Numpy is the fundamental package for numeric computing with Python. It provides powerful ways to create, store, and/or manipulate data, which makes it able to seamlessly and speedily integrate with a wide variety of databases. This is also the foundation that Pandas is built on, which is a high-performance data-centric package that we will learn later in the course.

In this lecture, we will talk about creating array with certain data types, manipulating array, selecting elements from arrays, and loading dataset into array. Such functions are useful for manipulating data and understanding the functionalities of other common Python data packages.

In [2]:

```
# You'll recall that we import a library using the `import` keyword as numpy's c
ommon abbreviation is np
import numpy as np
import math
```

Array Creation

In [3]:

```
# Arrays are displayed as a list or list of lists and can be created through lis
t as well. When creating an
# array, we pass in a list as an argument in numpy array
a = np.array([1, 2, 3])
print(a)
# We can print the number of dimensions of a list using the ndim attribute
print(a.ndim)
```

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

In [4]:

```
# If we pass in a list of lists in numpy array, we create a multi-dimensional ar
ray, for instance, a matrix
b = np.array([[1,2,3],[4,5,6]]) ##MATRIX!!
b
```

Out[4]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [5]:

```
# We can print out the length of each dimension by calling the shape attribute,
 which returns a tuple
b.shape ##RETURNS A TUPLE
```

Out[5]:

```
(2, 3)
```

In [6]:

```
# We can also check the type of items in the array
a.dtype
```

Out[6]:

```
dtype('int64')
```

In [7]:

```
# Besides integers, floats are also accepted in numpy arrays
c = np.array([2.2, 5, 1.1])
c.dtype.name
```

Out[7]:

```
'float64'
```

In [8]:

```
# Let's look at the data in our array
c
```

Out[8]:

```
array([2.2, 5. , 1.1])
```

Note that numpy automatically converts integers, like 5, up to floats, since there is no loss of precision. Numpy will try and give you the best data type format possible to keep your data types homogeneous, which means all the same, in the array.

In [11]:

```
# Sometimes we know the shape of an array that we want to create, but not what w
e want to be in it. numpy
# offers several functions to create arrays with initial placeholders, such as z
ero's or one's.
# Lets create two arrays, both the same shape but with different filler values
d = np.zeros((2,3))
print(d)

e = np.ones((2,3))
print(e)
```

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

In [12]:

```
# We can also generate an array with random numbers
np.random.rand(2,3)
```

Out[12]:

```
array([[0.83966547, 0.0716813 , 0.01595803],
       [0.80450225, 0.01109228, 0.45904509]])
```

You'll see zeros, ones, and rand used quite often to create example arrays, especially in stack overflow posts and other forums.

In [13]:

```
# We can also create a sequence of numbers in an array with the arrange() functi
on. The fist argument is the
# starting bound and the second argument is the ending bound, and the third argu
ment is the difference between
# each consecutive numbers

# Let's create an array of every even number from ten (inclusive) to fifty (excl
usive)
f = np.arange(10, 50, 2)
f
```

Out[13]:

```
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 4
0, 42,
       44, 46, 48])
```

In [14]:

```
# if we want to generate a sequence of floats, we can use the linspace() functio
n. In this function the third
# argument isn't the difference between two numbers, but the total number of ite
ms you want to generate
np.linspace( 0, 2, 15 ) # 15 numbers from 0 (inclusive) to 2 (inclusive)
```

Out[14]:

```
array([0.        , 0.14285714, 0.28571429, 0.42857143, 0.57142857,
       0.71428571, 0.85714286, 1.        , 1.14285714, 1.28571429,
       1.42857143, 1.57142857, 1.71428571, 1.85714286, 2.        ])
```

Array Operations

In [ ]:

```
# We can do many things on arrays, such as mathematical manipulation (addition,
 subtraction, square,
# exponents) as well as use boolean arrays, which are binary values. We can also
do matrix manipulation such
# as product, transpose, inverse, and so forth.
```

In [ ]:

```
# Arithmetic operators on array apply elementwise.

# Let's create a couple of arrays
a = np.array([10,20,30,40])
b = np.array([1, 2, 3,4])

# Now let's look at a minus b
c = a-b
print(c)

# And let's look at a times b
d = a*b ## the asterisk is for element-wise multiplication.
print(d)
```

The asterisk is for element-wise multiplication. Dot product of two arrays and/or matrices use the address operator ('@') So the dot product, d, of A and B where A and B are matrices is given by d = A @ B

In [15]:

```
# With arithmetic manipulation, we can convert current data to the way we want i
t to be. Here's a real-world
# problem I face - I moved down to the United States about 6 years ago from Cana
da. In Canada we use celcius
# for temperatures, and my wife still hasn't converted to the US system which us
es farenheit. With numpy I
# could easily convert a number of farenheit values, say the weather forecast, t
o celsius.

# Let's create an array of typical Ann Arbor winter farenheit values
farenheit = np.array([0,-10,-5,-15,0])

# And the formula for conversion is ((°F - 32) × 5/9 = °C)
celcius = (farenheit - 31) * (5/9)
celcius
```

Out[15]:

```
array([-17.22222222, -22.77777778, -20.        , -25.55555556,
       -17.22222222])
```

In [16]:

```
# Great, so now she knows it's a little chilly outside but not so bad.
```

Boolean Arrays

In [17]:

```
# Another useful and important manipulation is the boolean array. We can apply a
n operator on an array, and a
# boolean array will be returned for any element in the original, with True bein
g emitted if it meets the condition and False oetherwise.
# For instance, if we want to get a boolean array to check celcius degrees that
 are greater than -20 degrees
celcius > -20
```

Out[17]:

```
array([ True, False, False, False,  True])
```

In [18]:

```
# Here's another example, we could use the modulus operator to check numbers in
 an array to see if they are even. Recall that modulus does division but throws
 away everything but the remainder (decimal) portion)
celcius%2 == 0
```

Out[18]:

```
array([False, False,  True, False, False])
```

In [19]:

```
# Besides elementwise manipulation, it is important to know that numpy supports
 matrix manipulation. Let's
# look at matrix product. if we want to do elementwise product, we use the "*" s
ign
A = np.array([[1,1],[0,1]])
B = np.array([[2,0],[3,4]])
print(A*B)

# if we want to do matrix product, we use the "@" sign or use the dot function
print(A@B)
```

```
[[2 0]
 [0 4]]
[[5 4]
 [3 4]]
```

You don't have to worry about complex matrix operations for this course, but it's important to know that numpy is the underpinning of scientific computing libraries in python, and that it is capable of doing both element-wise operations (the asterix) as well as matrix-level operations (the @ sign). There's more on this in a subsequent course. A few more linear algebra concepts are worth layering in here. You might recall that the product of two matrices is only plausible when the inner dimensions of the two matrices are the same. The dimensions refer to the number of elements both horizontally and vertically in the rendered matricies you've seen here.

Some Linear Algebra Concepts

In [20]:

```
#We can use numpy to quickly see the shape of a matrix:
A.shape
```

Out[20]:

```
(2, 2)
```

In [22]:

```
# When manipulating arrays of different types, the type of the resulting array w
ill correspond to
# the more general of the two types. This is called upcasting.

# Let's create an array of integers
array1 = np.array([[1, 2, 3], [4, 5, 6]])
print(array1.dtype)

# Now let's create an array of floats
array2 = np.array([[7.0, 8.2, 9.1], [10.4, 11.2, 12.3]])
print(array2.dtype)
```

```
int64
float64
```

Integers (int) are whole numbers only, and Floating point numbers (float) can have a whole number portion and a decimal portion. The 64 in this example refers to the number of bits that the operating system is reserving to represent the number, which determines the size (or precision) of the numbers that can be represented.

In [23]:

```
# Let's do an addition for the two arrays
array3=array1+array2
print(array3)
print(array3.dtype)
```

```
[[ 8.  10.2 12.1]
 [14.4 16.2 18.3]]
float64
```

Notice how the items in the resulting array have been upcast into floating point numbers

In [25]:

```
# Numpy arrays have many interesting aggregation functions on them, such as  sum
(), max(), min(), and mean()
print(array3.sum())
print(array3.max())
print(array3.min())
print(array3.mean())
```

```
79.19999999999999
18.3
8.0
13.199999999999998
```

In [24]:

```
# For two dimensional arrays, we can do the same thing for each row or column
# let's create an array with 15 elements, ranging from 1 to 15,
# with a dimension of 3X5
# rmbr 2nd parameter of arange is excluded. //
b = np.arange(1,16,1).reshape(3,5)
print(b)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
```

Now, we often think about two dimensional arrays being made up of rows and columns, but you can also think of these arrays as just a giant ordered list of numbers, and the *shape* of the array, the number of rows and columns, is just an abstraction that we have for a particular purpose. Actually, this is exactly how basic images are stored in computer environments.

Let's take a look at an example and see how numpy comes into play.

In [27]:

```
# For this demonstration I'll use the python imaging library (PIL) and a functio
n to display images in the
# Jupyter notebook
from PIL import Image
from IPython.display import display

# And let's just look at the image I'm talking about
im = Image.open('chris.tiff')
display(im)
```

In [28]:

```
# Now, we can conver this PIL image to a numpy array
array=np.array(im)
print(array.shape)
array
```

```
(200, 200)
```

Out[28]:

```
array([[118, 117, 118, ..., 103, 107, 110],
       [113, 113, 113, ..., 100, 103, 106],
       [108, 108, 107, ...,  95,  98, 102],
       ...,
       [177, 181, 182, ..., 193, 198, 192],
       [178, 182, 183, ..., 193, 201, 189],
       [178, 182, 184, ..., 193, 201, 187]], dtype=uint8)
```

In [29]:

```
# Here we see that we have a 200x200 array and that the values are all uint8. Th
e uint means that they are
# unsigned integers (so no negative numbers) and the 8 means 8 bits per byte. Th
is means that each value can
# be up to 2*2*2*2*2*2*2*2=256 in size (well, actually 255, because we start at
 zero). For black and white
# images black is stored as 0 and white is stored as 255. So if we just wanted t
o invert this image we could
# use the numpy array to do so

# Let's create an array the same shape
mask=np.full(array.shape,255)
mask
```

Out[29]:

```
array([[255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255],
       ...,
       [255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255],
       [255, 255, 255, ..., 255, 255, 255]])
```

In [30]:

```
# Now let's subtract that from the modified array - element wise subtraction
modified_array=array-mask

# And lets convert all of the negative values to positive values
modified_array=modified_array*-1

# And as a last step, let's tell numpy to set the value of the datatype correctl
y
modified_array=modified_array.astype(np.uint8)
modified_array
```

Out[30]:

```
array([[137, 138, 137, ..., 152, 148, 145],
       [142, 142, 142, ..., 155, 152, 149],
       [147, 147, 148, ..., 160, 157, 153],
       ...,
       [ 78,  74,  73, ...,  62,  57,  63],
       [ 77,  73,  72, ...,  62,  54,  66],
       [ 77,  73,  71, ...,  62,  54,  68]], dtype=uint8)
```

In [31]:

```
# And lastly, lets display this new array. We do this by using the fromarray() f
unction in the python
# imaging library to convert the numpy array into an object jupyter can render
display(Image.fromarray(modified_array))
```

In [32]:

```
# Cool. Ok, remember how I started this by talking about how we could just think
of this as a giant array
# of bytes, and that the shape was an abstraction? Well, we could just decide to
reshape the array and still
# try and render it. PIL is interpreting the individual rows as lines, so we can
change the number of lines
# and columns if we want to. What do you think that would look like?
# Note that the total number of cells here are the same.
reshaped=np.reshape(modified_array,(100,400))
print(reshaped.shape)
display(Image.fromarray(reshaped))
```

```
(100, 400)
```

Can't say I find that particularly flattering. By reshaping the array to be only 100 rows high but 400 columns we've essentially doubled the image by taking every other line and stacking them out in width. This makes the image look more stretched out too.

This isn't an image manipulation course, but the point was to show you that these numpy arrays are really just abstractions on top of data, and that data has an underlying format (in this case, uint8). But further, we can build abstractions on top of that, such as computer code which renders a byte as either black or white, which has meaning to people. In some ways, this whole degree is about data and the abstractions that we can build on top of that data, from individual byte representations through to complex neural networks of functions or interactive visualizations. Your role as a data scientist is to understand what the data means (it's context an collection), and transform it into a different representation to be used for sensemaking.

Ok, back to the mechanics of numpy.

Indexing, Slicing and Iterating

Indexing, slicing and iterating are extremely important for data manipulation and analysis because these techinques allow us to select data based on conditions, and copy or update data.

Indexing

In [34]:

```
# First we are going to look at integer indexing. A one-dimensional array, works
in similar ways as a list -
# To get an element in a one-dimensional array, we simply use the offset index.
a = np.array([1,3,5,7])
a[2]
```

Out[34]:

```
5
```

In [38]:

```
# For multidimensional array, we need to use integer array indexing, let's creat
e a new multidimensional array
a = np.array([[1,2], [3, 4], [5, 6]])
a
```

Out[38]:

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [37]:

```
# if we want to select one certain element, we can do so by entering the index,
 which is comprised of two
# integers the first being the row, and the second the column
a[1,1] # remember in python we start at 0!
```

Out[37]:

```
4
```

In [39]:

```
# if we want to get multiple elements
# for example, 1, 4, and 6 and put them into a one-dimensional array
# we can enter the indices directly into an array function
np.array([a[0, 0], a[1, 1], a[2, 1]])
```

Out[39]:

```
array([1, 4, 6])
```

In [40]:

```
# we can also do that by using another form of array indexing, which essentially
"zips" the first list and the
# second list up // Note that what is passed in was a matrix, where [0,1,2] is t
he first row of values, and [0,1,1]
# is the 2nd row of values.
print(a[[0, 1, 2], [0, 1, 1]])
```

```
[1 4 6]
```

Boolean Indexing

In [41]:

```
# Boolean indexing allows us to select arbitrary elements based on conditions. F
or example, in the matrix we
# just talked about we want to find elements that are greater than 5 so we set u
p a conditon a >5
print(a >5)
# This returns a boolean array showing that if the value at the corresponding in
dex is greater than 5
```

```
[[False False]
 [False False]
 [False  True]]
```

In [71]:

```
# We can then place this array of booleans like a mask over the original array t
o return a one-dimensional
# array relating to the true values.
print (a)

print("\n",a[a>5])
```

```
[[ 1 50  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

 [50  6  7  8  9 10 11 12]
```

As we will see, this functionality is essential in the pandas toolkit which is the bulk of this course

Slicing

In [47]:

```
# Slicing is a way to create a sub-array based on the original array. For one-di
mensional arrays, slicing
# works in similar ways to a list. To slice, we use the : sign. For instance, if
we put :3 in the indexing
# brackets, we get elements from index 0 to index 3 (excluding index 3)
a = np.array([0,1,2,3,4,5])
print(a[:3])
```

```
[0 1 2]
```

In [48]:

```
# By putting 2:4 in the bracket, we get elements from index 2 to index 4 (exclud
ing index 4)
print(a[2:4])
```

```
[2 3]
```

In [49]:

```
# For multi-dimensional arrays, it works similarly, lets see an example
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
a
```

Out[49]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [50]:

```
# First, if we put one argument in the array, for example a[:2] then we would ge
t all the elements from the
# first (0th) and second row (1th), 2nd parameter is EXCLUSIVE
a[:2]
```

Out[50]:

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

In [52]:

```
# If we add another argument to the array, for example a[:2, 1:3], we get the fi
rst two rows but then the
# second and third column values only (excluding the 2nd parameter - so col inde
x 1, to col index 2)
a[:2, 1:3]
```

Out[52]:

```
array([[2, 3],
       [6, 7]])
```

So, in multidimensional arrays, the first argument is for selecting rows, and the second argument is for selecting columns

Passing By Reference

It is important to realize that a slice of an array is a view into the same data. This is called passing by reference. So modifying the sub array will consequently modify the original array

In [54]:

```
# Here I'll change the element at position [0, 0], which is 2, to 50, then we ca
n see that the value in the
# original array is changed to 50 as well

sub_array = a[:2, 1:3]
print("sub array index [0,0] value before change:", sub_array[0,0])
sub_array[0,0] = 50
print("sub array index [0,0] value after change:", sub_array[0,0])
print("original array index [0,1] value after change:", a[0,1])
print(a)
```

```
sub array index [0,0] value before change: 50
sub array index [0,0] value after change: 50
original array index [0,1] value after change: 50
[[ 1 50  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Trying Numpy with Datasets

Now that we have learned the essentials of Numpy let's use it on a couple of datasets

Here we have a very popular dataset on wine quality, and we are going to only look at red wines. The data fields include: fixed acidity, volatile aciditycitric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxidedensity, pH, sulphates, alcohol, quality

In [55]:

```
# To load a dataset in Numpy, we can use the genfromtxt() function. We can speci
fy data file name, delimiter
# (which is optional but often used), and number of rows to skip if we have a he
ader row, hence it is 1 here

# The genfromtxt() function has a parameter called dtype for specifying data typ
es of each column this
# parameter is optional. Without specifying the types, all types will be casted
 the same to the more
# general/precise type

wines = np.genfromtxt("datasets/winequality-red.csv", delimiter=";", skip_header
=1)
wines
```

Out[55]:

```
array([[ 7.4  ,  0.7  ,  0.   , ...,  0.56 ,  9.4  ,  5.   ],
       [ 7.8  ,  0.88 ,  0.   , ...,  0.68 ,  9.8  ,  5.   ],
       [ 7.8  ,  0.76 ,  0.04 , ...,  0.65 ,  9.8  ,  5.   ],
       ...,
       [ 6.3  ,  0.51 ,  0.13 , ...,  0.75 , 11.   ,  6.   ],
       [ 5.9  ,  0.645,  0.12 , ...,  0.71 , 10.2  ,  5.   ],
       [ 6.   ,  0.31 ,  0.47 , ...,  0.66 , 11.   ,  6.   ]])
```

Recall that we can use integer indexing to get a certain column or a row. For example, if we want to select the fixed acidity column, which is the first coluumn, we can do so by entering the index into the array. Also remember that for multidimensional arrays, the first argument refers to the row, and the second argument refers to the column, and if we just give one argument then we'll get a single dimensional list back.

In [56]:

```
# So all rows combined but only the first column from them would be
print("one integer 0 for slicing: ", wines[:, 0])
# But if we wanted the same values but wanted to preserve that they sit in their
own rows we would write
print("0 to 1 for slicing: \n", wines[:, 0:1])
```

```
one integer 0 for slicing:  [7.4 7.8 7.8 ... 6.3 5.9 6. ]
0 to 1 for slicing:
 [[7.4]
 [7.8]
 [7.8]
 ...
 [6.3]
 [5.9]
 [6. ]]
```

Note that both results when we execute them is so different! One of them is an array, another one of them is a matrix of 1 element arrays. This is another great example of how the shape of the data is an abstraction which we can layer intentionally on top of the data we are working with.

In [57]:

```
# If we want a range of columns in order, say columns 0 through 3 (recall, this
 means first, second, and
# third, since we start at zero and don't include the training index value), we
 can do that too
wines[:, 0:3]
```

Out[57]:

```
array([[7.4  , 0.7  , 0.   ],
       [7.8  , 0.88 , 0.   ],
       [7.8  , 0.76 , 0.04 ],
       ...,
       [6.3  , 0.51 , 0.13 ],
       [5.9  , 0.645, 0.12 ],
       [6.   , 0.31 , 0.47 ]])
```

In [58]:

```
# What if we want several non-consecutive columns? We can place the indices of t
he columns that we want into
# an array and pass the array as the second argument. Here's an example
wines[:, [0,2,4]]
```

Out[58]:

```
array([[7.4   , 0.    , 0.076],
       [7.8   , 0.    , 0.098],
       [7.8   , 0.04  , 0.092],
       ...,
       [6.3   , 0.13  , 0.076],
       [5.9   , 0.12  , 0.075],
       [6.    , 0.47  , 0.067]])
```

We can also do some basic summarization of this dataset. For example, if we want to find out the average quality of red wine, we can select the quality column. We could do this in a couple of ways, but the most appropriate is to use the -1 value for the index, as negative numbers mean slicing from the back of the list.

In [59]:

```
#We can then call the aggregation functions on this data. Remember that the -1 m
eans you are slicing from the last column!
wines[:,-1].mean()
```

Out[59]:

```
5.6360225140712945
```

Let's take a look at another dataset, this time on graduate school admissions. It has fields such as GRE score, TOEFL score, university rating, GPA, having research experience or not, and a chance of admission. With this dataset, we can do data manipulation and basic analysis to infer what conditions are associated with higher chance of admission. Let's take a look.

In [72]:

```
# We can specify data field names when using genfromtxt() to loads CSV data. Als
o, we can have numpy try and
# infer the type of a column by setting the dtype parameter to None
graduate_admission = np.genfromtxt('datasets/Admission_Predict.csv', dtype=None,
delimiter=',', skip_header=1,
                                   names=('Serial No','GRE Score', 'TOEFL Score'
, 'University Rating', 'SOP',
                                          'LOR','CGPA','Research', 'Chance of Ad
mit'))
graduate_admission[:10]
```

Out[72]:

```
array([( 1, 337, 118, 4, 4.5, 4.5, 9.65, 1, 0.92),
       ( 2, 324, 107, 4, 4. , 4.5, 8.87, 1, 0.76),
       ( 3, 316, 104, 3, 3. , 3.5, 8.  , 1, 0.72),
       ( 4, 322, 110, 3, 3.5, 2.5, 8.67, 1, 0.8 ),
       ( 5, 314, 103, 2, 2. , 3. , 8.21, 0, 0.65),
       ( 6, 330, 115, 5, 4.5, 3. , 9.34, 1, 0.9 ),
       ( 7, 321, 109, 3, 3. , 4. , 8.2 , 1, 0.75),
       ( 8, 308, 101, 2, 3. , 4. , 7.9 , 0, 0.68),
       ( 9, 302, 102, 1, 2. , 1.5, 8.  , 0, 0.5 ),
       (10, 323, 108, 3, 3.5, 3. , 8.6 , 0, 0.45)],
      dtype=[('Serial_No', '<i8'), ('GRE_Score', '<i8'), ('TOEFL_Sco
re', '<i8'), ('University_Rating', '<i8'), ('SOP', '<f8'), ('LOR',
'<f8'), ('CGPA', '<f8'), ('Research', '<i8'), ('Chance_of_Admit', '<
f8')])
```

In [61]:

```
# Notice that the resulting array is actually a one-dimensional array with 400 t
uples
graduate_admission.shape
```

Out[61]:

```
(400,)
```

In [76]:

```
# We can retrieve a column from the array using the column's name for example, l
et's get the CGPA column and
# only the first five values. //Only the ['CGPA'] column!
graduate_admission['CGPA'][0:5]
```

Out[76]:

```
array([9.65, 8.87, 8.  , 8.67, 8.21])
```

In [77]:

```
# Since the GPA in the dataset range from 1 to 10, and in the US it's more commo
n to use a scale of up to 4,
# a common task might be to convert the GPA by dividing by 10 and then multiplyi
ng by 4
graduate_admission['CGPA'] = graduate_admission['CGPA'] /10 *4 # "pass by refere
nce" effect
graduate_admission['CGPA'][0:20] #let's get 20 values
# This is changed on the data set as well! (Recall: Pass by reference)
#Click run this cell again and see what happened! :o
```

Out[77]:

```
array([3.86 , 3.548, 3.2  , 3.468, 3.284, 3.736, 3.28 , 3.16 , 3.2
,
       3.44 , 3.36 , 3.6  , 3.64 , 3.2  , 3.28 , 3.32 , 3.48 , 3.2
,
       3.52 , 3.4  ])
```

In [69]:

```
# Recall boolean masking. We can use this to find out how many students have had
research experience by
# creating a boolean mask and passing it to the array indexing operator
len(graduate_admission[graduate_admission['Research'] == 1]) ## See section on b
oolean indexing.
```

Out[69]:

```
219
```

In [70]:

```
# Since we have the data field chance of admission, which ranges from 0 to 1, we
can try to see if students
# with high chance of admission (>0.8) on average have higher GRE score than tho
se with lower chance of
# admission (<0.4)

# So first we use boolean masking to pull out only those students we are interes
ted in based on their chance
# of admission, then we pull out only their GPA scores, then we print the mean v
alues.
print(graduate_admission[graduate_admission['Chance_of_Admit'] > 0.8]['GRE_Scor
e'].mean())
print(graduate_admission[graduate_admission['Chance_of_Admit'] < 0.4]['GRE_Scor
e'].mean())
```

```
328.7350427350427
302.2857142857143
```

In [74]:

```
# Take a moment to reflect here, do you understand what is happening in these ca
lls?

# When we do the boolean masking we are left with an array with tuples in it sti
ll, and numpy holds underneath
# this a list of the columns we specified and their name and indexes
#See that the output is still in tuples.
graduate_admission[graduate_admission['Chance_of_Admit'] > 0.8][:10]
```

Out[74]:

```
array([( 1, 337, 118, 4, 4.5, 4.5, 9.65, 1, 0.92),
       ( 6, 330, 115, 5, 4.5, 3. , 9.34, 1, 0.9 ),
       (12, 327, 111, 4, 4. , 4.5, 9.  , 1, 0.84),
       (23, 328, 116, 5, 5. , 5. , 9.5 , 1, 0.94),
       (24, 334, 119, 5, 5. , 4.5, 9.7 , 1, 0.95),
       (25, 336, 119, 5, 4. , 3.5, 9.8 , 1, 0.97),
       (26, 340, 120, 5, 4.5, 4.5, 9.6 , 1, 0.94),
       (33, 338, 118, 4, 3. , 4.5, 9.4 , 1, 0.91),
       (34, 340, 114, 5, 4. , 4. , 9.6 , 1, 0.9 ),
       (35, 331, 112, 5, 4. , 5. , 9.8 , 1, 0.94)],
      dtype=[('Serial_No', '<i8'), ('GRE_Score', '<i8'), ('TOEFL_Sco
re', '<i8'), ('University_Rating', '<i8'), ('SOP', '<f8'), ('LOR',
'<f8'), ('CGPA', '<f8'), ('Research', '<i8'), ('Chance_of_Admit', '<
f8')])
```

In [75]:

```
# Let's also do this with GPA
print(graduate_admission[graduate_admission['Chance_of_Admit'] > 0.8]['CGPA'].me
an())
print(graduate_admission[graduate_admission['Chance_of_Admit'] < 0.4]['CGPA'].me
an())
```

```
9.276666666666667
7.555714285714287
```

Hrm, well, I guess one could have expected this. The GPA and GRE for students who have a higher chance of being admitted, at least based on our cursory look here, seems to be higher.

So that's a bit of a whirlwing tour of numpy, the core scientific computing library in python. Now, you're going to see a lot more of this kind of discussion, as the library we'll be focusing on in this course is pandas, which is built on top of numpy. Don't worry if it didn't all sink in the first time, we're going to dig in to most of these topics again with pandas. However, it's useful to know that many of the functions and capabilities of numpy are available to you within pandas.