Numeric Python (NumPy) and Scientific Python (SciPy)

Arrays and Math

While we know that arrays and lists exist in Python, we have not yet tried to do any real computation with those tools.

Let's get started by doing some simple computations:

- Calculate the dot product of two vectors of equal length

Dot Product

The **dot product** of two vectors is the sum of the products of the corresponding elements in each vector. We can write it as follows:

$$a \cdot b = \sum_{i=1}^{N} a_i \times b_i$$

Take 5 minutes, and write a function (or at least some pseudocode) to take two vectors of any (equal) length, and calculate the dot product.

Dot Product

```
def dotProd(v1, v2): # Define our function and arguments
  if len(v1) is len(v2): # Test equality of vector length
    dp = 0               # Initialize Dot Product Value
    for i in range(len(v1)): # Loop over all elements
      dp += v1[i]*v2[i]      # Add elements of the sum
    return dp            # Return the dot product
  else: # If vetors are of unequal length, return error
    raise RuntimeError("Vectors must be of equal length")
```

This code allows us to calculate the dot product.

Matrix Multiplication

We can multiply two matrices when the number of columns in the first matrix are equal to the number of rows in the second matrix:

- A is an $m \times n$ matrix (has m rows, n columns)
- B is an $n \times q$ matrix (has n rows, q columns)

In this case, the matrices are conforming, and can be multiplied together

- The resulting matrix, C is an $m \times q$ matrix

Matrix Multiplication

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 6 \\ 5 \end{bmatrix}$$

Matrix A has 2 columns, and B has 2 rows (conforming).

$$AB = C = \begin{bmatrix} 16 \\ 38 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \cdot 6 + 2 \cdot 5 \\ 3 \cdot 6 + 4 \cdot 5 \end{bmatrix}$$

Matrix Multiplication

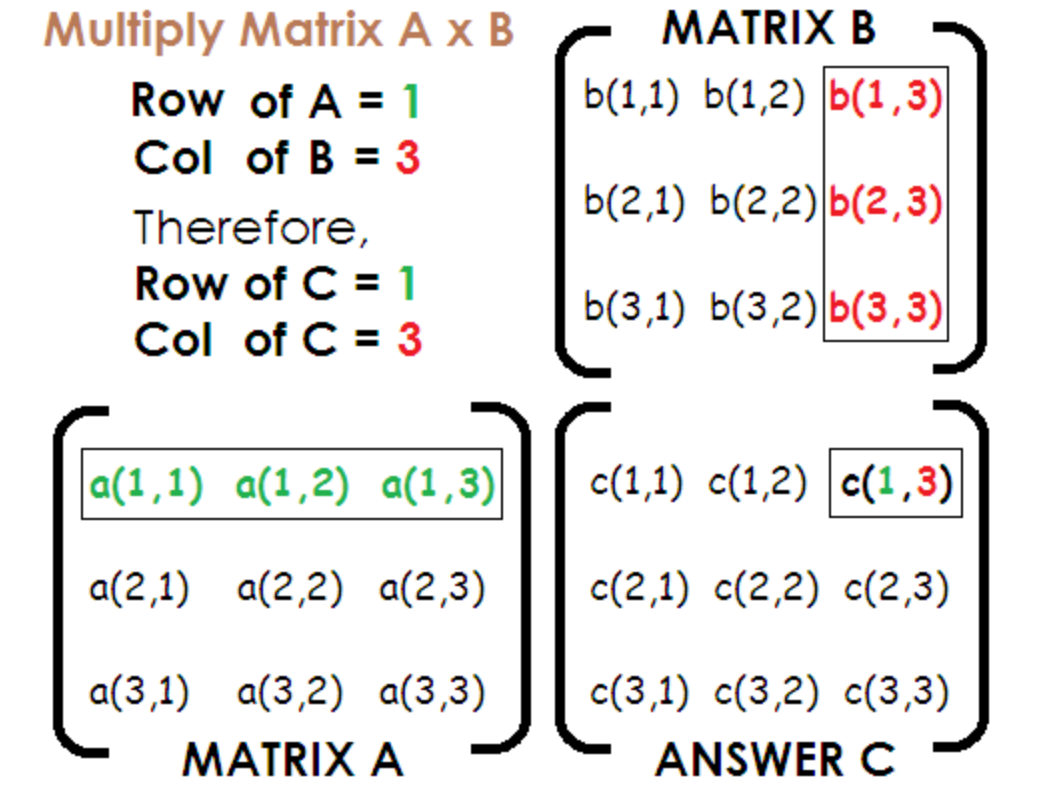
$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix}, B = \begin{bmatrix} b_{1,1} & \cdots & b_{1,p} \\ \vdots & \ddots & \vdots \\ b_{m,1} & \cdots & b_{m,p} \end{bmatrix}$$

Matrix A has m columns, and B has m rows (conforming). C will have shape $n \times p$

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \times b_{kj}$$

We calculate all elements of C in this manner

Matrix Multiplication



$$c(1,3) = [a(1,\underline{1})*b(\underline{1},3)] + [a(1,\underline{2})*b(\underline{2},3)] + [a(1,\underline{3})*b(\underline{3},3)]$$

Matrix Multiplication

Matrix A has m columns, and B has m rows (conforming). C will have shape $n \times p$

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \times b_{kj}$$

Write an algorithm (or at least some pseudocode, again) to perform arbitrary matrix multiplication given two conforming matrices.

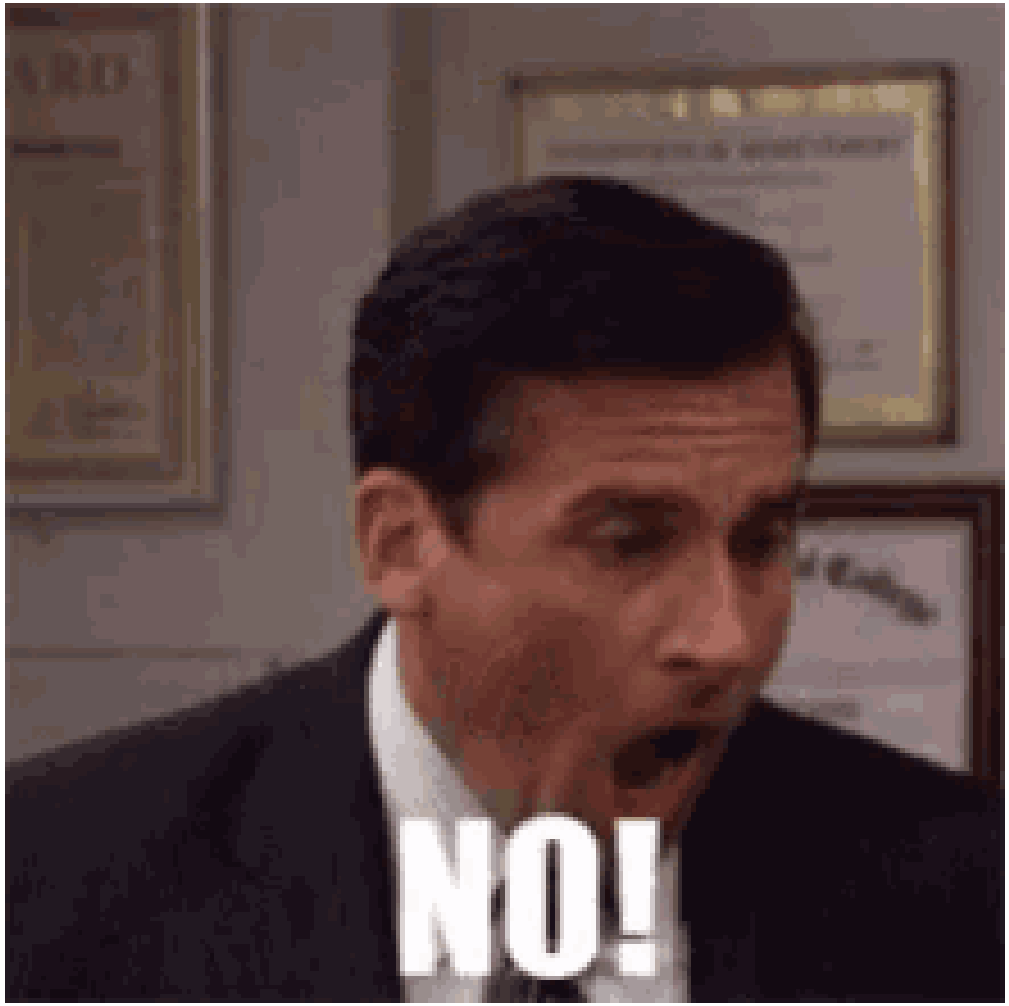
Matrix Multiplication

```
def matMul(a, b): # Define function, take 2 matrices
  for i in range(len(a)): # Make sure a is matrix
    if len(a[i]) is not len(a[0]): # If not,
      raise RuntimeError( # Raise an error
        "Matrix A is not correctly specified")
  for i in range(len(b)): # Make sure b is a matrix
    if len(b[i]) is not len(b[0]): # If not,
      raise RuntimeError( # Raise an error
        "Matrix B is not correctly specified")
  matrix = [] # Initialize new empty matrix
  if len(a[0]) is len(b): # Test for conformability
    for i in range(len(a)): # Iterate over rows of a
      row = [] # Create row of new matrix
      for j in range(len(b[0])): # Iterate over columns
        row.append(dotProd(a[i], # Append elements of col
          [b[n][j] for n in range(len(b))]))
      matrix.append(row) # Append rows to matrix
    return matrix # Return the newly calculated matrix
  else: # If matrices are nonconforming
    raise RuntimeError( #Raise an error
      "Matrices are nonconformable for multiplication")
```

Computations and Python

It is great that Python is so flexible that we can quickly write functions to do calculations like matrix multiplication.

- Do we WANT to write out functions to do all of the mathematical processes we need for different kinds of analysis?
  - Random number generators?
  - Matrix inversion algorithms?
  - Solving matrix equalities?


NO!

Numeric Python (Numpy)

Instead of writing our own algorithms, sometimes we prefer libraries with pre-written (and far more efficient) algorithms to solve complex mathematical problems.

The `numpy` library is the principal library for mathematical computation in Python.

Numpy Reference Page

Numpy - Arrays

The building blocks of `numpy` are arrays. Arrays can essentially be treated as equivalent to mathematical matrices, and are a special object type that takes data and stores it in formats that allow us to more easily apply mathematical functions to that data.

We will focus on creating arrays in two ways:

1. Creating arrays by coercing lists and tuples to the array type
2. Creating arrays using `numpy` commands

Numpy Arrays - List Coercion

```
>>> import numpy as np # import library as np object

>>> myList = [1, 2, 3, 4]

>>> myArray = np.array(myList)
>>> myArray
array([1, 2, 3, 4])
```

We can use the `np.array` function to generate an array from any arbitrary list (or tuple) of numbers

Numpy Arrays - Using Commands

```
>>> np.array([1,2,3,4]) # Specify each element
array([1, 2, 3, 4])
>>> np.zeros((3,3)) # Generate 3 x 3 array of zeros
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.eye(3) # Generate 3 x 3 identity matrix
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

The `array`, `zeros`, and `eye` functions are all ways to create arrays in `numpy`. Many other functions exist to generate arrays that we won't cover.

Numpy Arrays - Manipulation

```
>>> myArray = np.array([1,2,3,4]) # Specify each element
>>> np.shape(myArray)
(4,) # A vector with 4 elements
>>> myArray = np.array([[1,2,3,4]])
>>> np.shape(myArray)
(1, 4)
>>> myArray.reshape((2,2)) # transform to square matrix
array([[1, 2],
       [3, 4]])
>>> myArray.reshape((4,1)) # transform to column vector
array([[1],
       [2],
       [3],
       [4]])
>>> myArray = np.array([[1,2,3,4]])
>>> myArray.T # transposes the array if 2-D
```

Numpy Arrays - Operations

Add a scalar to an array:

```
>>> newArray = np.array(myArray) + 1
array([2, 3, 4, 5])
```

Add (or subtract) arrays:

```
>>> myArray - newArray # must have same shape
array([[-1, -1, -1, -1]])
```

Matrix Multiplication:

```
>>> myArray.T.dot(myArray) # to get 4 x 4 product
>>> myArray.T @ myArray # '@' only works in python >=3.5
```

Exercise

Write a function that accepts four arguments (a, b, c, and x), and calculates the output (y) of the following functional form:

$$y = a + b \cdot x + c \cdot x^2$$

Try to use matrix multiplication to calculate the answer.

Hint: Think about how a dot product might calculate the output of this equation.

Exercise - Answer

```
def squareFunc(a=1, b=1, c=1, x=1):
  coef = np.array([a, b, c])
  xs = np.array([1, x, x**2])
  return coef.dot(xs)
  # OR
  # return coef @ xs
```

Why write this with arrays? Because vectorized math using `numpy` is FAR more efficient computationally. This doesn't matter for our current use case, but is very important when writing large scale code!

Random Numbers

We generate random numbers for all sorts of tasks:

1. Simulations
2. Bootstrapping Procedures
3. Resampling

`numpy` has the functionality to allow us to generate many different and useful types of random numbers

Random Numbers

In order to generate ANY set of random numbers, it is common to start with a random number on the unit interval $[0, 1)$. This is easily done in `numpy`:

```
>>> import numpy as np
>>> np.random.rand() # Generates a single value
0.596137632067727 6
>>> np.random.rand(3) # Array of 3 random values
array([ 0.98936539,  0.82217552,  0.88597465])
```

This function draws from the uniform distribution, and can be utilized as the basis for ANY other random process.

Random Numbers and Pandas!

This random number generation is the basis for the sampling algorithm in `pandas`:

```
sampled_data = full_data_name.sample(10000, replace=False)
```

Exercise - Inverse Transform Sampling

Using the function `np.random.rand`, generate a sample of 10 observations from the *Exponential Distribution*, where $\lambda = 1$.

Hint: Look up the CDF of the Exponential Distribution, solve for x, and use it to generate your values

Exercise - Inverse Transform Sampling

CDF: $F = 1 - e^{-\lambda x} = y \rightarrow x = -ln(1-y)$

```
def expD(x): # Define my function
  if isinstance(x, float) | isinstance(x, int):
      # Test if argument is list or not
    return -1*np.log(1-x) # Return single value
  else: # If list
    return [-1*np.log(1-y) for y in x] # Return values
```

```
>>> expD(np.random.rand(10))
[0.2058003309362563 5, 0.905576715737244 3,
0.042040144990297 02, 0.023403038262802461,
0.365705234289153 14, 1.801576545427130 2,
0.187430938065661 19, 0.060997141650068795,
0.325888286499421 42, 0.547638761306388 5]
```

Distributional Calculations

Fortunately for us (since most distributions involve a BIT more work than the exponential distribution...), `numpy.random` includes functions for sampling from most distributions:

- Normal Distribution
- Poisson Distribution
- Log-normal Distribution
- Binomial Distribution
- Tons more

Distributional Calculations

Even greater statistical functionality is available through the `scipy.stats` module, which provides helper functions for many distributions.

Example: Normal Distribution

Example: Student's t Distribution

We can use this functionality when we build out statistical tests on regression analysis, or in many other cases where we intend to sample from a specific distribution.

Lab Time!