# Introduction to numpy:

### Package for scientific computing with Python

Numerical Python, or "Numpy" for short, is a foundational package on which many of the most common data science packages are built. Numpy provides us with high performance multi-dimensional arrays which we can use as vectors or matrices.

The key features of numpy are:

- ndarrays: n-dimensional arrays of the same data type which are fast and space-efficient. There are a number of built-in methods for ndarrays which allow for rapid processing of data without using loops (e.g., compute the mean).
- Broadcasting: a useful tool which defines implicit behavior between multi-dimensional arrays of different sizes.
- Vectorization: enables numeric operations on ndarrays.
- Input/Output: simplifies reading and writing of data from/to file.

**Additional Recommended Resources:**
Numpy Documentation (https://docs.scipy.org/doc/numpy/reference/)
*Python for Data Analysis* by Wes McKinney
*Python Data science Handbook* by Jake VanderPlas

# Getting started with ndarray

**ndarrays** are time and space-efficient multidimensional arrays at the core of numpy. Like the data structures in Week 2, let's get started by creating ndarrays using the numpy package.

## How to create Rank 1 numpy arrays:

In [3]:
```
import numpy as np

an_array = np.array([3, 33, 333])  # Create a rank 1 array

print(type(an_array))              # The type of an ndarray is: "<class 'nump
y.ndarray'>"
```

```
<class 'numpy.ndarray'>
```

In [4]:
```
# test the shape of the array we just created, it should have just one dimensi
on (Rank 1)
print(an_array.shape)
```

```
(3,)
```

In [5]:
```
# because this is a 1-rank array, we need only one index to accesss each eleme
nt
print(an_array[0], an_array[1], an_array[2])
```

```
3 33 333
```

In [6]:
```
an_array[0] =888            # ndarrays are mutable, here we change an element
 of the array

print(an_array)
```

```
[888  33 333]
```

# How to create a Rank 2 numpy array:

A rank 2 **ndarray** is one with two dimensions. Notice the format below of [ [row] , [row] ]. 2 dimensional arrays are great for representing matrices which are often useful in data science.

In [7]:
```
another = np.array([[11,12,13],[21,22,23]])   # Create a rank 2 array

print(another)  # print the array

print("The shape is 2 rows, 3 columns: ", another.shape)  # rows x columns


print("Accessing elements [0,0], [0,1], and [1,0] of the ndarray: ",
another[0, 0], ", ",another[0, 1],", ", another[1, 0])
```

```
[[11 12 13]
 [21 22 23]]
The shape is 2 rows, 3 columns:  (2, 3)
Accessing elements [0,0], [0,1], and [1,0] of the ndarray:  11 ,  12 ,  21
```

# There are many way to create numpy arrays:

Here we create a number of different size arrays with different shapes and different pre-filled values. numpy has a number of built in methods which help us quickly and easily create multidimensional arrays.

In [31]:
```
import numpy as np

# create a 2x2 array of zeros
ex1 = np.zeros((3,3))
print(ex1)
print("---------------")
print(ex1[:2,])
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
---------------
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

In [9]:
```
# create a 2x2 array filled with 9.0
ex2 = np.full((2,2), 9.0)
print(ex2)
```

```
[[ 9.  9.]
 [ 9.  9.]]
```

In [10]:
```
# create a 2x2 matrix with the diagonal 1s and the others 0
ex3 = np.eye(2,2)
print(ex3)
```

```
[[ 1.  0.]
 [ 0.  1.]]
```

In [11]:
```
# create an array of ones
ex4 = np.ones((1,2))
print(ex4)
```

```
[[ 1.  1.]]
```

In [12]:
```
# notice that the above ndarray (ex4) is actually rank 2, it is a 2x1 array
print(ex4.shape)

# which means we need to use two indexes to access an element
print()
print(ex4[0,1])
```

```
(1, 2)

1.0
```

In [ ]:
```
# create an array of random floats between 0 and 1
ex5 = np.random.random((2,2))
print(ex5)
```

# Array Indexing

## Slice indexing:

Similar to the use of slice indexing with lists and strings, we can use slice indexing to pull out sub-regions of ndarrays.

In [13]:
```
import numpy as np

# Rank 2 array of shape (3, 4)
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
print(an_array)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

Use array slicing to get a subarray consisting of the first 2 rows x 2 columns.

In [19]:
```
a_slice = np.array(an_array[:2, 1:3])
print(a_slice)
```

```
[[1000   13]
 [  22   23]]
```

When you modify a slice, you actually modify the underlying array.

In [21]:
```
print("Before:", an_array[0, 1])   #inspect the element at 0, 1
a_slice[0, 0] = 12    # a_slice[0, 0] is the same piece of data as an_array[0,
 1]
print("After:", an_array[0, 1])
```

```
Before: 1000
After: 1000
```

## Use both integer indexing & slice indexing

We can use combinations of integer indexing and slice indexing to create different shaped matrices.

In [22]:
```
# Create a Rank 2 array of shape (3, 4)
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
print(an_array)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

In [23]:
```
# Using both integer indexing & slicing generates an array of lower rank
row_rank1 = an_array[1, :]    # Rank 1 view

print(row_rank1, row_rank1.shape)  # notice only a single []
```

```
[21 22 23 24] (4,)
```

In [24]:
```
# Slicing alone: generates an array of the same rank as the an_array
row_rank2 = an_array[1:2, :]  # Rank 2 view

print(row_rank2, row_rank2.shape)   # Notice the [[ ]]
```

```
[[21 22 23 24]] (1, 4)
```

In [25]:
```
#We can do the same thing for columns of an array:

print()
col_rank1 = an_array[:, 1]
col_rank2 = an_array[:, 1:2]

print(col_rank1, col_rank1.shape)  # Rank 1
print()
print(col_rank2, col_rank2.shape)  # Rank 2
```

```
[12 22 32] (3,)

[[12]
 [22]
 [32]] (3, 1)
```

# Array Indexing for changing elements:

Sometimes it's useful to use an array of indexes to access or change elements.

In [26]:
```
# Create a new array
an_array = np.array([[11,12,13], [21,22,23], [31,32,33], [41,42,43]])

print('Original Array:')
print(an_array)
```

```
Original Array:
[[11 12 13]
 [21 22 23]
 [31 32 33]
 [41 42 43]]
```

In [27]:
```
# Create an array of indices
col_indices = np.array([0, 1, 2, 0])
print('\nCol indices picked : ', col_indices)

row_indices = np.arange(4)
print('\nRows indices picked : ', row_indices)
```

```
Col indices picked :  [0 1 2 0]

Rows indices picked :  [0 1 2 3]
```

In [28]:
```
# Examine the pairings of row_indices and col_indices.  These are the elements
 we'll change next.
for row,col in zip(row_indices,col_indices):
    print(row, ", ",col)
```

```
0 ,  0
1 ,  1
2 ,  2
3 ,  0
```

In [29]:
```
# Select one element from each row
print('Values in the array at those indices: ',an_array[row_indices, col_indic
es])
```

```
Values in the array at those indices:  [11 22 33 41]
```

In [ ]:
```
# Change one element from each row using the indices selected
an_array[row_indices, col_indices] += 100000

print('\nChanged Array:')
print(an_array)
```

# Boolean Indexing

## Array Indexing for changing elements:

In [ ]:
```
# create a 3x2 array
an_array = np.array([[11,12], [21, 22], [31, 32]])
print(an_array)
```

In [ ]:
```
# create a filter which will be boolean values for whether each element meets
 this condition
filter = (an_array > 15)
filter
```

Notice that the filter is a same size ndarray as an_array which is filled with True for each element whose corresponding element in an_array which is greater than 15 and False for those elements whose value is less than 15.

In [ ]:
```
# we can now select just those elements which meet that criteria
print(an_array[filter])
```

In [ ]:
```
# For short, we could have just used the approach below without the need for t
he separate filter array.

an_array[(an_array % 2 == 0)]
```

What is particularly useful is that we can actually change elements in the array applying a similar logical filter. Let's add 100 to all the even values.

In [ ]:
```
an_array[an_array % 2 == 0] +=100
print(an_array)
```

# Datatypes and Array Operations

## Datatypes:

In [ ]:
```
ex1 = np.array([11, 12]) # Python assigns the  data type
print(ex1.dtype)
```

In [ ]:
```
ex2 = np.array([11.0, 12.0]) # Python assigns the  data type
print(ex2.dtype)
```

In [ ]:
```
ex3 = np.array([11, 21], dtype=np.int64) #You can also tell Python the  data t
ype
print(ex3.dtype)
```

In [ ]:
```
# you can use this to force floats into integers (using floor function)
ex4 = np.array([11.1,12.7], dtype=np.int64)
print(ex4.dtype)
print()
print(ex4)
```

In [ ]:
```
# you can use this to force integers into floats if you anticipate
# the values may change to floats later
ex5 = np.array([11, 21], dtype=np.float64)
print(ex5.dtype)
print()
print(ex5)
```

# Arithmetic Array Operations:

In [ ]:
```
x = np.array([[111,112],[121,122]], dtype=np.int)
y = np.array([[211.1,212.1],[221.1,222.1]], dtype=np.float64)

print(x)
print()
print(y)
```

In [ ]:
```
# add
print(x + y)         # The plus sign works
print()
print(np.add(x, y))  # so does the numpy function "add"
```

In [ ]:
```
# subtract
print(x - y)
print()
print(np.subtract(x, y))
```

In [ ]:
```
# multiply
print(x * y)
print()
print(np.multiply(x, y))
```

In [ ]:
```
# divide
print(x / y)
print()
print(np.divide(x, y))
```

In [ ]:
```
# square root
print(np.sqrt(x))
```

In [ ]:
```
# exponent (e ** x)
print(np.exp(x))
```

# Statistical Methods, Sorting, and Set Operations:

## Basic Statistical Operations:

In [33]:
```
# setup a random 2 x 4 matrix
arr = 10 * np.random.randn(2,5)
print(arr)
```

```
[[-13.29415778   6.58124512 -18.04621171   5.76197528  13.49582377]
 [-29.93674421   2.30901788   0.73468679 -19.10746865  -5.95183744]]
```

In [34]:
```
# compute the mean for all elements
print(arr.mean())
```

```
-5.74536709543
```

In [35]:
```
# compute the means by row
print(arr.mean(axis = 1))
```

```
[ -1.10026506 -10.39046913]
```

In [36]:
```
# compute the means by column
print(arr.mean(axis = 0))
```

```
[-21.615451     4.4451315   -8.65576246  -6.67274668   3.77199316]
```

In [37]:
```
# sum all the elements
print(arr.sum())
```

```
-57.4536709543
```

In [39]:
```
# compute the medians
print(np.median(arr,axis=0))
#important- np.ndarray has no attribute median - call directly from the the np
 library and pass the array as the argument
```

```
[-21.615451     4.4451315   -8.65576246  -6.67274668   3.77199316]
```

## Sorting:

In [40]:
```
# create a 10 element array of randoms
unsorted = np.random.randn(10)

print(unsorted)
```

```
[ 0.06725739  0.32812902  0.79000774  1.43855797 -0.00846992 -1.25954441
  1.01884829  0.62523893  2.59885886  0.23700386]
```

In [41]:
```
# create copy and sort
sorted = np.array(unsorted)
sorted.sort()

print(sorted)
print()
print(unsorted)
```

```
[-1.25954441 -0.00846992  0.06725739  0.23700386  0.32812902  0.62523893
  0.79000774  1.01884829  1.43855797  2.59885886]

[ 0.06725739  0.32812902  0.79000774  1.43855797 -0.00846992 -1.25954441
  1.01884829  0.62523893  2.59885886  0.23700386]
```

In [42]:
```
# inplace sorting
unsorted.sort()

print(unsorted)
```

```
[-1.25954441 -0.00846992  0.06725739  0.23700386  0.32812902  0.62523893
  0.79000774  1.01884829  1.43855797  2.59885886]
```

# Finding Unique elements:

In [43]:
```
array = np.array([1,2,1,4,2,1,4,2])

print(np.unique(array))
```

```
[1 2 4]
```

# Set Operations with np.array data type:

In [ ]:
```
s1 = np.array(['desk','chair','bulb'])
s2 = np.array(['lamp','bulb','chair'])
print(s1, s2)
```

In [ ]:
```
print( np.intersect1d(s1, s2) )
```

In [ ]:
```
print( np.union1d(s1, s2) )
```

In [ ]:
```
print( np.setdiff1d(s1, s2) )# elements in s1 that are not in s2
```

In [ ]:
```
print( np.in1d(s1, s2) )#which element of s1 is also in s2
```

# Broadcasting:

Introduction to broadcasting.
For more details, please see:
https://docs.scipy.org/doc/numpy-1.10.1/user/basics.broadcasting.html (https://docs.scipy.org/doc/numpy-1.10.1/user/basics.broadcasting.html)

In [80]:
```
import numpy as np

start = np.zeros((4,3))
print(start)
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

In [ ]:
```
# create a rank 1 ndarray with 3 values
add_rows = np.array([1, 0, 2])
print(add_rows)
```

In [ ]:
```
y = start + add_rows  # add to each row of 'start' using broadcasting
print(y)
```

In [83]:
```
# create an ndarray which is 4 x 1 to broadcast across columns
add_cols = np.array([[0,1,2,3]])
add_cols = add_cols.T

print(add_cols)
```

```
[[0]
 [1]
 [2]
 [3]]
```

In [84]:
```
# add to each column of 'start' using broadcasting
y = start + add_cols
print(y)
```

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

In [85]:
```
# this will just broadcast in both dimensions
add_scalar = np.array([1])
print(start+add_scalar)
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

Example from the slides:

In [86]:
```
# create our 3x4 matrix
arrA = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(arrA)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In [87]:
```
# create our 4x1 array
arrB = [0,1,0,2]
print(arrB)
```

```
[0, 1, 0, 2]
```

In [88]:
```
# add the two together using broadcasting
print(arrA + arrB)
```

```
[[ 1  3  3  6]
 [ 5  7  7 10]
 [ 9 11 11 14]]
```

# Speedtest: ndarrays vs lists

First setup paramaters for the speed test. We'll be testing time to sum elements in an ndarray versus a list.

In [ ]:
```
from numpy import arange
from timeit import Timer

size    = 1000000
timeits = 1000
```

In [ ]:
```
# create the ndarray with values 0,1,2...,size-1
nd_array = arange(size)
print( type(nd_array) )
```

In [ ]:
```
# timer expects the operation as a parameter,
# here we pass nd_array.sum()
timer_numpy = Timer("nd_array.sum()", "from __main__ import nd_array")

print("Time taken by numpy ndarray: %f seconds" %
      (timer_numpy.timeit(timeits)/timeits))
```

In [ ]:
```
# create the list with values 0,1,2...,size-1
a_list = list(range(size))
print (type(a_list) )
```

In [ ]:
```
# timer expects the operation as a parameter, here we pass sum(a_list)
timer_list = Timer("sum(a_list)", "from __main__ import a_list")

print("Time taken by list:  %f seconds" %
      (timer_list.timeit(timeits)/timeits))
```

# Read or Write to Disk:

## Binary Format:

In [ ]:
```
x = np.array([ 23.23, 24.24] )
```

In [ ]:
```
np.save('an_array', x)
```

In [ ]:
```
np.load('an_array.npy')
```

## Text Format:

In [ ]:
```
np.savetxt('array.txt', X=x, delimiter=',')
```

In [ ]:
```
!cat array.txt
```

In [ ]:
```
np.loadtxt('array.txt', delimiter=',')
```

# Additional Common ndarray Operations

# Dot Product on Matrices and Inner Product on Vectors:

In [44]:
```
# determine the dot product of two matrices
x2d = np.array([[1,1],[1,1]])
y2d = np.array([[2,2],[2,2]])

print(x2d.dot(y2d))
print()
print(np.dot(x2d, y2d))
```

```
[[4 4]
 [4 4]]

[[4 4]
 [4 4]]
```

In [45]:
```
# determine the inner product of two vectors
a1d = np.array([9 , 9 ])
b1d = np.array([10, 10])

print(a1d.dot(b1d))
print()
print(np.dot(a1d, b1d))
```

```
180

180
```

In [ ]:
```
# dot produce on an array and vector
print(x2d.dot(a1d))
print()
print(np.dot(x2d, a1d))
```

# Sum:

In [78]:
```
# sum elements in the array
ex1 = np.array([[11,12],[21,22]])

print(np.sum(ex1))          # add all members
print(ex1.sum())
```

```
66
66
```

In [ ]:
```
print(np.sum(ex1, axis=0))  # columnwise sum
```

In [ ]:
```
print(np.sum(ex1, axis=1))  # rowwise sum
```

# Element-wise Functions:

For example, let's compare two arrays values to get the maximum of each.

In [46]:
```
# random array
x = np.random.randn(8)
x
```

Out[46]:
```
array([-0.84329306,  0.53577379, -0.91005308,  0.19388413,  1.1713282 ,
        1.26538085,  2.90171378,  0.53955055])
```

In [47]:
```
# another random array
y = np.random.randn(8)
y
```

Out[47]:
```
array([-0.36783814, -0.22151073,  1.76139501,  1.26496062,  0.12611553,
        1.15053687,  0.32329357, -1.64598334])
```

In [48]:
```
# returns element wise maximum between two arrays

np.maximum(x, y)
```

Out[48]:
```
array([-0.36783814,  0.53577379,  1.76139501,  1.26496062,  1.1713282 ,
        1.26538085,  2.90171378,  0.53955055])
```

# Reshaping array:

In [49]:
```
# grab values from 0 through 19 in an array
arr = np.arange(20)
print(arr)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

In [50]:
```
# reshape to be a 4 x 5 matrix
arr.reshape(4,5)
```

Out[50]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

# Transpose:

In [51]:
```
# transpose
ex1 = np.array([[11,12],[21,22]])

ex1.T
```

Out[51]:
```
array([[11, 21],
       [12, 22]])
```

# Indexing using where():

In [52]:
```
x_1 = np.array([1,2,3,4,5])

y_1 = np.array([11,22,33,44,55])

filter = np.array([True, False, True, False, True])
```

In [53]:
```
out = np.where(filter, x_1, y_1)
print(out)
```

```
[ 1 22  3 44  5]
```

In [54]:
```
mat = np.random.rand(5,5)
mat
```

Out[54]:
```
array([[ 0.67745669,  0.23937408,  0.19180203,  0.04914193,  0.59530901],
       [ 0.81153492,  0.97359325,  0.93238738,  0.78785179,  0.03321135],
       [ 0.35428268,  0.13906719,  0.93602509,  0.93450562,  0.47595539],
       [ 0.82772688,  0.94949964,  0.67273987,  0.34979533,  0.03106072],
       [ 0.83550762,  0.99417402,  0.52919541,  0.30741014,  0.99187241]])
```

In [55]:
```
np.where( mat > 0.5, 1000, -1) #ternary operator- if true, 1000 if false, -1.
 Same for previous example.
```

Out[55]:
```
array([[1000,   -1,   -1,   -1, 1000],
       [1000, 1000, 1000, 1000,   -1],
       [  -1,   -1, 1000, 1000,   -1],
       [1000, 1000, 1000,   -1,   -1],
       [1000, 1000, 1000,   -1, 1000]])
```

# "any" or "all" conditionals:

In [65]:
```
arr_bools = np.array([ True, False, True, True, False ])
```

In [67]:
```
arr_bools.any() #any of them true in the array
```

Out[67]:
```
False
```

In [64]:
```
arr_bools.all() #all of them true in the array
```

Out[64]: True

# Random Number Generation:

In [68]:
```
Y = np.random.normal(size = (1,5))[0]
print(Y)
```

```
[-0.06997155  0.59965832  0.06521293  0.19011809  0.16063399]
```

In [69]:
```
Z = np.random.randint(low=2,high=50,size=4)
print(Z)
```

```
[34  2 29 40]
```

In [70]:
```
np.random.permutation(Z) #return a new ordering of elements in Z
```

Out[70]:
```
array([40, 29, 34,  2])
```

In [71]:
```
np.random.uniform(size=4) #uniform distribution
```

Out[71]:
```
array([ 0.71689868,  0.06642476,  0.47574044,  0.72704265])
```

In [72]:
```
np.random.normal(size=4) #normal distribution
```

Out[72]:
```
array([-0.3189061 ,  2.55075888,  0.10701857,  0.99111263])
```

# Merging data sets:

In [73]:
```
K = np.random.randint(low=2,high=50,size=(2,2))
print(K)

print()
M = np.random.randint(low=2,high=50,size=(2,2))
print(M)
```

```
[[35  3]
 [36 23]]

[[26 45]
 [14 26]]
```

In [74]:
```
np.vstack((K,M))
```

Out[74]:
```
array([[35,  3],
       [36, 23],
       [26, 45],
       [14, 26]])
```

In [75]:
```
np.hstack((K,M))
```

Out[75]:
```
array([[35,  3, 26, 45],
       [36, 23, 14, 26]])
```

In [76]:
```
np.concatenate([K, M], axis = 0)
```

Out[76]:
```
array([[35,  3],
       [36, 23],
       [26, 45],
       [14, 26]])
```

In [77]:
```
np.concatenate([K, M.T], axis = 1)
```

Out[77]:
```
array([[35,  3, 26, 14],
       [36, 23, 45, 26]])
```

In [ ]: