Python Preliminaries

Luke Shulenburger

(adapted from previous school's presentation by James Shepherd)

Pre - preliminaries

- Class arrangement
- Have working python (2 or 3), need numpy, scipy, matplotlib, pandas, seaborn
  - Anaconda works well for instance
- Clone the git repository with materials from the school:
  - git clone https://github.com/lkwagner/StochasticSchool.git
  - Everything from today is in Day1 (update frequently)

Will use python throughout the week

Two parts

1. Introduce a few basic libraries
2. Exercises to solve in groups

Part 1: Basic Libraries

- 5 libraries are likely to be encountered
  - Numpy (essentials for scientific computing)
    - Class: numpy.ndarray – a multidimensional container
    - Function: numpy.array() – turns a list into an array
    - Method: numpy.ndarray.max – finds max of an array
    - Etc.
    - Powerful N-dimensional array object (ndarray or array)
    - Linear algebra, Fourier transform, and random number capabilities

Numpy example

```
~$ python
    >>> import numpy as np
    >>> a = np.array([2,3,4])
    >>> a
    array([2, 3, 4])
    >>> print(a)
    [2,3,4]
```

Numpy example

```
~$ python
    >>> import numpy as np
    >>> b = np.array([[1.2, 3.5, 5.1],[1.2, 3.5, 5.2]])
    >>> print(b)
    [[ 1.2  3.5  5.1]
     [ 1.2  3.5  5.2]]
    >>> b.shape
    (2, 3)
```

Numpy quick functionality

array creation

- np.random.randn – normally distributed random numbers
- np.zeros – an array of zeros
- np.arange – sequential array starting at 0 and going to n-1
- np.newaxis – increase dimension of array

simple math

- np.sqrt – square root
- np.exp – exponential function
- np.outer – outer product for vectors

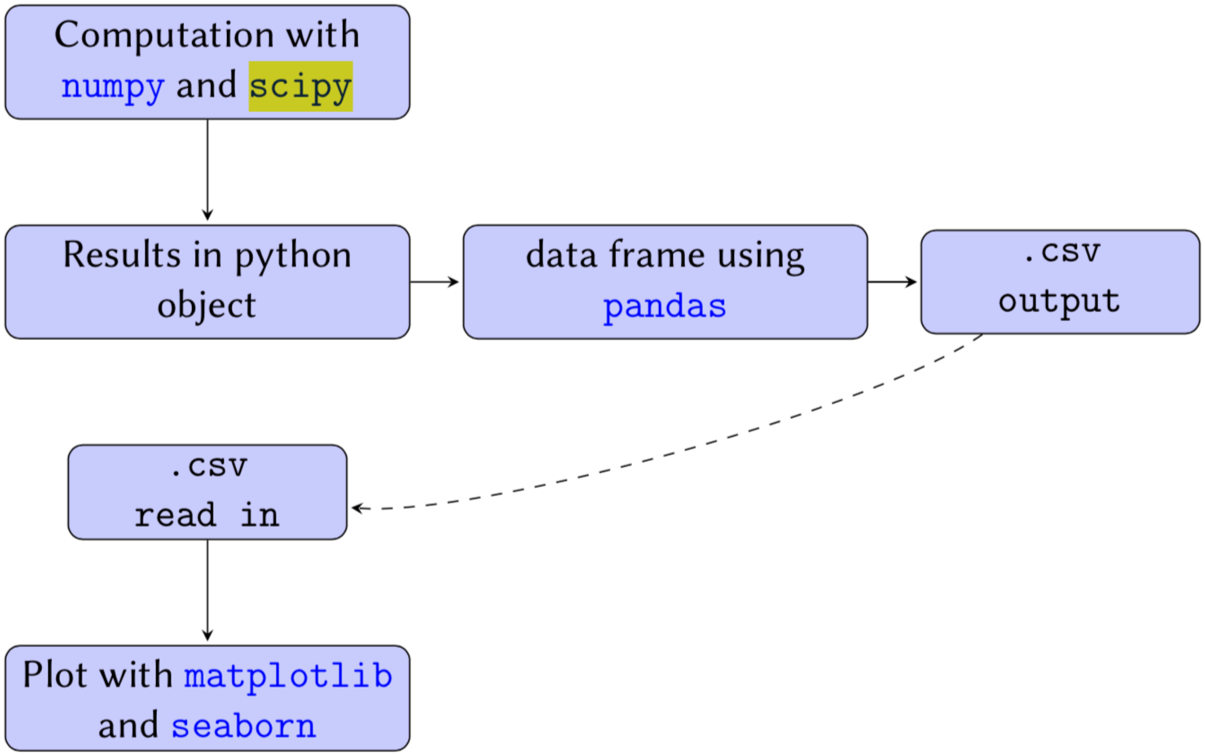
statistics

- np.sum – sum of elements
- np.mean – average
- np.std – standard deviation

Part 1: Basic Libraries

- 5 libraries are likely to be encountered
  - Numpy (essentials for scientific computing)
  - Scipy (augments numpy functionality)
    - More linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and more

Recommended workflow



Part 1: Basic Libraries

- 5 libraries are likely to be encountered
  - Numpy (essentials for scientific computing)
  - Scipy (augments numpy functionality)
  - Pandas (data manipulation)

Pandas example

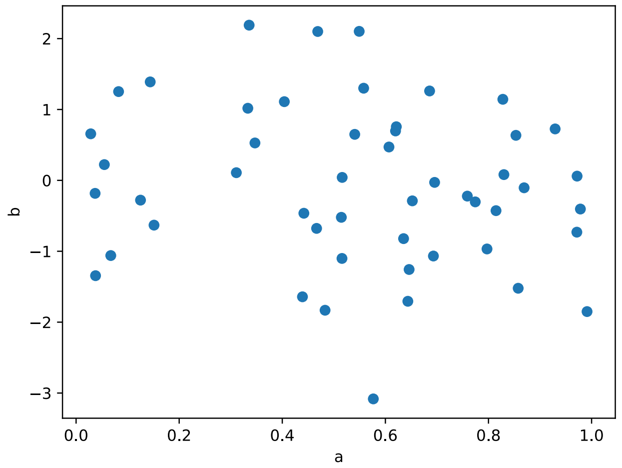
```
~$ python
   >>> import numpy as np
   >>> import pandas as pd
   >>> npts = 50
   >>> df={‘a’:np.random.random(npts),
             ’b’:np.random.random(npts),
             ‘category’:[‘cat1’]*int(npts/2) + [‘cat2’]*int(npts/2) }
   >>> df=pd.DataFrame(df)
```

Part 1: Basic Libraries

- 5 libraries are likely to be encountered
  - Numpy (essentials for scientific computing)
  - Scipy (augments numpy functionality)
  - Pandas (data manipulation)
  - Matplotlib (a 2d plotting library)

Continuing previous example

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.scatter('a','b',data=df)
>>> plt.xlabel('a'); plt.ylabel('b');
>>> plt.savefig("scatter.pdf",bbox_inches='tight')
```



Matplotlib quick functionality

plot creation

- plt.figure – creates a canvas
- plt.plot – produces a plot of bivariate data
- plt.scatter – scatter plot
- plt.subplots – return a subplot axes in a grid

simple formatting

- plt.xlabel and plt.ylabel – set axis labels
- plt.xlim and plt.ylim – set axis limits

Output

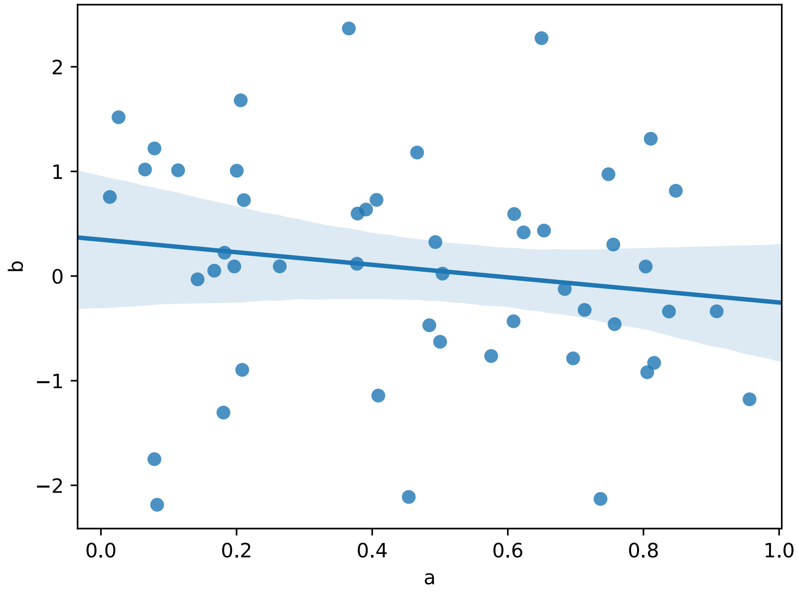
- plt.savefig – save file
- plt.show – see the plots on screen

Part 1: Basic Libraries

- 5 libraries are likely to be encountered
  - Numpy (essentials for scientific computing)
  - Scipy (augments numpy functionality)
  - Pandas (data manipulation)
  - Matplotlib (a 2d plotting library)
  - Seaborn (interfaces with matplotlib for drawing aractive statistical graphics)

Continuing previous example

```
>>> import seaborn as sea
>>> plt.figure()
>>> sea.regplot(x='a', y='b', data=df)
>>> plt.savefig("regression.pdf", bbox_inches='tight')
```



Part 2: Coding in groups

- Exercise 1: Compare exponential evaluation of a random number (interval [1,2]) with a truncated Taylor series
- Exercise 2: Generate list of points normally distributed in 3d around the origin and return a list of distances to the origin
- Exercise 3: Monte Carlo evaluation of π

Exercise 1:Compare exponential evaluation of a random number (interval [1,2]) with a truncated Taylor series

$$\exp(x) \approx \sum_{0}^{N} \frac{1}{n!} x^n$$

- Explicitly import numpy as np, scipy as sp and scipy.special
- Generate a random value with np.random.rand()
- For Taylor series, use sp.special.factorial(i) and np.power(x,i)
- Obtain comparison between exp(x) and Taylor series for 10 orders
- Can use np.sum(array) and np.multiply(array1,array2)

```
import numpy as np
import scipy as sp
import scipy.special
n=10
estimates=np.arange(n,dtype='float64')
x=np.random.random()+1
for i in range(n):
    d=np.arange(i)
    c=1.0/sp.special.factorial(d)
    d=np.power(x,d)
    estimates[i]=np.sum(np.multiply(c,d))

print("x=",x, "Exp(x)=",np.exp(x))
print(estimates)
print(estimates-np.exp(x))
```

Exercise 2: Take an array of positions normally distributed about the origin and return the distance to the origin

- Array of random points is generated for you (note randn instead of random)
- Will need to use numpy.sqrt
- Can take advantage of numpy.sum(x,axis=n)
  - For a given value of n, sums over the n’th indes of the array x
  - In the case of the example, the 0^{th} index is the points, and the 1^{st} index is the positions

```
import numpy as np

npts=10
pos=np.random.randn(npts,3)
#pos holds npts points in 3 dimensions where x,y and z are chosen
#as normally distributed around zero

#want to find the distance form the origin for each point in a new array dist
dist=np.sqrt(np.sum(pos**2,axis=1))

#alternative solution
import scipy as sp
import scipy.linalg
dist2=np.zeros(npts)
for i in np.arange(npts):
    dist2[i]=sp.linalg.norm(pos[i,:])

print(dist)
print(dist2)
```

Exercise 3:Monte Carlo evaluation of pi

- Use the formula $\pi = 4(\frac{area\ of\ unit\ circle}{area\ of\ unit\ square})$
- Find π by randomly sampling points inside a square and counting how many end up inside a circle

```
import numpy as np; import scipy as sp
import scipy.linalg
import scipy.stats
import matplotlib.pyplot as plt

nsamples=18 # number of sample sizes
estimate=np.zeros(nsamples)*1.0 # data arrays
error=np.zeros(nsamples)*1.0

for j in np.arange(nsamples)+1:
    npts=np.power(2,j) # double sampling points each time
    acircle=np.zeros(npts)
    pts=np.random.rand(npts,2)*2.0-1.0 # generate points inside unit square
    for i in range(len(pts)):
        if scipy.linalg.norm(pts[i]) < 1.0: # points inside the unit circle
            acircle[i]=1
    estimate[j-1]=np.mean(acircle)*4.0
    error[j-1]=sp.stats.sem(acircle)*4.0
    print (estimate[j-1], "+/-", error[j-1])
```