integral
  - **romb** - use Romberg Integration to compute integral

SciPy Integration

- **np.sin** defines the sine function
- Integral x=0 to x=π using **quad**



$$\int \sin(x)\,dx$$

$$\int_0^{\pi} \sin(x)dx = -\cos(x)\Big|_0^{\pi} = -\cos(\pi) - -\cos(0) = -(-1) - (-1) = 1 + 1 = 2$$

```
from scipy.integrate import *
result =scipy.integrate.quad(np.sin,0,np.pi)
print(result)

# 2 with a very small error margin!
```

```
(2.0, 2.220446049250313e-14)
```

```
result = scipy.integrate.quad(np.sin,- np.inf,+np.inf)

print(result)
# Integral does not converge
```

```
(0.0, 0.0)
```

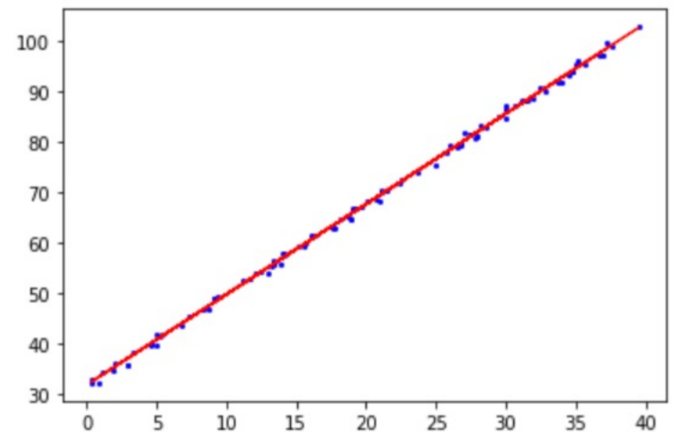
SciPy Optimization

- Provides several commonly used optimization algorithms:
  - Unconstrained and constrained minimization of multivariate scalar functions (minimize) using BFGS, Nelder-Mead Simplex, Newton Conjugate Gradient, COBYLA, SLSQP, …
  - Global (brute-force) optimization routines (e.g. basinhopping, differental_evoluton)
  - Least-squares minimization (least_squares) and curve iTng (curve_it) algorithms
  - Scalar univariate functions minimizers (minimize_scalar) and root finders (newton)
  - Multivariate equation system solvers (root) using hybrid Powell, Levenberg-Marquardt, large-scale Newton-Krylov, …

https://docs.scipy.org/doc/scipy/reference/optimize.html

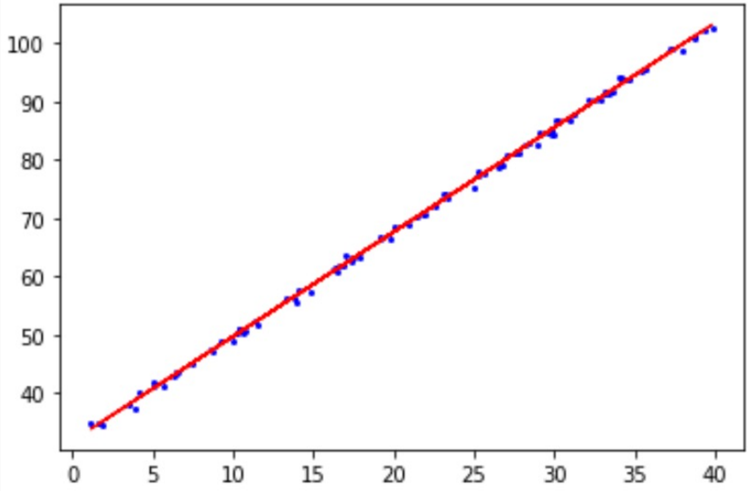
SciPy Curve Fitting

```
from pylab import *
from numpy import *
from numpy.random import *
from scipy.optimize import *

# linear regression
def linreg(x,a,b):
    return a*x+b

# data generation
input1 = randint(0,40,100)
x = input1 + rand(100)
y = (input1 * 1.8 + 32) + rand(100)

# curve fitting
attributes,variances= curve_fit(linreg,x,y)
# estimated y
y_modeled= x*attributes[0]+attributes[1]

# plot true and modeled results
plot(x,y,'ob',markersize=2)
plot(x,y_modeled,'-r',linewidth=1)
show()
```



SciPy Linear Regression

```
from pylab import *
from numpy import *
from scipy.stats import *

# data generation
input1 = random.randint(0,40,100)
x = input1+rand(100)
y = (input1*1.8+32)+rand(100)

# linear regression
slope,intercept,r_value,p_value,slope_std_error= stats.linregress(x,y)
# estimated y
y_modeled= x*slope+intercept

# plot true and modeled results
plot(x,y,'ob',markersize=2)
plot(x,y_modeled,'-r',linewidth=1)
show()
```



Lab *Numpy*