Introduction to NumPy

numpy, pandas, matplotlib, seaborn
tidyverse
tidyverse
Look what they need to mimic a fraction of our power
imgflip.com

- NumPy is an acronym for "Numerical Python" and is the fundamental Python package for **high performance computing** and data analysis.

- At the core of the NumPy package is the *ndarray* object, which stands for "*n*-dimensional array." There are several important differences between Numpy arrays and standard Python lists:

  - Numpy arrays utilize contiguous memory allocation and have a fixed size at creation. This is in contrast to Python lists, which can grow dynamically. Note that changing the size of an *ndarray* will create a new array and delete the original.

  - The elements in a Numpy array are all **required to be of the same data type**, and thus will be the same size in memory. This requirement forms the foundation for speedy computations.

    - Similar to vectors vs. lists in R.
    - Speed is the key!

  - In this chapter we only provide an overview and suggest that you check out the Numpy documentation, which can be found at: https://numpy.org/

Importing Numpy

- The Numpy package is not part of the standard Python release and may need to be installed separately. If you use pip, you can install Numpy with: (via Terminal)

```
pip install numpy
```

- Once Numpy is installed, you need to import the Numpy library as follows:

```
import numpy as np
```

- It is common practice to import Numpy with the alias '`np`'. This simply makes it easier to access functions as `np.function` as opposed to `numpy.function`.

1D Arrays and Vectorization

- Before we learn about how to create Numpy arrays, let's take a moment to understand a limitation of Python lists. Consider the following Python code that converts a list of weights from lbs. to kg and a list of heights from inches to meters:

```
# Create Python lists of height (inches)
height = [72, 68, 69, 68, 64, 72, 72]

# Convert height to meters
height_m = [0 for _ in range(len(height))] # just creates an empty list

for i in range(len(height)):
    height_m[i] = height[i]*0.0254

print(height_m)
```

- This produces the following output.

```
[1.8288, 1.7271, 1.7526, 1.7271, 1.6256, 1.8288, 1.8288]
```

- Numpy arrays give us a more elegant way of accomplishing this using *vectorization*, which could be defined as follows:

Def: ***Vectorization*** is the practice of replacing explicit loops with array expressions. In general, vectorized array operations will often be one to two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact seen in any kind of numerical computations.

- Before we learn about to use vectorization, we first need to learn how to create Numpy arrays.

- Numpy arrays can be created in several ways, but the easiest method is to use the `array` function to turn a Python list into an array:

```
np.array(list, type)
```

- Now consider the following Python code that creates a Numpy version of the weight list and converts the list to kg.

```
import numpy as np

# Create Python list of height (inch)
height = [72, 68, 69, 68, 64, 72, 72]

# Create Numpy version of height
np_height = np.array(height)

# Convert height to meters
np_height_m = np_height*0.0254

print(np_height_m)
```

- This produces the following output, which is identical to the previous version:

```
[1.8288, 1.7271, 1.7526, 1.7271, 1.6256, 1.8288, 1.8288]
```

- Not only is the use of vectorization easier than using a for-loop, it is also much faster because all of the elements of a Numpy array are of a single type. (less code + speed)

- It is also important to realize that Numpy array arrays behave differently than Python lists. For example, consider this:

```
import numpy as np

python_list1 = [2,4,6]
python_list2 = [1,3,5]

numpy_list1 = np.array(python_list1)
numpy_list2 = np.array(python_list2)

print("Adding the Python lists gives " + str(python_list1 + python_list2))
print("Adding the Numpy lists gives " + str(numpy_list1 + numpy_list2))
```

```
Adding the Python lists gives [2, 4, 6, 1, 3, 5]
Adding the Numpy lists gives [ 3  7 11]
```

- You can see that when adding the Python lists the result was to concatenate the lists, while the result when adding the Numpy arrays is to perform the additions elementwise.

- It is very important that you pay attention to the Python type you are dealing with because the behavior and outcome can be different.

Changing the Datatype

- `np.array()` has an additional parameter of `dtype` that can be used to define whether the elements are integers, floating point numbers, or complex numbers.

- Consider the following example that makes use of the `dtype` attribute of the Numpy array.

```
numbers = [10, 15, 20, 35]
numbers2 = [10.0, 15.1, 19.5,]

x = np.array(numbers)
y = np.array(numbers, dtype = "float")
z = np.array(numbers2)

print("The data type of x is " + str(x.dtype))
print("The data type of y is " + str(y.dtype))
print("The data type of z is " + str(z.dtype))
```

```
The data type of x is int32
The data type of y is float64
The data type of z is float64
```

- This implies that `x` is a 32-bit integer, while `y` is a 64-bit floating-point number (because we included the additional parameter `dtype = "float"`). Note that `z` is of type `float64` without having to include the parameter `dtype = "float"` because Python automatically recognized that these should be floats because of the included decimal point.

Indexing and Slicing in 1D Arrays

- It is important to remember that Python indexing starts from 0.

- Consider the indexing syntax below:

Syntax	Result
`x[start:end]`	Elements in `x` indexed from `start` to `end` (but `end` is not included).
`x[start:end:step]`	Elements in `x` indexed from `start` to `end` using the step size `step`. Note that `end` is not included, and the default step value is 1.
`x[start:]`	Elements in x indexed from `start` through the end of the array.
`x[:end]`	Elements in x indexed from the beginning through `end` (but `end` is not included).

- We illustrate these below.

```
age = np.array([18,23,89,11,35,37,26,52,76])
print("The array age is " + str(age))
print("The result of age[2] is " + str(age[2]))
print("The result of age[2:5] is " + str(age[2:5]))
print("The result of age[:3] is " + str(age[:3]))
print("The result of age[4:] is " + str(age[4:]))
print("The result of age[2:7:2] is " + str(age[2:5]))
print("The result of age[::-1] is " + str(age[::-1]))
```

```
The array age is [18 23 89 11 35 37 26 52 76]
The result of age[2] is 89
The result of age[2:5] is [89 11 35]
The result of age[:3] is [18 23 89]
The result of age[4:] is [35 37 26 52 76]
The result of age[2:7:2] is [89 11 35]
The result of age[::-1] is [76 52 26 37 35 11 89 23 18]
```

- Note that the last command, `age[::-1]`, reversed the array. However, keep in mind this command does not actually modify the array `age`.

Creating Sequences of Numbers

- You can create sequences of numbers using the `arange` command, which creates an instance of `ndarray` with evenly spaced values and returns the *reference* to it:

```
numpy.arange(start, stop, step)
```

- Below we illustrate the `arange` command.

```
print("The result of np.arange(1,8) is " + str(np.arange(1,8)))
print("The result of np.arange(2,10,2) is " + str(np.arange(2,10,2)))
print("The result of np.arange(2,10.1,2) is " + str(np.arange(2,10.1,2)))
print("The result of np.arange(5) is " + str(np.arange(5)))
print("The result of np.arange(-6,3) is " + str(np.arange(-6,3)))
print("The result of np.arange(5,1,-1) is " + str(np.arange(5,1,-1)))
```

```
The result of np.arange(1,8) is [1 2 3 4 5 6 7]
The result of np.arange(2,10,2) is [2 4 6 8]
The result of np.arange(2,10.1,2) is [  2.   4.   6.   8.  10.]
The result of np.arange(5) is [0 1 2 3 4]
The result of np.arange(-6,3) is [-6 -5 -4 -3 -2 -1  0  1  2]
The result of np.arange(5,1,-1) is [5 4 3 2]
```

- Note that the default step size is 1.

- Can you explain why `np.arange(2,10.1,2)` includes 10, while `np.arange(2,10,2)` does not?

- Note that the single argument command `np.arange(5)` creates a sequence that starts at zero and stops at 4 with an increment of 1, while `np.arange(5,1,-1)` has a step size of −1, and thus counts backwards from 5 to 1.

Indexing with Boolean Arrays

- In Numpy, we also have the ability to do list *subsetting* using Boolean arrays.

- Suppose we wanted to get all the ages above 35. The following syntax will generate a Numpy array containing Booleans, where it is True if the corresponding value is above 35, and False otherwise.

```
age = np.array([18,23,89,11,35,37,26,52,76])
print(age > 35)
```

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

- If we wanted to generate a new Numpy array only containing those values in age that are greater than 35, we can use the Boolean array inside square brackets to do *subsetting*:

```
old = age[age > 35]
print(old)
```

```
[89 37 52 76]
```

- Recall that Python doesn't allow "ternary" comparisons like `35 < age < 55`. Instead, we need to combine comparisons with logical operators **and**, **not**, and **or**, as shown in the table below.

Logical Operator	Interpretation	Interpretation Elaborated
&	"bitwise" and	returns true (1) if both elements are true, and false (0) otherwise
~	"bitwise" not	returns false (0) if the element is true and true (1) if the element is false
|	"bitwise" or	returns true (1) if at least one of the elements are true, and false (0) otherwise

- We illustrate this below.

```
age = np.array([18,23,89,11,35,37,26,52,76])
middle_age=age[(age>35) & (age<55)]
print(middle_age)
```

```
[37 52]
```

2D Arrays

- We can create 2-dimensional Numpy arrays by passing in a list of lists into the Numpy `array` function:

```
height = [72, 68, 69, 68, 64, 72, 72]
weight = [200, 165, 160, 135, 120, 162, 190]
metrics = np.array([height,weight])
print(metrics)
```

```
[[ 72  68  69  68  64  72  72]
 [200 165 160 135 120 162 190]]
```

- We can access the dimension, the size (total number of elements), and the shape using the *attributes* `ndim`, `size`, and `shape`, respectively:

```
print("metrics.ndim = " + str(metrics.ndim))
print("metrics.size = " + str(metrics.size))
print("metrics.shape = " + str(metrics.shape))
```

```
metrics.ndim = 2
metrics.size = 14
metrics.shape = (2, 7)
```

Indexing in 2D Arrays

- You can access the element in the *i*th row and *j*th column in two different ways:

```
print(metrics[0][2])
print(metrics[0,2])
```

```
69
69
```

- Both give the same result, but the second method, `metrics[0,2]`, is more compact.

- We can take slices of two-dimensional arrays as in the following example.

```
A = np.array([[0,1,2],[3,4,5],[6,7,8]])
print(A)
print(A[:2,:])
print(A[:,0:2])
print(A[:,0])
```

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

$\leftarrow$ The two-dimensional array `A`

```
[[0 1 2]
 [3 4 5]]
```

$\leftarrow$ `A[:2,:]` gives rows 0 up to 2 and all columns

```
[[0 1]
 [3 4]
 [6 7]]

[0 3 6]
```

`A[:,0:2]` gives all rows and columns 0 up to 2

`A[:,0]` gives all rows and the first column, note the dimension has dropped by one!

- Essentially, you simply specify a one-dimensional slice for each *axis*. One can also supply a number for an axis rather than a slice but doing so will drop the dimension by one.

Reshaping Arrays

- We can reshape a Numpy array without changing its data using the `reshape(.)` method of the `ndarray`.

```
x = np.arange(1,10)
A = x.reshape(3,3)
print(x)
print(A)
print(x)
```

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

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

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

- Note that the single-dimensional array `x` is reshaped into a 3×3 matrix `A`. Interestingly though, the `reshape(.)` method does not alter the shape of the original array `x`.

- **However, the `resize(.)` method of the `ndarray` does change the shape and size of the array.** For example:

```
x = np.arange(1,10)
A = x.reshape(3,3)
print(x)
x.resize(3,3)
print(x)
```

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

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

- If the dimension is given as −1 in a reshaping, the other dimensions are automatically calculated provided that the given dimension is a multiple of the total number of elements in the array. (like joker)

```
x = np.arange(1,16)
print(x)
B = x.reshape(5,-1)
print(B)
```

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

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

- In the above code we only indicated that we wanted 5 rows and the −1 instructed Python to automatically calculate the number of elements in the other dimension (so that each row has the same number of elements).

- As we will see later in the course, reshaping data is quite common as many of the Python algorithms for performing data analytics require the data to be in a particular shape.

Vectorization in Two-Dimensions

- We can also do vectorization (elementwise calculations) with two-dimensional Numpy arrays. Consider the following dataset, where each row provides the height and weight for a different individual.

Height (in)	**Weight (lbs)**
72	200
68	165
69	160
68	135
64	120
72	162

```
data = np.array([[72,200], [68,165], [69,160], [68,135], [64,120], [72,162]])
print(data)
```

```
[[ 72 200]
 [ 68 165]
 [ 69 160]
 [ 68 135]
 [ 64 120]
 [ 72 162]]
```

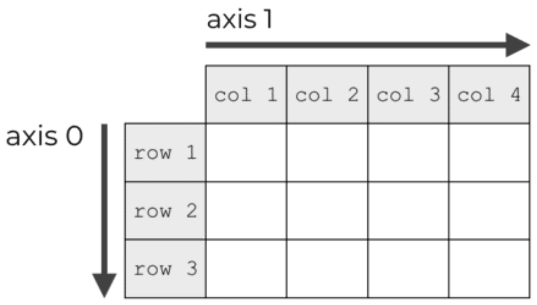
- Suppose we wanted to convert height from inches to meters and weight from lbs. to kg. This can be accomplished by multiplying each height by 0.0254 and each weight by 0.453592.

```
conversion = np.array([0.0254, 0.453592])
newData = data*conversion
print(newData)
```

```
[[ 1.8288   90.7184  ]
 [ 1.7272   74.84268 ]
 [ 1.7526   72.57472 ]
 [ 1.7272   61.23492 ]
 [ 1.6256   54.43104 ]
 [ 1.8288   73.481904]]
```

Understanding Axes Notation

- In Numpy, an *axis* refers to a single dimension of a multidimensional array. In a two-dimensional Numpy array, the axes are the directions along the rows and columns.



Axis 0 is the axis that runs downward along the rows, while axis 1 is the axis that runs horizontally across the columns.

- The following array has 2 axes. The first axis has length 2 and the second axis has a length of 3.

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

- As an example for how Numpy uses axes, consider the Numpy `sum` function, which returns the sum of array elements over the specified axis:

```
numpy.sum(arr, axis,…)
```

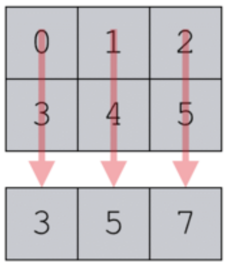
- Here, the first two parameters are `arr` (the input array) and `axis` (the axis along which we want to calculate the sum).

- Consider the following example:

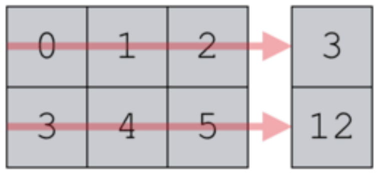
```
arr = [[14, 17, 12, 33, 44],
       [15, 6, 27, 8, 19],
       [23, 2, 54, 1, 4,]]

print('The sum of all values is ' + str(np.sum(arr)))
print()
print('The column sums are ' + str(np.sum(arr, axis=0)))
print()
print('The row sums are ' + str(np.sum(arr, axis=1)))
print()
```

```
The sum of all values is 279

The column sums are [52 25 93 42 67]

The row sums are [120  75  84]
```

- Note that when we set `axis=0`, the function sums down the columns. The axis parameter controls which axis will be *aggregated* or *collapsed*. So, when we set `axis=0` we are collapsing the rows:



- When we set `axis=1`, we collapse the columns:



Basic Statistics in Numpy

- Numpy is equipped with a number of statistical functions, including the following (see the Numpy documentation for the full suite of statistical functions).

Function	**Result**
`numpy.amin(.)`	Determines the minimum value of the element along a specified axis.
`numpy.amax(.)`	Determines the maximum value of the element along a specified axis.
`numpy.mean(.)`	Determines the mean value
`numpy.median(.)`	Determines the median value
`numpy.std(.)`	Determines the standard deviation
`numpy.var(.)`	Determines the variance
`numpy.average(.)`	Determines a weighted average

`numpy.percentile(.)`	Determines the *n*th percentile along an axis

- Consider the following example.

```
A = np.array([[31,64,70],[79,94,10],[49,91,60]])

print('Our array is:')
print(A)
print()
print('The median of all values is: ' + str(np.median(A)))
print()
print('The median of the first column is: ' + str(np.median(A[:,0])))
print()
print('The median of the second row is: ' + str(np.median(A[1,:])))
print()
print('The median along the first axis is: ' + str(np.median(A,axis=0)))
print()
print('The median along the second axis is: ' + str(np.median(A,axis=1)))
```

```
Our array is:
[[31 64 70]
 [79 94 10]
 [49 91 60]]

The median of all values is: 64.0

The median of the first column is: 49.0

The median of the second row is: 79.0

The median along the first axis is: [ 49.  91.  60.]

The median along the second axis is: [ 64.  79.  60.]
```

- Interestingly, there are `ndarray` *methods* that yield the same functionality as the Numpy functions. We demonstrate below:

```
list = np.array([10,-3,5,6,11,52,9,23,41,22])

print(list)
print('list.mean() is ', list.mean())
print('list.std()', list.std())
print('list.max()', list.max())
print('list.sum()', list.sum())

print()
array = np.array([[14, 17, 12, 33, 44],
       [15, 6, 27, 8, 19],
       [23, 2, 54, 1, 4,]])

Print(array)
print('The column means are', array.mean(axis=0))
print('The row means are', array.mean(axis=1))
```

```
[10 -3  5  6 11 52  9 23 41 22]
list.mean() is  17.6
list.std() 16.3474768695
list.max() 52
list.sum() 176

[[14 17 12 33 44]
 [15  6 27  8 19]
 [23  2 54  1  4]]
The column means are [ 17.33333333   8.33333333  31.          14.
22.33333333]
The row means are [ 24.   15.   16.8]
```

- You may be wondering which technique should you use, the `ndarray` methods or the Numpy functions. Unfortunately, there are no firm rules as to which method to employ. This is ultimately up to you, but you should use the method that is more readable in your code.

- We encourage you to explore the methods available to Numpy `ndarray.`

Missing Data

- In the real world, data is rarely clean and typically has some amount of missing data. To make matters even more complicated, different data sources may indicate missing data in different ways. Here we will discuss how Numpy chooses to represent missing numerical data.

- Numpy uses the data representation `NaN`, which is an acronym for Not a Number). This is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation. Consider the following:

```
data = np.array([2, np.nan, -1, 3])
print(data.dtype)
```

```
float64
```

- Note that Numpy chose a floating-point type for this array. You should be aware that `NaN` is a bit like a data-virus that infects any other object it touches. **That is, regardless of the operation, the result of arithmetic with `NaN` will be another `NaN`.**

```
print('2 + np.nan yields ' + str(2 + np.nan))
print('4*np.nan yields ' + str(4*np.nan))
print('The sum of the data array is ' + str(np.sum(data)))
```

```
2 + np.nan yields nan
4*np.nan yields nan
The sum of the data array is nan
```

- Note that we can use Numpy's `nansum(.)` function to return the sum of array elements over a given axis treating `NaN` as zero:

```
print('The sum of data array, ignoring NaN is ' + str(np.nansum(data)))
```

```
The sum of data array, ignoring NaN is 4.0
```

- To check whether an array contains missing values, we can use the `isnan(.)` function, which returns a Boolean array.

```
print('The result of np.isnan(data) is ' + str(np.isnan(data)))
```

```
The result of np.isnan(data) is [False  True False False]
```

Higher-Dimensional Arrays?

- There are a number of situations where you might see data with greater than two dimensions.

- *Panel data* is multi-dimensional data involving measurements over time. For example, data that tracks a cohort (group) of individuals over time could be structured as (`subjects, dates, attributes`).

- Another example is color-image data for multiple images, which is typically stored in four dimensions. Each image is a three-dimensional array of (height, width, channels), where channels are usually red, green, and blue (RGB) values. A collection of images is then just (`image_number, height, width, channels`).

- We can manually create a 3d array by passing in a list of lists of lists as follows:

```
threeD = np.array([[[10,20,30], [40,50,60]], [[15,25,35], [45,55,65]],
[[70,80,90],[75,85,95]]])
print(threeD)
```

```
[[[10 20 30]
  [40 50 60]]

 [[15 25 35]
  [45 55 65]]

 [[70 80 90]
  [75 85 95]]]
```

- We can obtain the dimension, total number of elements, and shape using the `ndim`, `size`, and `shape` attributes:

```
print("The dimension is " + str(threeD.ndim))
print("The total number of elements is " + str(threeD.size))
print("The shape is " + str(threeD.shape))
```

```
The dimension is 3
The total number of elements is 18
The shape is (3, 2, 3)
```

- We further illustrate below—can you follow along?

```
matrix3D = np.arange(36).reshape(3, 4, 3)
print(matrix3D)
print("matrix3D[1,3,2] = " + str(matrix3D[1,3,2]))
print("matrix3D[0,2] = " + str(matrix3D[0,2]))
print("matrix3D[0] = " + str(matrix3D[0]))
print("matrix3D[0,:,1] = " + str(matrix3D[0,:,1]))
print("matrix3D[0,:,1].reshape(4,1) = " + str(matrix3D[0,:,1].reshape(4,1)))
```

A single command created the 3d array of size 3×4×3 containing the numbers 0 through 35

This matrix is `A[0,.,.]`

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

This matrix is `A[1,.,.]`

```
 [[12 13 14]
  [15 16 17]
  [18 19 20]
  [21 22 23]]
```

This matrix is `A[2,.,.]`

```
 [[24 25 26]
  [27 28 29]
  [30 31 32]
  [33 34 35]]]
```

This accesses the single element at the location indexed by [1,3,2]

```
matrix3D[1,3,2] = 23
```

This extracts the row indexed by 2 in $A_{0\cdot\cdot}$

```
matrix3D[0,2] = [6 7 8]
```

This extracts the matrix $A_{0\cdot\cdot}$

```
matrix3D[0] = [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

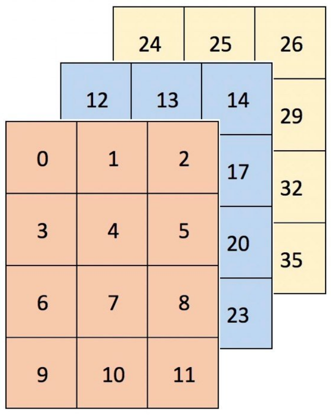
This extracts the column indexed by 1 in matrix $A_{0\cdot\cdot}$ Note that Python returns it as a “row”

```
matrix3D[0,:,1] = [ 1  4  7 10]
```

This reshapes the previous answer as a column

```
matrix3D[0,:,1].reshape(4,1) = [[ 1]
 [ 4]
 [ 7]
 [10]]
```

- Visual, the array matrix3D can be thought of as a container of three 4×3 grids, or a rectangular prism, and would look like this:



- Higher dimensional arrays can be tougher to picture, but they will still follow this "arrays within an array" pattern.

Broadcasting

- The following discussion is adapted from a blog post of Ayoosh Kathuria and the Python Data Science Handbook by Jake VanderPlas.

- Broadcasting is one of the best features of `ndarrays` as it allows you to perform arithmetic between `ndarrays` of different sizes.

- Numpy's operations are usually done element-by-element, which requires two arrays to have the exact same shape. For example, suppose we wanted to multiple every element of an array by 2. It appears that we would need to do something like what is illustrated in the example below.

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])

print('a = ', a)
print('b = ', b)
print('The result of a*b is', a*b)
```

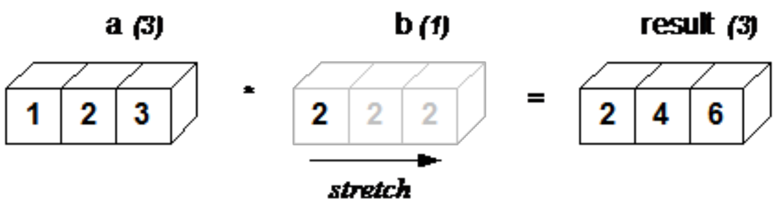
```
a =  [ 1.  2.  3.]
b =  [ 2.  2.  2.]
The result of a*b is [ 2.  4.  6.]
```

- Numpy's broadcasting rule **relaxes** the constraint of requiring the same shape when the array shapes meet certain criteria. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation. For example, consider the following code that accomplishes what we did in the last example.

```
a = np.array([1.0, 2.0, 3.0])
b = 2

print('a = ', a)
print('b = ', b)
print('The result of a*b is', a*b)
```

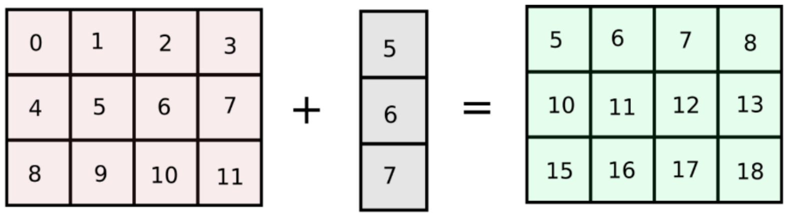
```
a =  [ 1.  2.  3.]
b =  2
The result of a*b is [ 2.  4.  6.]
```

- Note that the result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` being **stretched** or duplicated during the arithmetic operation into an array with the same shape as `a`, as illustrated in the figure below.



- The stretching analogy is only conceptual as Numpy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

- As another example, suppose we have a two-dimensional array of shape `(3, 4)` containing 3 rows and 4 columns. Further suppose that we want to add a column to each of the columns in our array. We graphically illustrate what we are trying to achieve below:



- As with most tasks in Python, there are a number of different ways we could accomplish this. For example, we could loop over the columns of the matrix (two-dimensional array) and add the column.

```
matrix = np.arange(12).reshape(3,4)
vector = np.array([5,6,7])

print(matrix)
print(vector)

num_cols = matrix.shape[1]

for col in range(num_cols):
      matrix[:, col] += vector

print()
print('The updated matrix is')
print(matrix)
```

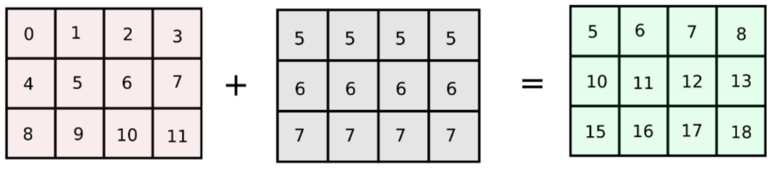
The number of columns

Note that `matrix[:, col]` returns the column as a row!

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

[5 6 7]

The updated matrix is
[[ 5  6  7  8]
 [10 11 12 13]
 [15 16 17 18]]
```

- While this accomplishes our goal, if the number of columns in our original array matrix are increased to a very large number, the code described will run slow as we are looping over the number of columns.

- Another way to accomplishing this would be to make a matrix of equal size as the original matrix with identical columns (which we will refer to as the column-stacking approach). This is illustrated and coded below.



```
matrix = np.arange(12).reshape(3,4)
vector = np.array([5,6,7])

add_matrix = np.tile(vector,(4,1)).T

print(matrix)
print(add_matrix)

matrix += add_matrix

print()
print('The updated matrix is')
print(matrix)
```

Here we are using Numpy's `tile(.)` function and the transpose operator—take a moment to learn about this on the web.

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

[[5 5 5 5]
 [6 6 6 6]
 [7 7 7 7]]

The updated matrix is
[[ 5  6  7  8]
 [10 11 12 13]
 [15 16 17 18]]
```

- This approach is much faster. However, while it worked well in the two-dimensional case, applying the same approach with higher dimensional arrays can be tricky.

- The good news is that we can use broadcasting to perform arithmetic operations on arrays of unequal sizes. According to the Numpy documentation

  Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python.

- Under the hood, Numpy does something similar to our column-stacking approach. However, we do not have to worry about stacking arrays in multiple directions explicitly.

- According to the Numpy documentation, the rule governing whether two arrays have compatible shapes for broadcasting can be expressed in a single sentence:

The Broadcasting Rule:

In order to broadcast, the size of the *trailing axes* for both arrays in an operation must either be the same size of one of them must be one.

- However, the Python Data Science Handbook offers a more comprehensive set of rules:

Rules of Broadcasting:

Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

- The size of the result array created by broadcast operations is the maximum size along each dimension from the input arrays. In the case the arrays are not compatible, you will get a `ValueError`.

- Let's revisit our previous example and see broadcasting in action.

```
matrix = np.arange(12).reshape(3,4)
vector = np.array([5,6,7]).reshape(3,1)


print(matrix)
print(vector)

matrix += vector

print()
print('The updated matrix is')
print(matrix)
```

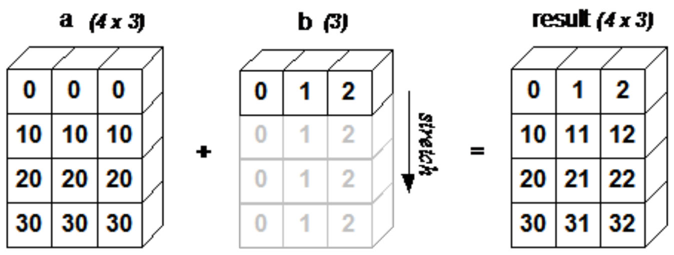
```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[5]
 [6]
 [7]]

The updated matrix is
[[ 5  6  7  8]
 [10 11 12 13]
 [15 16 17 18]]
```

- Consider the following example adapted from the Numpy documentation.

```
a = np.array([[ 0.0,  0.0,  0.0],
             [10.0, 10.0, 10.0],
             [20.0, 20.0, 20.0],
             [30.0, 30.0, 30.0]])

b = np.array([0, 1, 2])

print('a =')
print(a)
print('b = ', b)
print('a+b =')
print(a+b)
```

```
a =
[[  0.   0.   0.]
 [ 10.  10.  10.]
 [ 20.  20.  20.]
 [ 30.  30.  30.]]

b =  [0 1 2]

a+b =
[[  0.   1.   2.]
 [ 10.  11.  12.]
 [ 20.  21.  22.]
 [ 30.  31.  32.]]
```

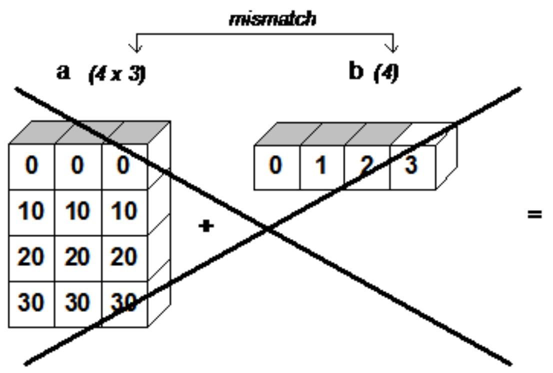
- You can see that b is added to each row of a, as illustrated below.



- If b is longer than the rows of a, as illustrated in the code and figure below, an exception is raised because of the incompatible shapes.

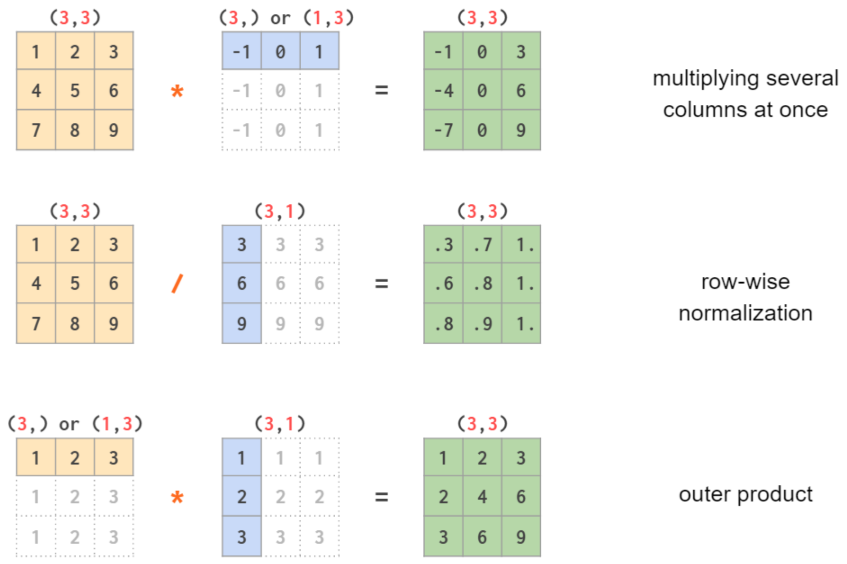
```
a = np.array([[ 0.0,  0.0,  0.0],
             [10.0, 10.0, 10.0],
             [20.0, 20.0, 20.0],
             [30.0, 30.0, 30.0]])

b = np.array([0, 1, 2,3])

print('a =')
print(a)
print('b = ', b)
print('a+b =')
print(a+b)
```

```
a =
[[  0.    0.    0.]
 [ 10.   10.   10.]
 [ 20.   20.   20.]
 [ 30.   30.   30.]]

b =  [0 1 2 3]

a+b =
Traceback (most recent call last):
  File "OneDExamples.py", line 279, in <module>
    print(a+b)
ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```



- Can you reshape b and make the operation work?
- When the trailing dimensions of the arrays are unequal, broadcasting fails because it is impossible to align the values in the rows of the 1st array with the elements of the 2nd arrays for element-by-element addition.

- (skipped) The website TowardDataScience has a nice graphical illustration of some different arithmetic operations in 2d:



- In 3d and above the broadcasting is even less intuitive and using it requires knowledge of the broadcasting rules in their generic forms.

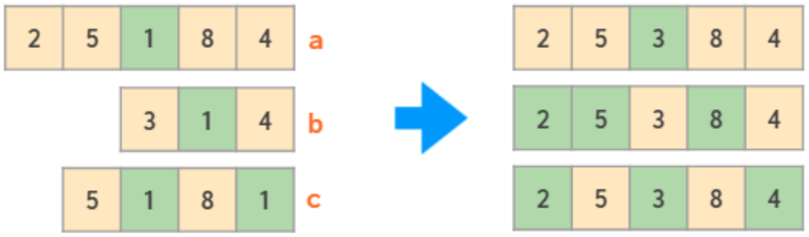
- As a final example, consider the following Python code.

```
a = np.ones((2,5,1,8,4))
b = np.ones((3,1,4))
c = np.ones((5,1,8,1))

d = a+b+c
print('The shape of d is ', d.shape)
```

Note that the `np.ones(.)` function returns an `ndarray` of given shape where all the elements are equal to 1.

- Can you guess the shape of `d`?

- Applying the broadcasting rules, the arrays `b` and `c` are stretched as illustrated below.



- Thus, the final size/shape is `(2, 5, 3, 8, 4)` as indicated by the Python output:

```
The shape of d is  (2, 5, 3, 8, 4)
```

- As we will see throughout this course, broadcasting is going to be a core technique needed to efficiently use Python for data science.

Mathematical Functions in Numpy

Matlab has a number of built-in math functions:

Function	**Result**
`abs(x)`	Absolute value of `x`
`sqrt(x)`	Square root of `x`
`log(x)`	Natural log of `x`
`log10(x)`	Computes $\log_{10}$(`x`)
`log2(x)`	Computes $\log_2$(`x`)
`exp(x)`	Exponential: e^x
`round(x)`	Rounds `x` to the nearest integer
`floor(x)`	Rounds `x` to the nearest integer toward $-\infty$
`ceil(x)`	Rounds `x` to the nearest integer toward ∞
`sign(x)`	Returns a value of -1 if `x` < 0, returns a value of 0 if `x` $= 0$, and returns a value of $+1$ if `x` > 0.
`mod(a,m)`	Modulo operator: remainder after division of `a` by `m`.

Below is a Python session illustrating some of these functions in action.

```
>>> np.sqrt(85)
9.2195444572928871

>>> x = np.array([4,9,25])
>>> np.sqrt(x)
array([ 2.,  3.,  5.])

>>> np.round(8.4)
8.0

>>> np.floor(-8.6)
-9.0

>>> np.floor(8.6)
8.0

>>> np.sign(-23)
-1

>>> np.mod(25,4)
1
```

Notice that Numpy used vectorization here to compute the square root of each entry of `x`.

- The trigonometric functions `sin`, `cos`, and `tan` all assume that angles are represented in radians.

- The trigonometric functions `sin`, `cos`, and `tan` all assume that angles are represented in radians. The value of π is a built-in constant, `np.pi`, in Python. Keep in mind that π is not a machine number (it cannot be represented exactly as a floating-point number) and therefore the constant `pi` in Python is only an approximation. Usually, this is not important; however, you may notice some surprising results—for example:

```
>>> np.sin(np.pi)

1.2246467991473532e-16
```

Trigonometric Functions (radians)

Function	**Result**
`sin(x)`	Sine of `x` in radians
`sinh(x)`	Hyperbolic sine of `x` in radians
`arcsin(x)`	Inverse sine of `x` in radians
`cos(x)`	Cosine of `x` in radians
`cosh(x)`	Hyperbolic cosine of `x` in radians
`arccos(x)`	Inverse cosine of `x` in radians
`tan(x)`	Tangent of `x` in radians
`tanh(x)`	Hyperbolic tangent of `x` in radians
`arctan(x)`	Inverse tangent of `x` in radians

- Numpy has built-in functions, `rad2deg` and `deg2rad`, to convert an angle from radians to degrees and degrees to radians, respectively.

- In the Python session below, we illustrate some of these functions in action.

```
>>> np.rad2deg(np.pi)
180.0

>>> np.deg2rad(90)
1.5707963267948966

>>> x = np.array([90, 180, 360])
>>> np.deg2rad(x)
array([ 1.57079633,  3.14159265,  6.28318531])
```

Wrapping Up

- There is a lot more to learn about Numpy, but this is a good solid start.

- We will learn more about Numpy as we need it. However, we do suggest that you check out the official Numpy documentation and this great guide available from BetterProgramming.