Lecture 2: Numpy Module for Scientific Computing: Arrays

Python Modules

Python is organized around **modules**. A module is a collection of *functions* or *classes*.

- For now, you can think of a function or a class as some program that already exists, that you can make use of for your program, by *calling* the function or class.
- Some widely used modules and what they stand for

1. `numpy *` - Numerical functions for Python
2. `scipy` - Scientific functions for Python
3. `statsmodel` - Library of statistical tool (very similar to R)
4. `sklearn` - Data science methods
5. `pandas *` - File input, output and data organization methods.
6. `matplotlib *` - Plotting and Data Visualization functions.
7. `seaborn *` - A data science focused data visualization module. Pretty!

- How do I learn more about this. Google is your friend. Take more classes after this at UCI or online.
- '*' Will be introduced in this class

Functions and Classes

- A **function** is a specialized operation that has been programmed to take one or more *input* variables and return one or more *output* variables.
- When your call a function, you will write a line of code that looks something like this

```
output_variables = function_name(input_variables)
```

- There can be more than one output_variables, and more than one input_variables, separated by commas.
- A **class** is a template to create an **object** that has **methods** associated with it.

```
output_variables = object.method(input_variables)
```

Numpy

- The `numpy` **module** is used in almost all numerical computation using Python. I do not think I have ever written a program without numpy.
- To use `numpy` functions you need to *import* the module, **sub-module** , or a specific **function** inside a module or submodule.

Import the numpy module

- Because `numpy` is so generally useful, we usually import the entire numpy module. There are two ways to do this:
- `import numpy`
- This tells python to import `numpy`
- `import numpy as np`
- This tells python to import the `numpy` module and use the abbreviation `np` to refer to it in your program. This is convenient as we refer to functions as `np.function`.
- Note that you could have called it whatever abbreviation you want. `np` is widely used as an abbreviation for `numpy`.

In [4]:

```
import numpy as np
```

Creating `numpy` arrays

- In the `numpy` module the basic data type is an **array**.
- Working with an array is a lot like working with a list, but arrays are more disciplined, so we can do more with them.
- The critically important property is this:

Arrays only have 1 data type

- There are a number of ways to initialize new numpy arrays, for example from

1. using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`
2. creating an array filled with the same value at every entry using functions such as `zeros` or `ones`
3. creating an array filled with random numbers using functions such as `random`

- In working with data we will learn also how to create numpy arrays by

1. reading data from different types of files
2. converting other data types like `list` using functions such as `array`

Initializing an array of zeros

- Probably the most common way to create an array is to initialize all the elements of the array with the value 0.
- This can be done with the function `zeros`.
- A function takes input arguments
- To invoke this function use `np.zeros`. It is the function `zeros` in the `np` module.
- In principle, there could be another function called `zeros` in another module. Specifying the module avoids confusion.
- An array always has a **size**.

- In this example below i initialize an array called my_array with 5 elements, all of which have the value 0.

In [5]:

```
my_array = np.zeros(5)
print(my_array)
type(my_array)
```

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

Out[5]:

```
numpy.ndarray
```

- Notice that by default the array is floating point (note the decimal) and not integer.
- The type of variable is a numpy.ndarray

- An array with 2 dimensions is called a **matrix**. In this case, I have to specify the 2-dimensions in a special type of list called a `tuple` which has round brackets `()` instead of square `[]`

In [6]:

```
### Lets initialize a matrix
my_array2 = np.zeros((3,4))
print(my_array2)
```

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

- Notice the form of the array above. It has an outer square bracket, and then 3 separate numpy arrays of size 4 stacked on top of each other.

- Now I'm going to initialize a slightly different array. This is an array which has two dimensions. It has 1 row and 5 columns.

In [7]:

```
my_array3 = np.zeros((1,5))
print(my_array3)
```

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

- Take a look at the Variables pane, we can compare the dimensions of my_array3 to my_array. my_array3 has an extra dimension of size 1.
- Dimensions of size 1 are not really helpful. `numpy` has a function to get rid of them called `squeeze`

In [8]:

```
my_array4 = np.squeeze(my_array3)
print(my_array4)
```

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

Examining Array Size, Shape, and Data Type

- Arrays generated with numpy are `objects` that have built in `methods` . Some of these methods return properties of the arrays, and some of them can be used to manipulate the array

1. `shape` tells us the shape of an array
2. `size` tells us the total number of elements in an array
3. `dtype` query the array for the type of data contained in it.

In [9]:

```
#array shape
array_shape = my_array.shape
print(array_shape)
array2_shape = my_array2.shape
print(array2_shape)
```

```
(5,)
(3, 4)
```

In [10]:

```
#array size
array_size = my_array.size
print(array_size)
array2_size = my_array2.size
print(array2_size)
```

```
5
12
```

In [11]:

```
### Alternatively, you could obtain the same information by using the corresponding `numpy` functions
array2_shape = np.shape(my_array2)
print(array2_shape)
array2_size = np.size(my_array2)
print(array2_size)###
```

```
(3, 4)
12
```

But its a good idea to gain some experience with the object-oriented syntax, because some things cannot be done without it.

In [12]:

```
#dtype is only available as an object method
print(my_array2.dtype)
```

```
float64
```

- Notice an important difference between a list and an array.
- An array can have a data type because there can only be one data type in an array.
- A list cannot have a data type because it could have many different data types. Only an element of a list can have a data type.

Initializing an array of ones or any arbitrary value.

- Similar to the function `zeros` numpy has a function called `ones`
- You can use it to make arrays containing arbitrary values
- An array of ones can be multiplied by an arbitrary constant to make an array of that value.

In [13]:

```
my_one_array = np.ones(4)
print(my_one_array)
```

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

In [14]:

```
my_fives_array = 5*np.ones(10)
print(my_fives_array)
```

```
[5. 5. 5. 5. 5. 5. 5. 5. 5. 5.]
```

Array-generating functions

- `zeros` and `ones` are examples of *array generating* functions, which populates an array of user specified size with the value zero or one respectively.
- There are a number of very useful array generating functions in the numpy module.
- We will introduce the 2 of the most important ones here:
- `arange` - creates a range of linearly spaced values specifying interval
- `linspace` - lineary spaced values specifying number of values

- `arange` creates an array containing values from a starting point to an ending point with a user specified step.
- With `arange` **the ending point is not included in the array.**

In [15]:

```
# create a range
#using arange the stop point IS NOT included
x = np.arange(0, 10, 1) # arguments: start, stop, step
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

In [16]:

```
y = np.arange(-4, 4.1, 0.5)  #notice I stopped it at 4.1 instead of 4.
print(y)
```

```
[-4.  -3.5 -3.  -2.5 -2.  -1.5 -1.  -0.5  0.   0.5  1.   1.5  2.   2.5
  3.   3.5  4. ]
```

- `linspace` creates an array from a starting point to an ending point, with the number of points specified by the user rather than the step.
- With `linspace` **the ending point is included in the array**

In [17]:

```
# using linspace, both end points ARE included
z = np.linspace(0, 10, 26) #arguments: start, stop, number of points
print(z)
```

```
[ 0.   0.4  0.8  1.2  1.6  2.   2.4  2.8  3.2  3.6  4.   4.4  4.8  5.2
  5.6  6.   6.4  6.8  7.2  7.6  8.   8.4  8.8  9.2  9.6 10. ]
```

In [18]:

```
np.shape(z)
```

Out[18]:

```
(26,)
```

Converting other data types into an array

- The function `array` can be used to convert other data types into arrays.
- The most common use is to convert a **list** into a **ndarray**

In [19]:

```
my_list = [1,2,3]
print('my_list = ', my_list)
my_array = np.array(my_list)
print('my_array = ',my_array)
# Notice that the list is separated by commas while arrays are separated by spaces.
```

```
my_list =  [1, 2, 3]
my_array =  [1 2 3]
```

You should be able to tell if something is a list or an array by the commas

In [20]:

```
my_list = [2.0,2.9,13]
print('my_list = ', my_list)
my_array = np.array(my_list)
print('my_array = ',my_array)
# Notice that in this example my list had two floating point numbers and 1 integer.  By default, it will convert the integer into a floating point because an array can only have 1 data type
```

```
my_list =  [2.0, 2.9, 13]
my_array =  [ 2.   2.9 13. ]
```

- I could try to force the array to produce integers. This will cause all decimal values to be removed.

In [21]:

```
my_array_int = np.array(my_list,dtype = 'int') # notice I specified the dtype
print(my_array_int)
```

```
[ 2  2 13]
```

In [22]:

```
my_kawhi_list = ['Kawhi','Leonard','June',29,1991,79.0]
print('my_kawhi_list = ', my_kawhi_list)
my_kawhi_array = np.array(my_kawhi_list)
print('my_kawhi_array = ',my_kawhi_array)
#Notice that the presence of a string converts all of the elements of the list into an array of strings.
```

```
my_kawhi_list =  ['Kawhi', 'Leonard', 'June', 29, 1991, 79.0]
my_kawhi_array =  ['Kawhi' 'Leonard' 'June' '29' '1991' '79.0']
```

Mathematical Functions in numpy

Numpy has built in a large number of mathematical functions. Next class we will deep dive into this starting with random number generators.

Basic Math

Basic operations will work on numpy arrays, element by element:

- +, addition
- -, subtraction
- *, multiplication
- / division
- **, exponentiation
- //, floor division or integer division
- %, remainder

THEY WILL ONLY WORK WITH A SCALAR NUMBER OR WITH ARRAYS OF THE SAME SIZE

In [23]:

```
a = np.array([0.5, 1,2])
b = np.array([4,6,8])
c = np.array([-1,1])
print(a)
print(b)
print(c)
#lets test the operations above.
```

```
[0.5 1.  2. ]
[4 6 8]
[-1  1]
```

- Arithmetic with scalar numbers.
- We can use the usual arithmetic operators to multiply, add, subtract, and divide arrays with scalar numbers.

In [24]:

```
print(b)
add = b+2
print(add)
subtract = b-2
print(subtract)
multiply = b*2
print(multiply)
divide = b/2
print(divide)
```

```
[4 6 8]
[ 6  8 10]
[2 4 6]
[ 8 12 16]
[2. 3. 4.]
```

This is just a self exercise, to make sure you are comfortable with doing basic math operations.

verify what happens when you do

- a+b
- a-b
- a*b
- a/b
- a**2
- b/3
- b//3
- b%3

You dont need to submit these responses. Just do it for yourself in the box below, the results should make sense to you.

Also confirm you cannot do this

- a+c

This is because the arrays do not match in size.

In [ ]:

Manipulating Sign and Data Type

There are also some basic manipulation of the sign and type of data:

- `abs` , computes the absolute value
- `rint` , rounds to the nearest integer
- `floor` , return the first integer value less than the number
- `ceil` , return the first integer higher than the number
- `sign` , returns -1 for negative values and 1 for positive values

In [25]:

```
c = np.array([-1.75, -0.75, -0.5, 0, 0.5, 0.75, 1.75])
print(c)
#let's test the operations above
print('abs: ',np.abs(c))
print('rint: ',np.rint(c))
print('floor: ',np.floor(c))
print('ceil: ',np.ceil(c))
print('sign: ',np.sign(c))
```

```
[-1.75 -0.75 -0.5   0.    0.5   0.75  1.75]
abs:  [1.75 0.75 0.5  0.   0.5  0.75 1.75]
rint:  [-2. -1. -0.  0.  0.  1.  2.]
floor:  [-2. -1. -1.  0.  0.  0.  1.]
ceil:  [-1. -0. -0.  0.  1.  1.  2.]
sign:  [-1. -1. -1.  0.  1.  1.  1.]
```

Index slicing

- Indexing with arrays works identically to indexing with lists.
- Index slicing is the technical name for the syntax `v[lower:upper:step]` to extract part of an array:

In [26]:

```
A = [7, 2, 1, 4, 2, 5, 6, 9] # here A is a list
print(A)
A = np.array(A) #here I converted A into a numpy array so i could do math.
print(A)
a = A[1:3] #notice it is inclusive of the lower bound and exclusive of the upper bound
       #also notice I did not specifiy a step, so it defaulted to 1.
print(a)
```

```
[7, 2, 1, 4, 2, 5, 6, 9]
[7 2 1 4 2 5 6 9]
[2 1]
```

In [27]:

```
print(A)
b = A[6:]  #start at element 6 go to end of array
```

```
[7 2 1 4 2 5 6 9]
```

In [28]:

```
print(A)
c = A[1:7:2]  #In this example, i use step of size 2. Since my upper bound is 7, index 7 is excluded
print(c)
```

```
[7 2 1 4 2 5 6 9]
[2 4 5]
```

In [29]:

```
# We can modify this set of entries in the array if we correctly match the dimensions
print(A)
newvalues = np.array([-5,-10,-15]) # I made a new array from a list
A[1:7:2] = newvalues
print(A)
```

```
[7 2 1 4 2 5 6 9]
[  7  -5   1 -10   2 -15   6   9]
```

Indexing backwards from the end of an array

- When indexing into arrays, we sometimes want to get the last element, or a certain number of elements of the list or array.
- Of course if you know how long the array is in advance, you can easily solve this.
- Python supports indexing from the end of an array using negative indexes.

In [30]:

```
w = np.array([-10,-5, -2.5,0,2.5,5,10])
last_w = w[-1]
print(last_w)
last_2w = w[-2]
print(last_2w)
last_all3 = w[-3:]
print(last_all3)
```

```
10.0
5.0
[ 2.5  5.  10. ]
```

Indexing into arrays manually and using `range`

- Indexing into arrays works identically to indexing into lists
- The `range` function can be helpful in indexing into arrays
- The `range` function is useful for automatically making list of indices.

In [31]:

```
c = np.array([-1.75, -0.75, -0.5, 0, 0.5, 0.75, 1.75])
## this is the 4th element of c
print(c[3])
## This is the first 3 elements of c
print(c[0:3])
## you could also do this with the range function
print(c[range(4)])
## This is the last 2 elements of c
print(c[-2:])
## This is the 1st,3rd, and 5th element of c
print(c[0:5:2])
## You could also do this with the range function
print (c[range(0,5,2)])
## This is the first, 4th, and 8th element of c
print(c[range(0,7,3)])
```

```
0.0
[-1.75 -0.75 -0.5 ]
[-1.75 -0.75 -0.5   0.  ]
[0.75 1.75]
[-1.75 -0.5   0.5 ]
[-1.75 -0.5   0.5 ]
[-1.75  0.    1.75]
```

Indexing Ranges:

- In python, indexing can be done with defaults to the full range of values. This can be very useful when writing code to index into an array whose length may change in different runs of the code, e.g., trials of an experiment.
- Some ways we could make use of this incude

1. `x[n:]` - start with the index n entry of x all the way to the last entry
2. `x[:m]` - take the first m elements
3. `x[::n]` - start at the begining go the end and take every nth entry in the array.
4. `x[-n:]` - take the last n entries in the array

- Some syntax examples are shown below. Run the block of code

In [32]:

```
x = np.arange(-10,11,2)
print('all ')
print(x)
print('from the 3rd item to end is x[2:]')
print(x[2:])
print('take the first 4 items is x[:4]')
print(x[:4])
print('every other item is x[::2]')
print(x[::2])
print('every other item starting from the 2nd item is x[1::2]')
print(x[1::2])
```

```
all
[-10  -8  -6  -4  -2   0   2   4   6   8  10]
from the 3rd item to end is x[2:]
[-6 -4 -2  0  2  4  6  8 10]
take the first 4 items is x[:4]
[-10  -8  -6  -4]
every other item is x[::2]
[-10  -6  -2   2   6  10]
every other item starting from the 2nd item is x[1::2]
[-8 -4  0  4  8]
```

Arrays strictly control data type.

- First, lets make an array of integers and an array of floating point numbers.

In [33]:

```
A = np.array([7, 0, 6, 2, 4, 2, 0])
print(A)
B = np.array([-4.75,-3.35,-5.95])
print(B)
```

```
[7 0 6 2 4 2 0]
[-4.75 -3.35 -5.95]
```

In [34]:

```
## Now lets copy B into our integer array A , starting at the 3rd entry to the 5th entry
print(A[3:6])  # print out the current values of A from the 3rd to 5th entry
A[3:6] = B  # replace these with B
print(B)
print(A[3:6]) #print new values in A
```

```
[2 4 2]
[-4.75 -3.35 -5.95]
[-4 -3 -5]
```

- A is an integer array. If you place floating point numbers in A, it will **crop** the decimal values.
- When you create an array, it has a data type, and when you put something into the array it will convert it to that data type.

- When you create an array you can force its data type.

In [35]:

```
lst = [1,2,3]
lst_int = np.array(lst)
lst_float = np.array(lst,dtype='float')
print(lst)
print(lst_int)
print(lst_float)
```

```
[1, 2, 3]
[1 2 3]
[1. 2. 3.]
```

In [36]:

```
lst2 = [1.5,2.5,3.5]
lst2_int = np.array(lst2)
lst2_float = np.array(lst2,dtype='int')
print(lst2)
print(lst2_int)
print(lst2_float)
```

```
[1.5, 2.5, 3.5]
[1.5 2.5 3.5]
[1 2 3]
```