Day5_Intro_to_Numpy

August 4, 2021

Day 5: Introduction to Numpy

Amanda Farah

afarah@uchicago.edu

Goals for the day:

- Create arrays using numpy
- Create n-dimensional arrays
- Do array-wise operations and math
- Create 2D arrays, use linear algebra
- Load data in numpy
- Learn Boolean indexing
- Use various numpy functions

Functions Learned:

- Import numpy library: `import numpy as np`
- Create a numpy array: `np.arange(start,stop)`
- For floats: `np.linspace(start,stop,num)`
- Make an array of booleans: `bool_array= []`
- Check the shape of a loaded array: `print(data.shape)`
- Check the size of a loaded array: `print(data.size)`
- Check the dtype of a loaded array: `print(data.dtype)`

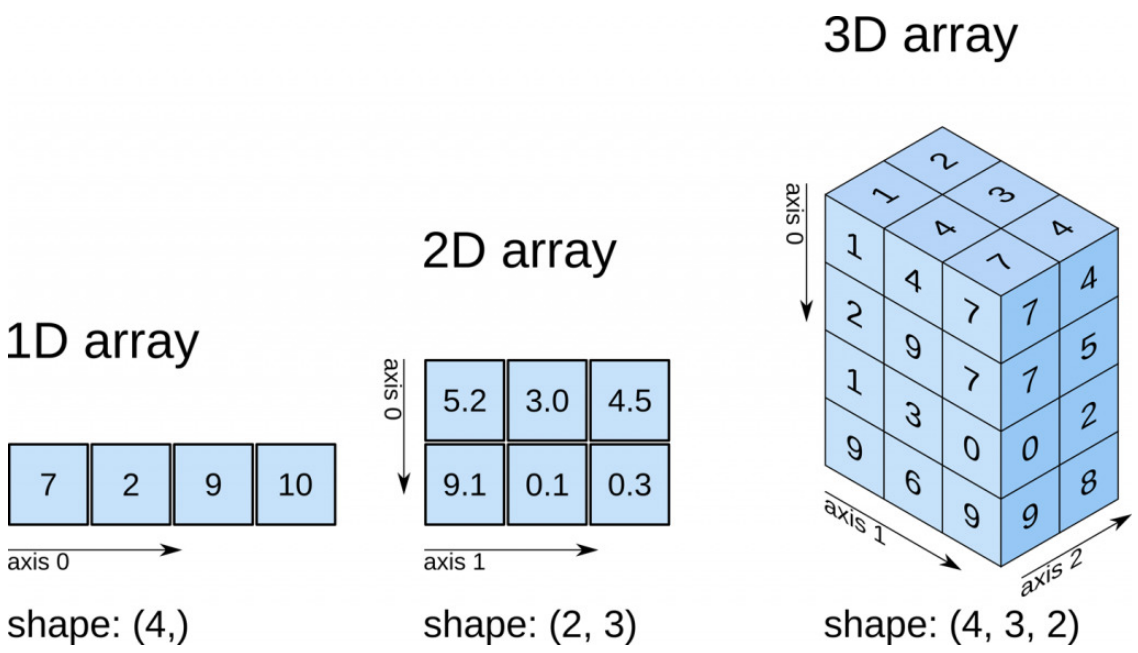
We need to import a library called `numpy` (short for Numperical Python). Libraries are a collection of lots and lots of code that other people wrote. You can use other people's code to make things easier for you, so you don't have to write it yourself every time you make a new program. Thanks, library developers!

In general you should use NumPy if you want to do fancy things with numbers, especially if you have matrices or arrays. It is the bread and butter of scientific computing with Python! We can load NumPy using: `import numpy as np` where here we are telling python that we will use the nickname `np` every time we really mean `numpy` so that we save time typing.

[1]:
```
import numpy as np
```

1. Numpy Arrays

`numpy` arrays are the central data structure of the `numpy` library. An array is a grid of values that all have the same type. It can have any number of dimensions and any shape.



1.1 Initialize a 1D array of numbers placed in ascending order.

[2]:
```
# np.arange(stop) or np.arange(start,stop) or np.arange(start,stop,step)
# np.arange() is the numpy array version of python's built-in function range()
# it starts at zero and outputs integers by default
print(np.arange(5))
#you can change the default behavior by putting in additional arguments
print(np.arange(3,10,dtype=float))
```

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

[3]:
```
# np.linspace(start,stop,num)
print(np.linspace(0,9) )
```

```
[0.         0.18367347 0.36734694 0.55102041 0.73469388 0.91836735
 1.10204082 1.28571429 1.46938776 1.65306122 1.83673469 2.02040816
 2.20408163 2.3877551  2.57142857 2.75510204 2.93877551 3.12244898
 3.30612245 3.48979592 3.67346939 3.85714286 4.04081633 4.2244898
 4.40816327 4.59183673 4.7755102  4.95918367 5.14285714 5.32653061
 5.51020408 5.69387755 5.87755102 6.06122449 6.24489796 6.42857143
 6.6122449  6.79591837 6.97959184 7.16326531 7.34693878 7.53061224
 7.71428571 7.89795918 8.08163265 8.26530612 8.44897959 8.63265306
 8.81632653 9.        ]
```

[4]:
```
#default number of entries is 50, so if you want a different length, must␣
 ↪explicitly specify
print(np.linspace(0,9,num=20))
```

```
[0.         0.47368421 0.94736842 1.42105263 1.89473684 2.36842105
 2.84210526 3.31578947 3.78947368 4.26315789 4.73684211 5.21052632
 5.68421053 6.15789474 6.63157895 7.10526316 7.57894737 8.05263158
 8.52631579 9.        ]
```

[5]:
```
#Use dtype to change the output format, default is float
print(np.linspace(0,9,num=10,dtype=int))
```

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

1.1.1 Check-in [1]: What is the difference between `linspace()` and `arange()`?

What are the differences in default behavior? Under what conditions do they make the same output? When might one be more useful than the other?

Answer these questions briefly in the Slack.

[6]:
```
#play around here to answer the above questions
```

Answer

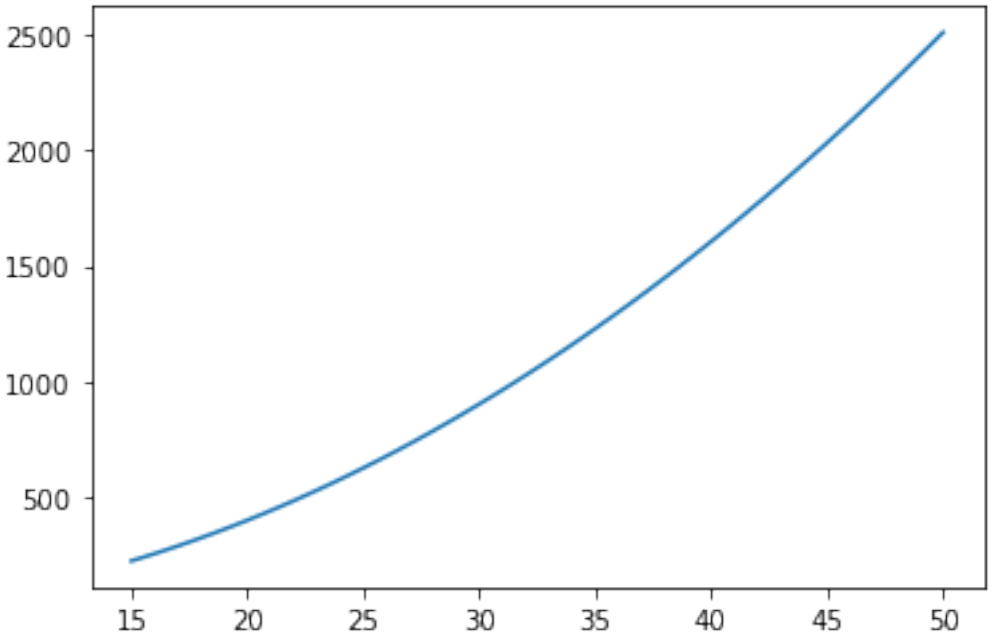
[7]:
```
#range always puts out integers (although they can still have dtype float and␣
 ↪act like floats)
#linspace can output numbers with decimals
#you can make linspace act like range if you do this:
print(np.arange(5))
print(np.linspace(0,4,num=5,dtype=int))
```

```
[0 1 2 3 4]
[0 1 2 3 4]
```

[9]:
```
def func_to_plot(x):
    return x**2 + 5
```

[10]:
```
import matplotlib.pyplot as plt

x = np.linspace(15,50)
y = func_to_plot(x)
plt.plot(x,y)
plt.show()
```



1.2 Initialize a 1D array from a list of your choosing

Arrays can be initialized from lists, like the ones you made on Day 2.

[11]:
```
#make a list here named l
l =  [10,13.4,25.777] #fill me in so you don't get an error!
print(l)
```

```
[10, 13.4, 25.777]
```

[12]:
```
arr_from_list = np.array(l)
arr_from_list
```

[12]:
```
array([10.   , 13.4  , 25.777])
```

1.3 Make a 1D numpy array full of zeros/ones.

`np.zeros(10)` `np.ones(10)`

[13]:
```
#Make a 1D numpy array full of zeros with 10 elements
a = np.zeros(10,dtype=int)
a
```

[13]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

1.3.1 Check-in [2]:

Make a 1D numpy array full of ones with 13 elements

[14]:
```
#now you try: Make a 1D numpy array full of ones with 13 elements
```

Answer

[15]:
```
a = np.ones(13)
a
```

[15]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

1.4 Make a N-dimensional ndarray full of zeros/ones .

```
a = np.zeros((3,3))
```

[16]:
```
#make an N-dimensional ndarray full of zeros:
#here I will make a 3x5 array:
a = np.zeros((3,5))
a #has 3 rows and 5 columns
```

[16]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

[17]:
```
#check its size and shape!
print(a.size) #should be 3*5
print(a.shape)
```

```
15
(3, 5)
```

1.4.1 Check-in [3]:

make a 10x6 2d array full of ones and check its size and shape

[18]:
```
#now you try: make an 10X6 ndarray full of ones and check its size and shape
```

Answer

[19]:
```
a = np.ones((10,6))
print(a.shape)
print(a.size)
```

```
(10, 6)
60
```

1.5 Initialize an array with random numbers drawn from a distribution of your choosing

numpy has great random number capabilities that are really easy to use. Look up `np.random.uniform()`, `numpy.random.randint()`, `numpy.random.seed()` for a few good examples. Here, I'll use the standard Gaussian distribution $\mathcal{N}(0,1)$

[20]:
```
# Initializing a random matrix with specified dimensions
# np.random.randn(): Return a sample (or samples) from the "standard normal"␣
 ↪distribution.
a = np.random.randn(5,2)
#this should return a 5x2 array full of random samples from N(0,1)
a
```

[20]:
```
array([[ 1.07477666, -2.5118637 ],
       [-0.37275407,  0.19008447],
       [-0.5941813 ,  0.04880786],
       [ 0.4393388 , -0.93783413],
       [-0.39662578,  1.17075649]])
```

1.6 Figure out the datatype of the elements in an array

[21]:
```
#lets make a 3x7 ndarray full of 1s
a = np.ones((3,7))
#what datatype did numpy assign to this array?
print(a.dtype)
```

```
float64
```

it assigned it as a float64! This is the default behavior of numpy arrays, but if I want it to be something else I can specify

[22]:
```
a = np.ones((3,7),dtype=int)
print(a.dtype)
a = str(a)
a #note that this whole array is a single string now, different rows are␣
 ↪separated by "newline"s, indicated by \n
```

```
int64
```

[22]:
```
'[[1 1 1 1 1 1 1]\n [1 1 1 1 1 1 1]\n [1 1 1 1 1 1 1]]'
```

[23]:
```
#if you initialize an array from a list, it will take whatever dtype your list␣
 ↪had
arr_from_list.dtype
```

[23]:
```
dtype('float64')
```

2. Reading in Data

In order to read data from elsewhere in our google drive, we have to give this notebook permission to access our google drive. Run the cell below, click on the link it outputs, and copy the authorization code and paste it into the box that appears. Then press enter.

[24]:
```
loaddir = '../data/' #Make sure the paths end in '/'
savedir = '../output/'
```

The expression `numpy.loadtxt(...)` is a function call that asks Python to run the function `loadtxt()` that belongs to the numpy library.

`numpy.loadtxt` has two parameters: the name of the file we want to read, and the delimiter that separates values on a line. These both need to be strings, so we put them in quotes.

In this case, that output is the data we just loaded. By default, only a few rows and columns are shown (with ... to omit elements when displaying big arrays).

[25]:
```
np.loadtxt(loaddir+'star_formation_rate_MD.csv', delimiter=',')
```

[25]:
```
array([[ 1.018, -1.958],
       [ 1.055, -1.825],
       [ 1.026, -1.72 ],
       [ 1.124, -1.751],
       [ 1.151, -1.634],
       [ 1.262, -1.692],
       [ 1.308, -1.554],
       [ 1.292, -1.507],
       [ 1.377, -1.416],
       [ 1.485, -1.437],
       [ 1.533, -1.315],
       [ 1.56 , -1.215],
       [ 1.693, -1.241],
       [ 1.693, -1.198],
       [ 1.699, -1.135],
       [ 1.877, -1.108],
       [ 2.008, -1.229],
       [ 1.899, -0.933],
       [ 2.14 , -1.022],
       [ 2.174, -0.954],
       [ 2.098, -0.938],
       [ 2.106, -0.801],
       [ 2.131, -0.837],
       [ 2.449, -0.948],
       [ 2.589, -0.931],
       [ 2.459, -0.826],
       [ 2.77 , -0.741],
       [ 2.837, -0.857],
       [ 3.071, -0.793],
```

```
       [ 3.121, -0.735],
       [ 3.299, -0.74 ],
       [ 3.234, -0.851],
       [ 3.247, -0.893],
       [ 3.761, -0.866],
       [ 4.024, -0.966],
       [ 4.008, -1.029],
       [ 4.624, -1.345],
       [ 4.831, -1.271],
       [ 5.006, -1.678],
       [ 6.01 , -1.396],
       [ 6.879, -1.634],
       [ 8.   , -1.781],
       [ 8.032, -1.992],
       [ 8.941, -2.065],
       [ 9.012, -2.187],
       [ 2.518, -0.889]])
```

Cool! Now lets save all that data as a variable

[26]:
```
data = np.loadtxt(loaddir+'star_formation_rate_MD.csv', delimiter=',')
```

`data` is now a numpy array that we can work with in the same way that we worked with the ones we created from scratch!

2.1 Check-in [4]: Check the dtype and size of a loaded array

[27]:
```
#now you try: verify that "data" can in fact be treated like the numpy arrays␣
 ↪you created earlier.
#print the size and dtype of "data"
```

Answer

[28]:
```
print(data.shape)
print(data.size)
print(data.dtype)
```

```
(46, 2)
92
float64
```

Break

3. Indexing

Numpy arrays can act like lists when convenient - they index just like lists, but can do so much more as well!

Let's start by applying the indexing you did on Day 2 to a numpy array.

Check-in [5]: lists vs np arrays

[29]:
```
animals = ['cat', 'dog', 'squid', 'moose', 'falcon'] #list from day 2
animal_arr = np.array(animals) #make an array out of this list!
```

[30]:
```
#return the first element of animal_arr in this cell
```

[31]:
```
#return the first two elements of animal_arr
```

[32]:
```
#return the last element
```

Reflect: Did you get the same results as on Day 2?

Answer

[33]:
```
animals = ['cat', 'dog', 'squid', 'moose', 'falcon'] #list from day 2
animal_arr = np.array(animals)
print('first element: ', animal_arr[0])
print('first two elements: ', animal_arr[:2])
print('last element: ', animal_arr[-1])
print('last element in a different way: ', animal_arr[4])
```

```
first element:  cat
first two elements:  ['cat' 'dog']
last element:  falcon
last element in a different way:  falcon
```

3.1 Fancy Indexing: Slicing

Use colon to slice arrays:

- Keep all elements: `[:]`.
- Explicitly specify start and end element `[start:stop]`.
- Specify the step/increment `[start:stop:step]`

[34]:
```
#1d array
arr = np.arange(12)
#keep all of the elements
arr[:] #returns the full array
```

[34]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

[35]:
```
#return all elements between indicies 2 and 7
arr[3:7]
```

[35]:
```
array([3, 4, 5, 6])
```

[36]:
```
#return every other element (step=2) between 0 and 8
print('look at every other element of the 1d array, from the 0th to the 8th')
print(arr[1:8:2])
```

```
look at every other element of the 1d array, from the 0th to the 8th
[1 3 5 7]
```

[37]:
```
# 2d array
a = np.arange((12)).reshape((6,2))
print(a)
print('just look at the middle 4 rows of the 2d array')
print(a[1:5,:])
```

```
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
just look at the middle 4 rows of the 2d array
[[2 3]
 [4 5]
 [6 7]
 [8 9]]
```

Reflect: what do you think this will return: `a[0:8:2]` ? Guess before you try in the cell below. Put your guess in the Slack before you run the cell!

[38]:
```
# try here!
a[0:8:2,:]
```

[38]:
```
array([[0, 1],
       [4, 5],
       [8, 9]])
```

3.1.1 For your future reference: specify elements and slices

you can also indicate explicitly which elements you want:

- 1d array: `[[element1,element2]]`
- 2d array (specify rows): `[[row1,row2],:]` means all of rows `row1` and `row2`
- 2d array (specify columns): `[:,[col1,col2]]`
- 2d array (specify coordinates: row and column): `[[row1,row3],[col4,col6]]`

[39]:
```
#specify elements for 1d array
print(np.random.random(12)[[1,3,4]])
#you can also indicate explicitly which rows you want
print(a[[1,3,4],:])
#and which coordinates
print(a[[1],[0]])
```

```
[0.40945339 0.6070913  0.31350515]
[[2 3]
 [6 7]
```

```
 [8 9]]
[2]
```

Reverse the array:

- Use np.flip
- Use ::-1

These are like `reverse()`, the built-in function for lists that we learned on Day 2

[40]:
```
#Reverse the array
print(arr[::-1])
print(np.flip(arr))
```

```
[11 10  9  8  7  6  5  4  3  2  1  0]
[11 10  9  8  7  6  5  4  3  2  1  0]
```

3.2 Fancy Indexing: Index with Boolean arrays

Slice the array using customized conditions/Boolean operators.

If you have an array `arr` with contents of any dtype and size `N x M` and an array `bool_arr` with Boolean contents and same size `N x M`, `arr[bool_arr]` will return the values of `arr` where `bool_arr` is `True`.

Sidenote: This is the simplest way to "mask", although there are certain things you can't do with this. Look up "masked arrays" using the `np.ma` module, especially if you work with images or spatial data

[41]:
```
arr = np.arange((12))
#Pick out the ones larger than certain number
loc = arr>3 #this returns a list of Booleans
print(arr[loc])
```

```
[ 4  5  6  7  8  9 10 11]
```

[42]:
```
# a different way to get the same result
# use the numpy function np.where(), which returns an array of Booleans
print(arr[np.where(arr > 3)])
```

```
[ 4  5  6  7  8  9 10 11]
```

[43]:
```
#Pick out the odd numbers
loc = (arr % 2 == 1)
print(loc)
print(arr[loc])
```

```
[False  True False  True False  True False  True False  True False  True]
[ 1  3  5  7  9 11]
```

[44]:
```
#even numbers
print(arr[(arr % 2 == 0)])
```

```
[ 0  2  4  6  8 10]
```

3.2.1 Check-in [6] : Boolean indexing

create your own Boolean array called `bool_array` and use it to index a 1d numpy array. Paste your `bool_array` in the slack!

[45]:
```
# now you try:
arr_to_index = np.arange(5)
# make an array or list of booleans that has the same length as arr_to_index
#bool_array =
```

[46]:
```
# then, use bool_arr to index arr. Try to predict what you will get before you␣
↪run the cell
```

Answer:

[47]:
```
#here is one example of what your bool_array could be.
#you can make it a list or an array, here i made it a list
bool_array = [True, False,True,True,False]
arr_to_index[bool_array]
```

[47]:
```
array([0, 2, 3])
```

Break

4. Why NumPy?

We went through all of this stuff to show you how numpy arrays act similarly to lists, how there are numpy functions that mirror the built-in functions that python has, etc. So why use numpy?

1. Because you can do SO MUCH MORE with a numpy array than a list! In addition to the functions that mirror built-in python functions for lists, there are hundreds of other functions that the numpy developers have made available to us. We will explore some of those here
2. Because there are "array-wise operations" that you can do on numpy arrays that make things way faster and simpler by avoiding for loops.
3. The ability to do linear algebra really easily and quickly on numpy arrays is a result of (1) and (2) - we will check some of that out as well! When you hear that python (and R, and matlab) are "vectorized" languages, this is what that means! You can treat arrays like vectors and matricies instead of dealing with each individual element in the arrays.

4.1 Numpy functions

Numpy functions usually have intuitive names. I usually just guess what I think they are called. If I'm wrong, I can just google it!

[48]:
```
#lets start out with a 1D array with 10 random elements
arr = np.random.uniform(size=10)
```

```
#lets find the standard deviation of elements in the array
print(arr.std())
print(np.std(arr))
#two different ways to call functions on numpy arrays, all of which are valid!
```

```
0.22676866194268755
0.22676866194268755
```

4.1.1 Check-in [7]: averages

[49]:
```
#now you try: what is the mean of the elements in the array? Guess the function␣
↪name!
```

Answer

[50]:
```
np.mean(arr) #or, arr.mean()
```

[50]:
```
0.26137701914252537
```

4.1.2 Examples of useful `numpy` functions

[51]:
```
#an assortment of useful functions:
med = np.median(arr)
per_90 = np.percentile(arr,90) #90th percentile
summation = np.sum(arr)
print(med, per_90,summation)
```

```
0.251097419891664 0.44455491292735305 2.6137701914252536
```

[52]:
```
#you can also do more math with numpy
print(np.exp(arr) )#take e^(all elements in array)
print(np.power(arr,5)) #raise all elements to the 5th power
```

```
[1.24586758 2.30271215 1.02640162 1.33407849 1.07128417 1.32625967
 1.00323971 1.2205136  1.49372575 1.33716274]
[5.13400077e-04 4.03699664e-01 1.20170576e-08 1.98965223e-03
 1.54801346e-06 1.79488503e-03 3.54009996e-13 3.14216334e-04
 1.04040493e-02 2.07063952e-03]
```

4.1.3 Numpy functions on `ndarrays`

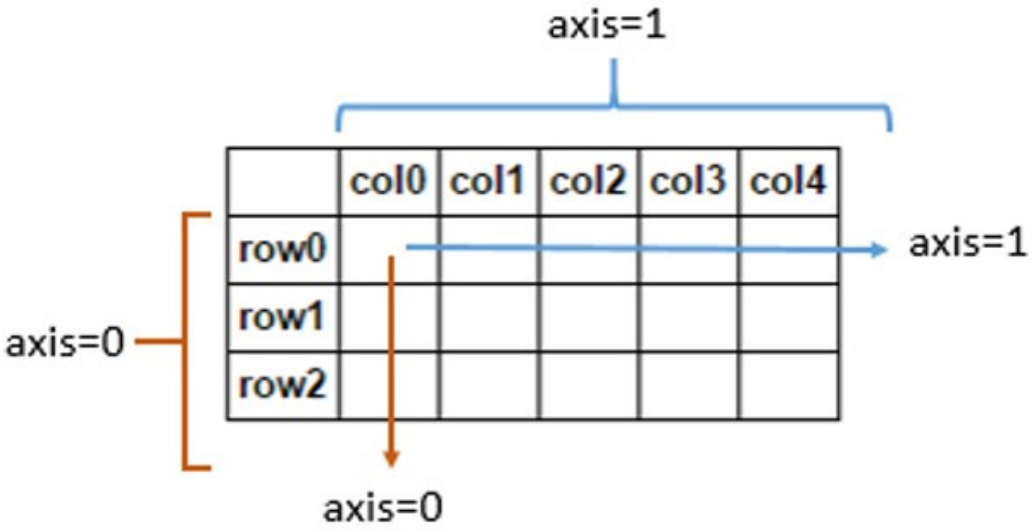
Unlike lists, ndarrays are truely N-dimensional (you can have lists inside of lists inside of lists, but you cant have a 2D or 3D list. With numpy arrays, you have multi-dimensional objects with multiple *axes*. You can perform operations along those axes.

[53]:
```
#lets make a 3x5 array with random variables in it.
arr_2d = np.random.uniform(size=(3,5))
# This array is 2D and we can think of it like a matrix
```

```
arr_2d
```

[53]:
```
array([[0.0833124 , 0.12588925, 0.73937308, 0.54665309, 0.28097398],
       [0.19883202, 0.43469326, 0.71689917, 0.69471867, 0.6031843 ],
       [0.93390934, 0.64611197, 0.47591701, 0.36351978, 0.23012451]])
```

we can think of this array as having an x and y axis. Numpy calls these the 1st and 0th axis. Its reversed from how you might think about it - axis 0 goes down the columns, and axis 1 goes across the rows.



now we can take the mean along the 0th axis. This should return 5 values since we are taking the mean across all rows

[54]:
```
arr_2d.mean(axis=0)
```

[54]:
```
array([0.40535125, 0.40223149, 0.64406309, 0.53496384, 0.37142759])
```

4.1.4 Check-in [8] Medians across columns

Paste your answer in the Slack!

[55]:
```
# now you try: take the *median* across all columns.
# What axis do you need to choose?

#before you run the cell, predict how many values it should return.
```

We don't *have* to do things along an axis, we can instead do operations on all elements. But it is usually convenient to.

[56]:
```
arr_2d.mean() #returns one value, the mean of all elements in all the rows and␣
 ↪columns
```

[56]: 0.4716074548129855

4.1.5 Check-in [9]: apply numpy functions to different axes of arrays with many dimensions

Make a 3x5x7 array of your choosing. You can fill it with random numbers, ones, zeros, tens, you name it. Just make sure they are floats

[57]:
```
# find the 20th percentiles along the 0th axis
```

[58]:
```
# find the sums along the 2nd axis
```

[59]:
```
# find the standard deviations along the 1st axis
```

4.2 Array-wise operations

These are faster than for loops. Lets start with an example

4.2.1 Summing two arrays

we will add each element of the arrays together in this example, but the same thing works with multiplication, division, raising to a power, etc.

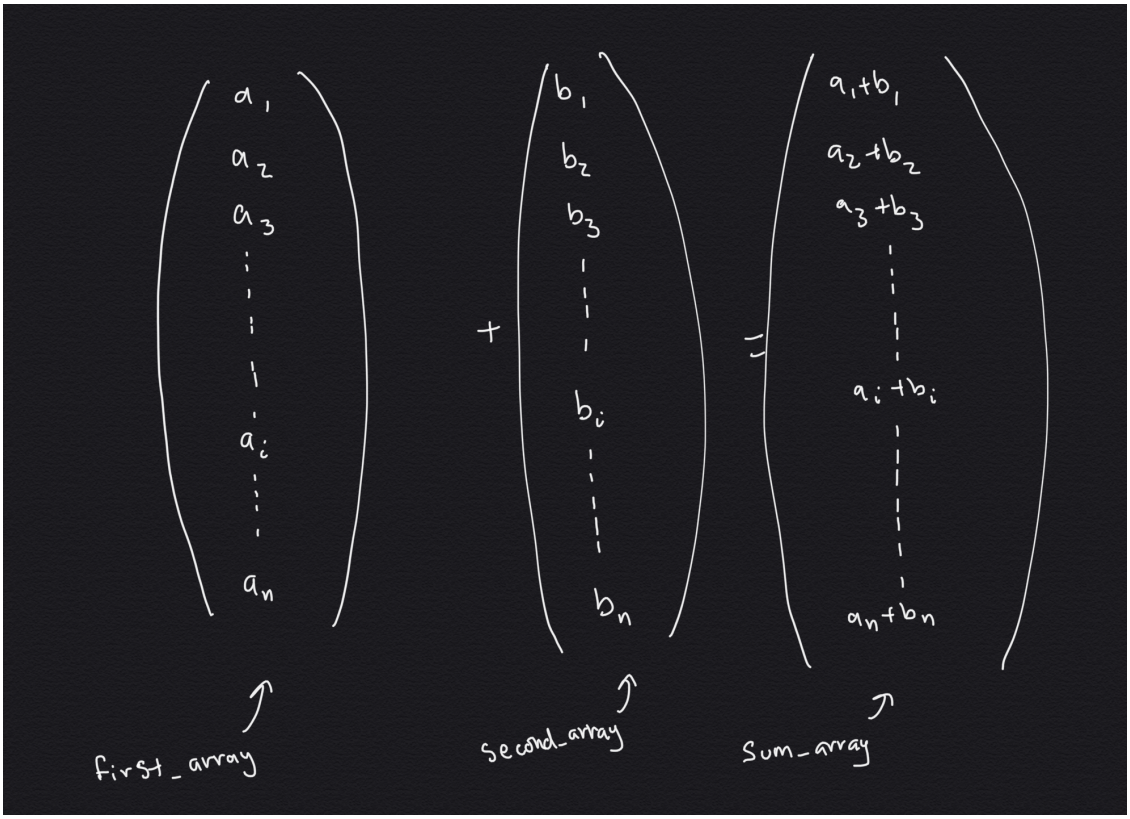
[60]:
```
#define two arrays
first_array = np.arange(10)
second_array = np.arange(30,40)
```

[ ]:
```
#you try: make a new array called sum_array that adds each element of␣
 ↪first_array to the same element in second_array.
#lets do this using a for loop and indexing, I'll get it started for you

#this creates an array with the same dimensions as first_array, but filled with␣
 ↪zeros. We will fill this in the for loop
sum_array = np.zeros_like(first_array)

#fill in sum_array, element by element
for i in range(len(first_array)):
    the_sum = #fill me in!
    sum_array[i] = the_sum
print(sum_array)
```

that was a lot of code! lets try to do it better with numpy. Since 1D numpy arrays act like vectors, we can add them together with the + operator and it will add each component (element) of array 1 to array 2 like this:



[62]:
```
sum_array = first_array + second_array
print(sum_array)
#check to make sure the output is the same as when we used the for loop
```

```
[30 32 34 36 38 40 42 44 46 48]
```

this was a lot less code, and it also takes less computing time. you can't tell with small arrays like these, but when you are working with large datasets, doing things in arrays instead of loops can save you hours of computing time.

Lists don't have these same capabilites, so its best to use numpy arrays

[63]:
```
#lists don't do this! they use the + operator to "concatinate", which is rarely␣
 ↪what we want
list1 = [56,7]
list2 = [33,2]
list1+list2
```

[63]:
```
[56, 7, 33, 2]
```

4.2.2 Size matching for array-wise operations

quick note: array-wise operations like addition, subtraction, multiplication, and division (+-*/) all need to happen between arrays of the same length, or between an array and a scalar.

[64]:
```
#try to add arrays of different sizes
third_array = np.arange(60)
first_array + third_array #this returns an error
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-64-54fab041c471> in <module>
      1 #try to add arrays of different sizes
      2 third_array = np.arange(60)
----> 3 first_array + third_array #this returns an error

ValueError: operands could not be broadcast together with shapes (10,) (60,)
```

[65]:
```
#try to multiply an array and a scalar
first_array*5 #this works fine!
```

[65]:
```
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45])
```

There are many more array-wise operations that are very useful that we don't have time for today. I'd encourage you to check some out here or anywhere else on the web.

4.3 Linear Algebra

In numpy, 1d arrays are like vectors and 2d arrays are like matricies. We can do simple linear algebra operations on them easily. This comes in handy when you need to solve systems of equations, approximate integrals, and otherwise combine multiple arrays. Lets check out some of the basic capabilites

[ ]:
```
#make a 3x4 2d array of your choosing:
arr_linalg = #fill me in!
```

4.3.1 Matrix Properties

- Transpose: flip axes 0 and 1 of array `arr` with command `arr.T`
- Diagonal: list the elements on the diagonal of a matrix `arr` with `arr.diagonal()`
- Trace: list the sum of the diagonal elements with `arr.trace()`

[ ]:
```
print(arr_linalg)
#transpose the matrix arr_linalg
#before you run the cell, try to predict what the outcome will be.
print(arr_linalg.T)
```

[68]:
```
#now you try: return the diagonal of the matrix
```

[69]:
```
#now you try: return the trace of the matrix. Check and see if this is the sum␣
↪of the diagonal components
```

4.3.2 Dot Products

[ ]:
```
vec_2 = np.random.uniform(size=5)
vec_1.dot(vec_2)
```

4.3.3 Matrix multiplication

Matrix multiplication is really annoying and tedious if you ever have to do it, so its nice that numpy has built-in functions to do it for you!

[71]:
```
#make two 5x5 2D arrays
arr_1 = np.arange(25).reshape(5,5)
arr_2 = np.arange(25,50).reshape(5,5)

#and one length 5 1d array
vec_1 = np.arange(5)
```

[72]:
```
vec_1
```

[72]:
```
array([0, 1, 2, 3, 4])
```

[73]:
```
print('array 1 matrix multiplied with array 2: ', np.matmul(arr_1,arr_2))
print('vector 1 matrix multiplied with array 1: ', np.matmul(vec_1,arr_1))
```

```
array 1 matrix multiplied with array 2:  [[ 400  410  420  430  440]
 [1275 1310 1345 1380 1415]
 [2150 2210 2270 2330 2390]
 [3025 3110 3195 3280 3365]
 [3900 4010 4120 4230 4340]]
vector 1 matrix multiplied with array 1:  [150 160 170 180 190]
```

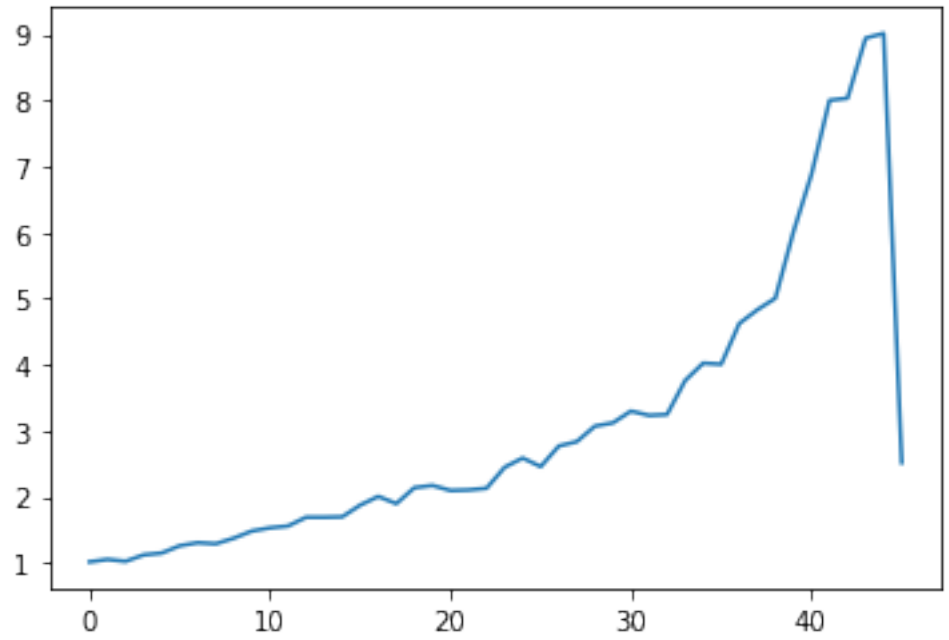
5. Homework: Slicing and Dicing

1. load in `'GBM_sensitivity.csv'`, located under `loaddir` and save it as a numpy array. This dataset contains the sensitivity of a satellite to the brightness of faraway stars exploding as a function of the distance to those explosions
2. print last 10 items (slice the array) - skip this if we didn't get to slicing today
3. This dataset just has the sensitivities, but these sensitivites are a function of distance to the explosion. Pretend you know that each sensitivity measurement was taken at evenly spaced intervals between 50 Mpc and 47800 Mpc (Megaparcecs are confusing measurements of distance that astrophysicists use a lot - you dont need to know what a Mpc is to do this problem). Make a 1d numpy array called `distances` that is the same length as your dataset and contains evenly spaced numbers between 50 and 47800.
4. print the elements of the your dataset with distances between `1e4` and `1e5` using Boolean indexing
5. find the mean distance of points with sensitivity greater than `5e50` with Boolean indexing and `np.mean()`

[ ]:
```
#blank cell
```

[74]:
```
plt.plot(data[:,0])
```

[74]:
```
[<matplotlib.lines.Line2D at 0x7fe0c6a15af0>]
```



5.1 Answer:

[77]:
```
sens_curve  = np.genfromtxt(loaddir+'GBM_sensitivity.csv',delimiter=',')

print('last 10 elements:')

print(sens_curve[:-10])

#create distances
distances = np.linspace(50,47800,num=len(sens_curve))
print('distances: ',distances)

print('distance between 1e4 and 1e5')

print(sens_curve[np.logical_and(distances>1e4,distances<1e5)])

print('mean distance of points with sensitivity greater than 5e50')

print(np.mean(sens_curve[sens_curve>5e50],axis=0))
```

```
last 10 elements:
[1.08322750e+47 1.65510738e+47 2.52912654e+47 3.60148034e+47
```

```
 4.94981922e+47 7.83606507e+47 9.68699713e+47 1.37930964e+48
 1.89586736e+48 2.34358194e+48 3.33711894e+48 5.09980959e+48
 7.13496370e+48 1.14977017e+49 1.75724049e+49 2.68566207e+49
 3.82488771e+49 5.64327714e+49 8.62635494e+49 1.27285198e+50
 1.74999880e+50 2.49297971e+50 3.81145212e+50 5.62541694e+50
 8.60430685e+50 1.22605669e+51 1.74750684e+51 2.32038851e+51
 3.30784246e+51 4.09292510e+51 5.43517471e+51 6.49224669e+51
 8.03451699e+51 9.94402994e+51 1.23062911e+52 1.47074042e+52
 1.75693390e+52 2.02595339e+52]
distances:  [   50.          1065.95744681  2081.91489362  3097.87234043
  4113.82978723  5129.78723404  6145.74468085  7161.70212766
  8177.65957447  9193.61702128 10209.57446809 11225.53191489
 12241.4893617  13257.44680851 14273.40425532 15289.36170213
 16305.31914894 17321.27659574 18337.23404255 19353.19148936
 20369.14893617 21385.10638298 22401.06382979 23417.0212766
 24432.9787234  25448.93617021 26464.89361702 27480.85106383
 28496.80851064 29512.76595745 30528.72340426 31544.68085106
 32560.63829787 33576.59574468 34592.55319149 35608.5106383
 36624.46808511 37640.42553191 38656.38297872 39672.34042553
 40688.29787234 41704.25531915 42720.21276596 43736.17021277
 44752.12765957 45768.08510638 46784.04255319 47800.        ]
distance between 1e4 and 1e5
[3.33711894e+48 5.09980959e+48 7.13496370e+48 1.14977017e+49
 1.75724049e+49 2.68566207e+49 3.82488771e+49 5.64327714e+49
 8.62635494e+49 1.27285198e+50 1.74999880e+50 2.49297971e+50
 3.81145212e+50 5.62541694e+50 8.60430685e+50 1.22605669e+51
 1.74750684e+51 2.32038851e+51 3.30784246e+51 4.09292510e+51
 5.43517471e+51 6.49224669e+51 8.03451699e+51 9.94402994e+51
 1.23062911e+52 1.47074042e+52 1.75693390e+52 2.02595339e+52
 2.42018656e+52 2.79198013e+52 3.21948460e+52 3.71406782e+52
 3.99055341e+52 4.76832877e+52 5.30802238e+52 5.70316641e+52
 6.57700183e+52 6.82425654e+52]
mean distance of points with sensitivity greater than 5e50
2.248146852242842e+52
```