Unit 2
Numpy, Pandas, Matplotlib
(Execute "Numpy, Pandas, and Matplotlib.ipynb" along)

IST 718 – Big Data Analytics

Daniel E. Acuna

http://acuna.io

The Scientific Python Ecosystem

- The following commonly-used Python modules are a part of an ecosystem referred to as ***SciPy***:

NumPy
Base N-dimensional array package

SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2D Plotting

IP[y]:
IPython

IPython
Enhanced Interactive Console

Sympy
Symbolic mathematics

pandas
Data structures & analysis

Python Modules and Packages

- A ***module*** is single Python file that is intended to be imported into other Python scripts.
  - The Python Standard Library is the standard foundation of the language and does not need to be imported into a Python script.
  - Other reusable code is imported as modules.

- A ***package*** is a collection of Python modules under a common namespace.
  - Think of packages as folders, and modules as files in a folder.

Numpy

- Vanilla Python lists do not do some of the standard linear algebra

In [1]:
```
A = [1, 2, 3, 4]
4*A
```

Out[1]:
```
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Numpy (2)

- Numpy stores multi-dimensional arrays and allows to make linear computations very efficiently

In [2]:
```
import numpy as np
```

Creating arrays

- You can create arrays in two ways
  - From a Python list or tuple
  - From a special matrix generator
  - Other libraries generate Numpy arrays

In [4]:
```
print("From vanilla Python", np.array([[1,2,3,4]]))
print("From generators", np.zeros((3, 3)))
```

```
From vanilla Python [[1 2
3 4]]
From generators [[0. 0.
0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

ND-arrays store only one data type

- For efficiency reasons, an array only stores one datatype

In [5]:
```
A0 = np.array([0, 2, "cat"])
print(A0)
print(A0.dtype)
```

```
['0' '2'
'cat']
<U21
```

Accessing elements

- You access individual elements with `A[i]` syntax
- Slices with `A[start:stop:step]`
- In multiple dimensions: `A[i, j]` or `A[start:stop:step, start:stop:step]`

In [6]:

```
# some examples
A1 = np.array([1, 2, 3, 4])
A2 = np.array([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]])
```

In [7]:

```
print('A1[0] => ', A1[0])
print('A1[0:3:2] =>', A1[0:3:2])
```

```
A1[0] =>  1
A1[0:3:2] =>
[1 3]
```

In [8]:

```
print('A2[0] =>', A2[0])
print('A2[0:3:2] =>', A2[0:3:2])
```

```
A2[0] => [1 2 3]
A2[0:3:2] => [[1
2 3]
 [7 8 9]]
```

In [10]:

```
print('A2[0:3:2, 0:2] =>', A2[:, 0:2])
```

```
A2[0:3:2, 0:2] =>
[[1 2]
 [4 5]
 [7 8]]
```

Matrix operations

- Standard linear algebra operations work in Numpy
- Scalar times matrix
- Matrix times matrix
- Operations such as inverse, transpose

In [11]:

```python
# some operations are in another package
import numpy.linalg as la

A = np.array([[2, 3],
              [3, 4],
              [5, 6]])
B = np.array([[1, 2, 3],
              [4, 5, 6]])
C = np.array([[1, 3],
              [3, 2],
              [-1, 6]])
```

In [13]:

```python
# Matrix transpose
print(A.shape)
print(A.T)
```

```
(3,
2)
[[2
3 5]
 [3
4
6]]
```

In [14]:

```python
2*A
```

Out[14]:

```
array([[ 4,
6],
       [ 6,
8],
       [10, 1
2]])
```

In [15]:
```
4 + A
```

Out[15]:
```
array([[ 6,
7],
       [ 7,
8],
       [ 9, 1
0]])
```

In [16]:

```
# multiplication
D = A.dot(B)
A.dot(B)
```

Out[16]:

```
array([[14, 19, 2
4],
       [19, 26, 3
3],
       [29, 40, 5
1]])
```

In [17]:
```
# to the proper division we need to do A*C^-1
A.dot(np.invert(B))
```

Out[17]:
```
array([[-19, -24, -2
9],
       [-26, -33, -4
0],
       [-40, -51, -6
2]])
```

In [20]:
```
# checking some identities it should be similar to D
D.dot(D.inv())
```

```
---------------------------------------------------------------------------
-----
AttributeError                            Traceback (most recent call
 last)
<ipython-input-20-d9dd1ec4e874> in <module>
      1 # checking some identities it should be similar to D
----> 2 D.dot(D.inv())

AttributeError: 'numpy.ndarray' object has no attribute 'inv'
```

Random number generation

- Sometimes it is important to generate random numbers to do simulations

In [21]:
```python
# there are many random number generators
list(filter(lambda x: '_' not in x, dir(np.random)))
```

Out[21]:
```
['Lock',
 'RandomStat
e',
 'beta',
 'binomial',
 'bytes',
 'chisquare',
 'choice',
 'dirichlet',
 'division',
 'exponentia
l',
 'f',
 'gamma',
 'geometric',
 'gumbel',
 'hypergeometr
ic',
 'info',
 'laplace',
 'logistic',
 'lognormal',
 'logseries',
 'mtrand',
 'multinomia
l',
 'normal',
 'np',
 'operator',
 'pareto',
 'permutatio
n',
 'poisson',
 'power',
```

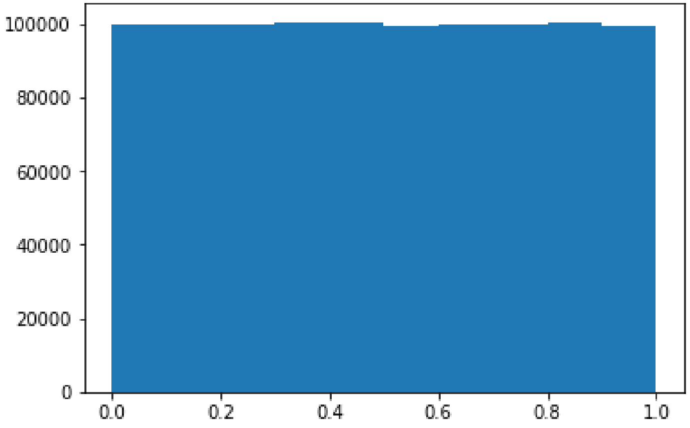
In [22]:
```
# generate 5 by 5 matrix with uniform numbers from 0 to 1
np.random.random(size=(5, 5))
```

Out[22]:
```
array([[0.19708972, 0.89209304, 0.95379111, 0.26793235, 0.3001178
6],
       [0.02543531, 0.89079478, 0.13111967, 0.36656423, 0.0136048
3],
       [0.37863567, 0.57496413, 0.74735069, 0.8001178 , 0.4829663
6],
       [0.64574183, 0.57900522, 0.11668   , 0.61869067, 0.8173352
4],
       [0.00972401, 0.67398739, 0.91426178, 0.3889263 , 0.7937894
3]])
```

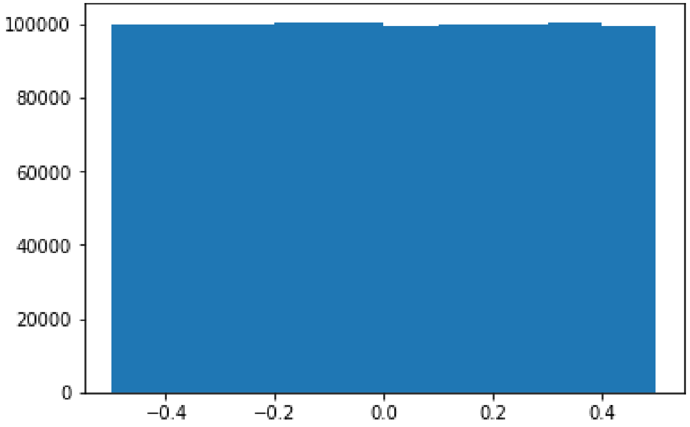
In [23]:
```
import matplotlib.pyplot as plt
```

In [26]:
```
x = np.random.random(1000000)
y = x - 1/2
z = y**2
```

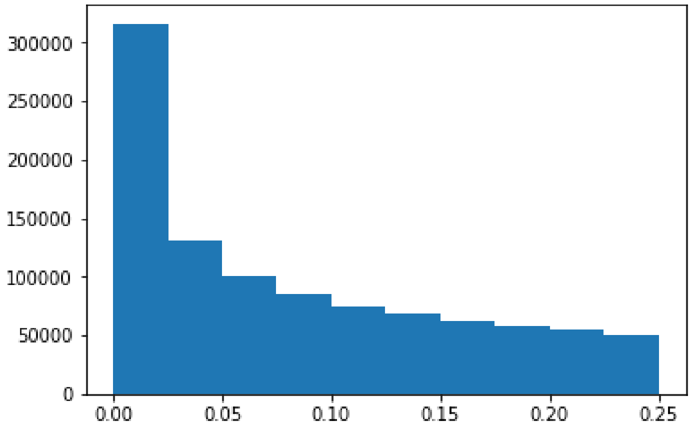
In [27]:
```
plt.hist(x);
```



In [29]:
```
plt.hist(y);
```



In [30]:
```
plt.hist(z);
```



In [ ]:
```
# generate 5 by 5 matrix with normal (Gaussian) distribution
# mean 5 and standard deviation 2.
np.random.normal(size=(5, 5), loc=5., scale=2.)
```

Aggregate operations

- There could be several summary operations that we could do across one dimension or several dimensions
- For example, the average per row or column

In [31]:
```
A3 = np.random.random(size=(3, 5))
```

In [32]:
```
# mean across all dimension
A3.mean()
```

Out[32]:
```
0.607608313389
5238
```

In [33]:
```
# mean across rows (dimension 0)
A3.mean(axis=0)
```

Out[33]:
```
array([0.54012639, 0.73519846, 0.63335786, 0.6085694 , 0.520789
45])
```

In [34]:
```
# mean across columns (dimension 1)
A3.mean(axis=1)
```

Out[34]:
```
array([0.63337467, 0.57860971, 0.610840
57])
```

In [35]:
```python
# there are many such operations
list(filter(lambda x: '_' not in x, dir(A3)))
```

Out[35]:
```
['T',
 'all',
 'any',
 'argmax',
 'argmin',
 'argpartiti
on',
 'argsort',
 'astype',
 'base',
 'byteswap',
 'choose',
 'clip',
 'compress',
 'conj',
 'conjugat
e',
 'copy',
 'ctypes',
 'cumprod',
 'cumsum',
 'data',
 'diagonal',
 'dot',
 'dtype',
 'dump',
 'dumps',
 'fill',
 'flags',
 'flat',
 'flatten',
 'getfield',
 'imag',
```

```
 'item',
 'itemset',
 'itemsize',
 'max',
 'mean',
 'min',
 'nbytes',
 'ndim',
 'newbyteord
er',
 'nonzero',
 'partitio
n',
 'prod',
 'ptp',
 'put',
 'ravel',
 'real',
 'repeat',
 'reshape',
 'resize',
 'round',
 'searchsort
ed',
 'setfield',
 'setflags',
 'shape',
 'size',
 'sort',
 'squeeze',
 'std',
 'strides',
 'sum',
 'swapaxes',
 'take',
 'tobytes',
 'tofile',
 'tolist',
```

```
 'tostring',
 'trace',
 'transpos
e',
 'var',
 'view']
```

Chaining operations

- Because each Numpy operations returns an array, we can easily chain operations

In [36]:
```
A4 = np.random.random(size=(3, 5))
```

In [37]:
```
A4.T.mean(axis=0).sum()
```

Out[37]:
```
1.787325297515
4903
```

Selecting elements with ”masks”

In [38]:
```
# we can create boolean masks
A4 > 0.5
```

Out[38]:
```
array([[False, False,  True, False, Fals
e],
       [ True, False, False,  True, Fals
e],
       [ True,  True,  True,  True,  Tru
e]])
```

In [39]:
```
# and then use those broadcast operations to get the values
A4[A4>0.5]
```

Out[39]:
```
array([0.74422763, 0.59494737, 0.80191027, 0.98855342, 0.94314
084,
       0.63716751, 0.8574581 , 0.9332497 ])
```

In [40]:
```python
# you can also select entire rows using the same idea
print(A4.sum(axis=1))
print(A4.sum(axis=1)>2.5)
```

```
[2.16763326 2.40942365 4.3595
6957]
[False False  True]
```

In [42]:
```python
print(A4.shape)
A4.sum(axis=1)>2.5
```

```
(3

5)
```

Out[42]:
```
array([False, False,  Tr
ue])
```

In [43]:
```python
# select rows
A4[A4.sum(axis=1)>2.5]
```

Out[43]:
```
array([[0.98855342, 0.94314084, 0.63716751, 0.8574581 , 0.9332497
]])
```

In [44]:
```python
A4[:, A4.sum(axis=0) > 2]
```

Out[44]:
```
array([[0.41837726, 0.4083544
9],
       [0.59494737, 0.8019102
7],
       [0.98855342, 0.8574581
]])
```

Operation broadcasting

- Sometimes we might want to apply an operation to each row (or dimension)
- For example, subtract the mean row across a matrix

$$A_{\text{centered}} = (a_{ij} - \sum_z a_{zj})_{ij}$$

- Or "standardize" columns

$$A_{\text{standardized}} = (\frac{a_{ij} - \sum_z a_{zj}}{\text{std}(a_{:j})})_{ij}$$

In [46]:
```
A5 = np.random.random(size=(10, 3))
print(A5)
```

```
[[0.619677   0.9195601  0.18886
623]
 [0.43514718 0.24511337 0.62454
57 ]
 [0.49923623 0.31229564 0.51049
018]
 [0.00556886 0.17766314 0.21766
184]
 [0.4876964  0.92616645 0.94835
705]
 [0.82362688 0.26083034 0.75379
557]
 [0.65414665 0.9352082  0.15394
135]
 [0.21128807 0.69399779 0.43452
52 ]
 [0.03648994 0.40582307 0.84703
833]
 [0.15991117 0.79577689 0.25394
839]]
```

In [49]:

```python
# centered
# print(A5.mean(axis=0))
A5 - A5.mean(axis=0)
```

Out[49]:

```
array([[ 0.22639816,  0.3523166 , -0.3044507
5],
       [ 0.04186834, -0.32213013,  0.1312287
2],
       [ 0.10595739, -0.25494786,  0.0171731
9],
       [-0.38770998, -0.38958036, -0.2756551
5],
       [ 0.09441757,  0.35892295,  0.4550400
7],
       [ 0.43034804, -0.30641316,  0.2604785
9],
       [ 0.26086782,  0.3679647 , -0.3393756
3],
       [-0.18199077,  0.12675429, -0.0587917
8],
       [-0.3567889 , -0.16142043,  0.3537213
5],
       [-0.23336767,  0.22853339, -0.2393686
]])
```

Apply other functions to arrays

There are many functions that you can apply to arrays. In fact all functions that look convenient are a wrap to more explicit functions. For example, `A + 3` is a wrap around `np.add(A, 3)`

In [ ]:
```
A + 3
```

In [ ]:
```
np.add(A, 3)
```

In [51]:
```
np.sin(A)
```

Out[51]:
```
array([[ 0.90929743,  0.1411200
1],
       [ 0.14112001, -0.7568025
],
       [-0.95892427, -0.2794155
]])
```

In [52]:
```
# exponential

np.exp(A)
```

Out[52]:
```
array([[  7.3890561 ,  20.0855369
2],
       [ 20.08553692,  54.5981500
3],
       [148.4131591 , 403.4287934
9]])
```

Activity: Linear regression

- Linear regression is a simple linear prediction method
- $age = (30\ 20\ 33\ 25\ 50)^T$
- $income = (25000\ 22000\ 21000\ 27000\ 40000)$
- Let's assume a simple model

$$\hat{income} = b_0 + b_1 age$$

- Use $b = (20000\ 5000)^T$
- Define the matrices X, y, and b for the predictions Xb

In [ ]:

```python
# define X
X = np.array([
    [1, 30],
    [1, 20],
    [1, 33],
    [1, 25],
    [1, 50]
])
b = np.array([
    200000,
    50000
])

# lets define simple model for making predictions
# define X, y, and b
```

In [ ]:

```python
X.dot(b)
```

In [ ]:

```python
type(X.dot(b))
```

In [ ]:

```python
income_hat = X.dot(b).reshape(-1, 1)
print(income_hat)
```

In [ ]:

```python
income_hat.shape
```

π estimation

In [53]:
```
xy = np.random.random(size=(1000000, 2))
```

In [60]:
```
d2 = (xy**2).sum(axis=1, keepdims=True)
```

In [62]:
```
p = (d2 < 1).sum()/d2.shape[0]
```

In [65]:
```
len(d2)
```

Out[65]:
```
100
000
0
```

In [63]:
```
pi = 4*p
```

In [64]:
```
pi
```

Out[64]:
```
3.14
1888
```

In [ ]:
```
#(d2 < 1).mean()
```

Activity: Compute the Mean Squared Error

- Sometimes, we want to compute the mean squared error of a model

$$MSE(b) = \frac{1}{n} \sum_{i=1}^{n} (\hat{income}_i - income_i)^2$$

- Write a function `mse` that takes X, b, and y, and computes MSE

```
In [ ]: def mse(X, b, y):
            pass
```

Activity: Computing the gradient of MSE

$$\Delta MSE = \left(\frac{dMSE(b_0)}{db_0} \quad \frac{dMSE(b_1)}{db_1}\right)^T$$

- Define a function `grad` that takes X and b and returns the gradient of MSE

In [ ]:

```
def grad(X, b):
    pass
```

Activity: Gradient descent

- The following algorithm finds the b that minimizes MSE

```
b = random vector
for i in [1, ..., n]:
    b = b - L grad(X, b)
```

- where L is known as the learning rate.
- Display the `mse` after each iteration. The `mse` should decrease after each iteration

Pandas

- One of the problems with numpy arrays is that the columns and rows do not have names
- Also Numpy arrays can hold only one dataset
- Sometimes, we want to store something like a "spreadsheet" with names for columns and different datatypes

What is pandas?

- ***pandas*** is an open-source library with easy-to-use data structures and functions that simplifies data analysis and modeling in Python, including:
    - `DataFrame` and `Series` data structures
    - tools for reading and writing data in CSV format, Excel, and SQL databases
    - "group by" operator on data sets
    - merging and joining data sets

- pandas data structures are used in many other Python libraries, so it is a good library to be familiar with.

Using Pandas in Your Programs

- We'll need to import some packages and modules to use Pandas

```
import pandas as pd
import numpy as np
from pandas import DataFrame, Series
```

Essential Pandas: Series and DataFrames

- Pandas has two data structures
  - `Series` → A Labeled list of data,
  - `DataFrame` → A dictionary of `Series`
- The `DataFrame` is a table of data, and the `Series` represents one column in that table.
- NOTE: Pandas is “smart enough” to create a `DataFrame` from a list of dictionary, too.

Demo: Exploring Pandas DataFrame and Series

Demo: Pandas Data Manipulations

Row and Column Extractions

Loading From a File to Pandas is Easy

- Use the `read_csv` pandas method to load data.
- It assumes first row is a header, but it can be manually overridden.
- http://pandas.pydata.org/pandas-docs/stable/io.html (http://pandas.pydata.org/pandas-docs/stable/io.html)

Demo: Exploring a data set in pandas

matplotlib

- A 2D and 3D plotting library that can be used by Python as well as other frameworks.
- Has a large API with functions that can graph just about any plot imaginable.