Northeastern University

INFO 6105
Data Sci Eng Mth & Tools
Lecture 2 Introduction to NumPy

9 January 2019

NumPy

Part 1

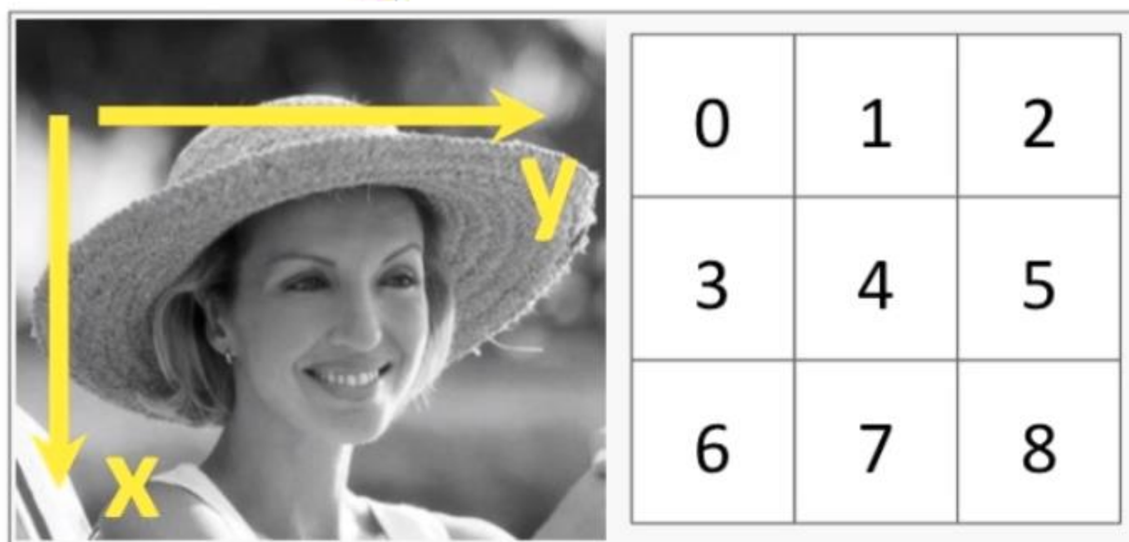
INTRODUCTION TO NumPy & SOME LINEAR ALGEBRA

NumPy

- **In this class, we learn and use basic tools for Data Science.**
  - **These consist of theories in statistics and probability and linear algebra, in the 4 basic Data Science libraries written for Python: NumPy, Pandas, SciPy, and Scikit-learn**
- **Numpy adds Python support for large multi-dimensional arrays and matrices, along with a library of high-level mathematical functions to operate on these arrays**
- **Numpy is the first and lowest level data science extension for Python**
  - **It focuses on number calculations, reads in fixed datatypes, improves RAM efficiency, and teaches you to think in Vectors**
  - **But you already think in vectors and matrices from your R homework!**

NumPy

- Package for scientific computing in Python
  - Multidimensional and larger arrays
- Includes:
  - Data structures
  - Array routines
    - Shape management
    - Sorting
    - Linear algebra
    - Statistical
  - High performance functions



NumPy Introduction

- `import numpy as np`
- `import matplotlib.pyplot as plt`
- `x = np.linspace(0,5,300)`
- `#x = np.arrange(0,5,0.015)`
- `y = np.cos(x)`
- `myplot = plt.plot(x,y)`
- ```
  plt.xlabel('x’)
  plt.ylabel('cos(x)’)
  plt.plot(x,y)
  ```


  ```

NumPy Introduction (continued)

```
points = np.arrange(-5, 5, 0.01)
x, y = np.meshgrid(points, points)
z = np.sqrt(x ** 2, y ** 2)
import matplotlib as plt
plt.imshow(z, cmap = plt.cm.gray]; plt.colorbar()
plt.title("image plot of $\sqrt[x^2 + y^2}$")
plt.show()
```

NumPy Ndarray Objects

- Python
  - Arrays can grow dynamically
- NumPy
  - Arrays have *fixed* size at creation
  - Elements of *same* type
  - Advanced mathematical operations
  - Multidimensional array set, faster and more efficient than basic Python package
  - Stores data in contiguous blocks of memory, NumPy algorithms written in C, with no type checking overhead
  - Default data types:
    - `bool_` #True or False
    - `int_` #long
    - `uint8` #0 to 255
    - `float16` #half-precision fp
    - `complex` #2 32//64/128 floats
    - `object` #std python object

NumPy arrays

- `help(‘numpy’)`
- `import numpy as np`
- `np.array(‘1,2,3,4')`
- `np.array(np.mat(’1,2; 3,4’))`
- `array2 = np.array([[2,4,6,8], [3,6,9,12], [4,8,12,16]])`
- `type(array2)`
- `array2.ndim`
- `array2.shape`
- `array2.dtype`
- `array2.itemsize`
- `array2.data`

Array generation

- `arange(0, 10, 0.1) # arguments: start, stop, step`
- `linspace(0, 10, 25) # using linspace, both end points ARE included`
- `logspace(0, 10, 10, base=e)`
- `x, y = mgrid[0:5, 0:5]`
- `random.rand(5,5) # uniform random numbers in [0,1]`
- `random.randn(5,5) # standard normal distributed random numbers`
- `zeros((3,3)); ones((3,3))`
- `A = array([[n+m*10 for n in range(5)] for m in range(5)]) #list comprehensions`

Saving to a file

- `save("random-matrix.npy", M)`
- `!file random-matrix.npy`
- `random-matrix.npy: data`
- `load("random-matrix.npy")`

Performance on array of a million integers

- `import numpy as np`
  `myarr = np.arange(1000000)`
  `mylist = list.range(1000000))`
- **Multiply by 2:**
  - `%time for _ in range(10): myarr2 = myarr * 2`
  - `%time for _ in range(10): mylist2 = [x * 2 for x in mylist]`
- **Order of magnitude more performing, and also use less memory!**

Array operations

- **Does this remind you of our R labs?**
- `import numpy as np`
- `data = np.random.randn(4,4)`
- `data`
- `data * 10`
- `arr = np.array[[1,2,3], [4,5,6], [7,8,9]]`
- `arr * arr`
- `np.sqrt(arr)`

Array slicing

```
arr = np.arange(10)
arr
arr[5:8]
arr[5:8] = 10
slice = arr[:8]
slice[1] = 0
arr #remember: no copying!
```

High dimensional operations

- `np.empty((2,3,4))`
- `arr = np.array([[[1,2,3], [4,5,6], [7,8,9]], [10,11,12]]])`
- `arr[0,2]`
- `arr[0][2]`

Boolean indexing

- `names = np.array('bob', 'joe','amy', 'bob','sue']`
- `names`
- `names == 'bob'`
- `data = np.random.randn(7,4)`
- `data`
- `data[names == 'bob']`

Array reshaping, transposing, & iterating

- `arr = np.arange(32).reshape((8,4))`
- `arr`
- `arr.T`

- `v = np.array([1,2,3]); np.shape(v)   #(3,)`
- `np.shape(v[:, np.newaxis]) #(3, 1)`
- `np.shape(v[np.newaxis, :]) #(1, 3)`

```
M = array([[1,2], [3,4]])
for row in M:
   print("row", row)

   for element in row:
      print(element)
```

```
M = array([[ 1,  4],
           [ 9, 16]])

if (M > 5).any():
   print("at least one element in M is larger than 5")
else:
   print("no element in M is larger than 5")

if (M > 5).all():
   print("all elements in M are larger than 5")
else:
   print("all elements in M are not larger than 5")
```

Linear algebra with NumPy

```
From numpy.linalg import inv, qr
X = np.random.randn(5, 5)
mat = X.T.dot(X)
inv(mat)
q, r = qr(mat)
r

A = array([[n+m*10 for n in range(5)] for m in range(5)]) #list comprehensions
A * A # element-wise multiplication
np.dot(A, A)   # Matrix multiplication
M = np.matrix(A); M * M # Matrix multiplication
M.T
np.linalg.inv(M) # Matrix inverse
np.linalg.det(M) # determinant
```

Vectorizing for performance

- To get good performance we should try to avoid looping over elements in our vectors and matrices, and instead use vectorized algorithms
  - The first step in converting a scalar algorithm to a vectorized algorithm is to make sure that the functions we write work with vector inputs

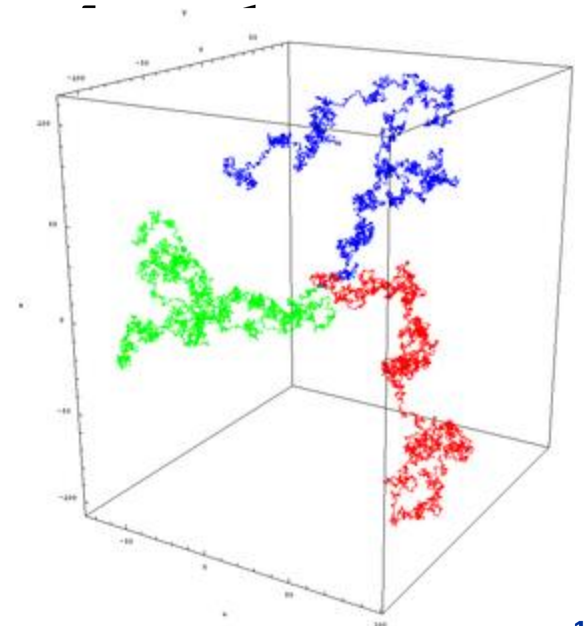
```
def Theta(x):
    """
    Scalar implemenation of the Heaviside step function.
    """
    if x >= 0:
        return 1
    else:
        return 0
```

- **`Theta(array([-3,-2,-1,0,1,2,3]))`**
- **`Theta_vec = vectorize(Theta)`**
- **`Theta_vec(array([-3,-2,-1,0,1,2,3]))`**
- **or…**

```
def Theta(x):
    """
    Vector-aware implemenation of the Heaviside step function.
    """
    return 1 * (x >= 0)
```

Statistics with NumPy: Random Walks

- **Reference @ https://en.Wikipedia.org/wiki/Random_walk**
- `import random`
- `position = 0`
- `walk = [position]`
- `steps = 1000`
- ```
  for i in range(steps):
     step = 1 if random.randint(0,1)
     position += step
     walk.append(position)
  ```
- `plt.plot(walk[:100])`


  ```

Statistics: Cumulative sum

- `Walk` **is simply the cumulative sum of the random steps**
  - The **cumulative sum** is not the **cumulative sum** of the values. Instead it is the **cumulative sum** of differences between the values and the average. Because the average is subtracted from each value, the **cumulative sum** also ends at zero
  - **https://en.wikipedia.org/wiki/CUSUM**
  - **An important statistical measure that is used in machine learning for anomaly detection**
- `nsteps = 1000`
- `draws = np.random.randint(0, 2, size = nsteps)`
- `steps = np.where(draws > 0, 1, -1)`
- `walks = steps.cumsum()`
- **Compute first crossing time: Step at which the random walk reaches a particular value**
  - `(np.abs(walk) >= 10).argmax()`

NumPy and pandas

- **Although `NumPy` is the basis for *vectorized computing* in Python (what you've been doing in labs with R), `pandas` will be the library that will give us better higher-level tools to manipulate lots of data**


CLASS
DISMISSED
iStockphoto