Lecture 11

Operations

Use `+`, `-`, `*`, `/` and `**` to perform element wise addition, subtraction, multiplication, division and power.

In [2]:
```
import numpy as np
```

In [3]:
```
x=np.array([1,2,3])
y=np.array([4,5,6])
```

In [4]:
```
print(x + y) # elementwise addition     [1 2 3] + [4 5 6] = [5  7  9]
print(x - y) # elementwise subtraction  [1 2 3] - [4 5 6] = [-3 -3 -3]
```

```
[5 7 9]
[-3 -3 -3]
```

In [5]:
```
print(x * y) # elementwise multiplication  [1 2 3] * [4 5 6] = [4  10  18]
print(x / y) # elementwise divison         [1 2 3] / [4 5 6] = [0.25  0.4  0.5]
```

```
[ 4 10 18]
[0.25 0.4  0.5 ]
```

In [6]:
```
print(x**2) # elementwise power  [1 2 3] ^2 =  [1 4 9]
```

```
[1 4 9]
```

Dot Product:

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = x_1 y_1 + x_2 y_2 + x_3 y_3$$

In [7]:
```
x.dot(y) # dot product  1*4 + 2*5 + 3*6
```

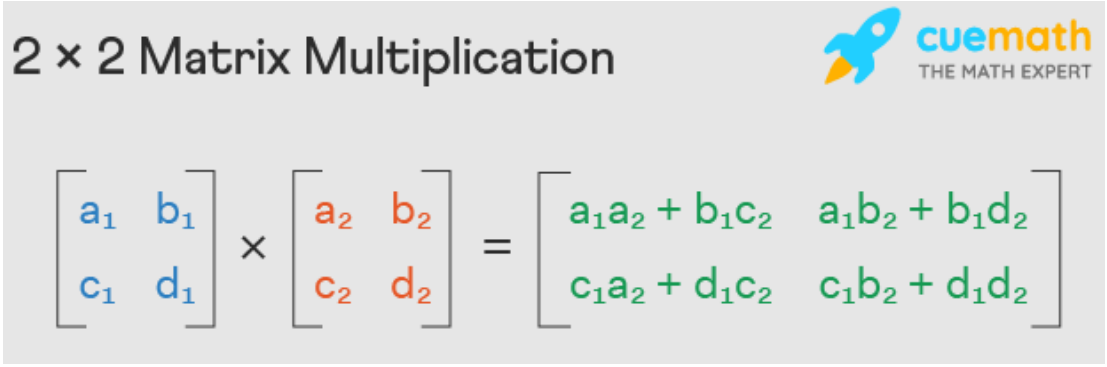
Out[7]:
```
32
```

In [8]:
```
x @ y
```

Out[8]:
```
32
```

In [10]:
```
np.matmul(x,y)
```

Out[10]:
```
32
```



In [12]:
```
a @ b
```

Out[12]:
```
array([[ 28,  31],
       [100, 112]])
```

In [170…
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
a * b
```

Out[170]:
```
array([[ 5, 12],
       [21, 32]])
```

In [11]:
```
a = np.arange(0,6).reshape(2,3)
b = np.arange(6,12).reshape(3,2)
print(a)
print(b)
```

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

In [171…
```
a @ b
```

Out[171]:
```
array([[19, 22],
       [43, 50]])
```

An inverse matrix is a matrix multiplied by which the original matrix A results in the identity matrix E:

$$AA^{-1} = A^{-1}A = E.$$

In [13]:
```
c = np.array([[1, 2, 7], [5, 6, 8],[6, 7, 3]])
c_inv = np.linalg.inv(c)
c_inv @ c
```

Out[13]:
```
array([[ 1.00000000e+00,  1.77635684e-15,  0.00000000e+00],
       [-1.77635684e-15,  1.00000000e+00, -1.33226763e-15],
       [ 2.22044605e-16,  0.00000000e+00,  1.00000000e+00]])
```

Let's look at transposing arrays. Transposing permutes the dimensions of the array.

In [14]:
```
y = np.array([2, 3, 4])
```

In [15]:
```
z = np.array([y, y**2])
z
```

Out[15]:
```
array([[ 2,  3,  4],
       [ 4,  9, 16]])
```

The shape of array `z` is `(2,3)` before transposing.

In [16]:
```
z.shape
```

Out[16]:
```
(2, 3)
```

Use `.T` to get the transpose.

In [17]:
```
z.T
```

Out[17]:
```
array([[ 2,  4],
       [ 3,  9],
       [ 4, 16]])
```

The number of rows has swapped with the number of columns.

In [18]:
```
z.T.shape
```

Out[18]:
```
(3, 2)
```

Use `.dtype` to see the data type of the elements in the array.

In [19]:
```
z.dtype
```

Out[19]:
```
dtype('int32')
```

Use `.astype` to cast to a specific type.

In [20]:
```
z = z.astype('f')
z.dtype
```

Out[20]:
```
dtype('float32')
```

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

In [22]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

Out[22]:
```
array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

In [23]:
```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

In [24]:
```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
```

In [25]:
```
# concatenate along the 0-th axis
np.concatenate([grid, grid])
```

Out[25]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

In [27]:
```
# concatenate along the 1-st axis
np.concatenate([grid, grid], axis=1)
```

Out[27]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the `np.vstack` (vertical stack) and `np.hstack` (horizontal stack) functions:

In [28]:
```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

# vertically stack the arrays
np.vstack([x, grid])
```

Out[28]:
```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

In [29]:
```
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

Out[29]:
```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

Similary, `np.dstack` will stack arrays along the third axis.

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`. For each of these, we can pass a list of indices giving the split points:

In [30]:
```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

Notice that *N* split-points, leads to *N + 1* subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

In [31]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

Out[31]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [44]:
```
upper, middle, lower = np.split(grid, [1,3])
print(upper)
print(middle)
print(lower)
```

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

In [38]:
```
upper, middle, lower = np.split(grid, [1,3], axis=1)
print(upper)
print(middle)
print(lower)
```

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

In [32]:
```
upper, middle, lower = np.vsplit(grid, [1,3])
print(upper)
print(middle)
print(lower)
```

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

In [33]:
```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

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

The following table lists the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

For example, calling `reduce` on the `add` ufunc returns the sum of all elements in the array:

In [42]:
```
x = np.arange(1, 6)
np.add.reduce(x)
```

Out[42]:
```
15
```

Similarly, calling `reduce` on the `multiply` ufunc results in the product of all array elements:

In [43]:
```
np.multiply.reduce(x)
```

Out[43]:
```
120
```

If we'd like to store all the intermediate results of the computation, we can instead use `accumulate`:

In [60]:
```
np.add.accumulate(x)
```

Out[60]:
```
array([ 1,  3,  6, 10, 15], dtype=int32)
```

In [61]:
```
np.multiply.accumulate(x)
```

Out[61]:
```
array([  1,   2,   6,  24, 120], dtype=int32)
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum` , `np.prod` , `np.cumsum` , `np.cumprod`), which we'll explore a bit later.

Outer products

Finally, any ufunc can compute the output of all pairs of two different inputs using the `outer` method. This allows you, in one line, to do things like create a multiplication table:

In [62]:
```
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

Out[62]:
```
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

Another extremely useful feature of ufuncs is the ability to operate between arrays of different sizes and shapes, a set of operations known as *broadcasting*.

Ufuncs: Learning More

More information on universal functions (including the full list of available functions) can be found on the NumPy (http://www.numpy.org) and SciPy (http://www.scipy.org) documentation websites.

Recall that you can also access information directly from within IPython by importing the packages and using IPython's tab-completion and help (`?`) functionality.

Aggregations: Min, Max, and Everything In Between

Often when faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" values in a dataset, but other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in `sum` function:

In [63]:
```
L = np.random.random(100)
sum(L)
```

Out[63]:
```
51.553048485350516
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same in the simplest case:

In [64]:
```
np.sum(L)
```

Out[64]:
```
51.55304848535052
```

However, because it executes the operation in compiled code, NumPy's version of the operation is computed much more quickly:

In [65]:
```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
641 ms ± 39.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
3.32 ms ± 333 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings, and `np.sum` is aware of multiple array dimensions, as we will see in the following section.

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

In [66]:
```
min(big_array), max(big_array)
```

Out[66]:
```
(1.0606113913791404e-06, 0.9999989464589519)
```

NumPy's corresponding functions have similar syntax, and again operate much more quickly:

In [67]:
```
np.min(big_array), np.max(big_array)
```

Out[67]:
```
(1.0606113913791404e-06, 0.9999989464589519)
```

In [68]:
```
%timeit min(big_array)
%timeit np.min(big_array)
```

```
275 ms ± 43.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.72 ms ± 23.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

For `min`, `max`, `sum`, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

In [69]:
```
print(big_array.min(), big_array.max(), big_array.sum())
```

```
1.0606113913791404e-06 0.9999989464589519 500250.8287074438
```

Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

Multi dimensional aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

In [70]:
```
M = np.random.random((3, 4))
print(M)
```

```
[[0.34725079 0.22654289 0.84485336 0.45120624]
 [0.37410255 0.89292408 0.17687681 0.38880296]
 [0.51939008 0.92973522 0.64492323 0.16325743]]
```

By default, each NumPy aggregation function will return the aggregate over the entire array:

In [71]:
```
M.sum()
```

Out[71]:
```
5.959865643765183
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying `axis=0`:

In [72]:
```
M.min(axis=0)
```

Out[72]:
```
array([0.34725079, 0.22654289, 0.17687681, 0.16325743])
```

The function returns four values, corresponding to the four columns of numbers.

Similarly, we can find the maximum value within each row:

In [73]:
```
M.max(axis=1)
```

Out[73]:
```
array([0.84485336, 0.89292408, 0.92973522])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array that will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the

first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

```
In [45]: A = np.arange(36).reshape((6,6))
         A
```

```
Out[45]: array([[ 0,  1,  2,  3,  4,  5],
                [ 6,  7,  8,  9, 10, 11],
                [12, 13, 14, 15, 16, 17],
                [18, 19, 20, 21, 22, 23],
                [24, 25, 26, 27, 28, 29],
                [30, 31, 32, 33, 34, 35]])
```

```
In [56]: A.sum(axis=0)
```

```
Out[56]: array([ 90,  96, 102, 108, 114, 120])
```

```
In [57]: A.sum(axis=1)
```

```
Out[57]: array([ 15,  51,  87, 123, 159, 195])
```

Other aggregation functions

NumPy provides many other aggregation functions, but we won't discuss them in detail here. Additionally, most aggregates have a `NaN` -safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point `NaN` value. Some of these `NaN` -safe functions were not added until NumPy 1.8, so they will not be available in older NumPy versions.

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

In [59]:
```
A = A.astype(np.float16)
A
```

Out[59]:
```
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.],
       [24., 25., 26., 27., 28., 29.],
       [30., 31., 32., 33., 34., 35.]], dtype=float16)
```

In [73]:
```
A = np.arange(36).reshape((6,6)).astype(np.float16)
A
```

Out[73]:
```
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.],
       [24., 25., 26., 27., 28., 29.],
       [30., 31., 32., 33., 34., 35.]], dtype=float16)
```

In [77]:
```
A[1,[1,3,5]] = np.nan
A
```

Out[77]:
```
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6., nan,  8., nan, 10., nan],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.],
       [24., 25., 26., 27., 28., 29.],
       [30., 31., 32., 33., 34., 35.]], dtype=float16)
```

In [78]:
```
A.sum(axis=0)
```

Out[78]:
```
array([ 90.,  nan, 102.,  nan, 114.,  nan], dtype=float16)
```

In [79]:
```
A.sum(axis=1)
```

Out[79]:
```
array([ 15.,  nan,  87., 123., 159., 195.], dtype=float16)
```

In [81]:
```
np.sum(A, axis=1)
```

Out[81]:
```
array([ 15.,  nan,  87., 123., 159., 195.], dtype=float16)
```

In [80]:
```
A.nansum(axis=1)
```

```
---------------------------------------------------------------
AttributeError                Traceback (most recent call last)
Cell In[80], line 1
----> 1 A.nansum(axis=1)

AttributeError: 'numpy.ndarray' object has no attribute 'nansum'
```

In [82]:
```
np.nansum(A, axis=1)
```

Out[82]:
```
array([ 15.,  24.,  87., 123., 159., 195.], dtype=float16)
```

The formula for the mean value is

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{1}{n}(x_1 + \ldots + x_n)$$

The formula for the sample standard deviation is

$$S = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Computation on Arrays: Broadcasting & Comparisons, Masks, and Boolean Logic

We saw in the previous lesson how NumPy's universal functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.

Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

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

In [2]:
```
a = np.array([0, 1, 2])
b = np.array([5, 6, 7])
a + b
```

Out[2]:
```
array([5, 7, 9])
```

Broadcasting allows these types of binary operations to be performed on arrays of different sizes – for example, we can just as easily add a scalar (think of it as a zero-dimensional array) to an array:

In [3]:
```
a + 5
```

Out[3]:
```
array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value `5` into the array `[5, 5, 5]`, and adds the results. The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

In [4]:
```
M = np.ones((3, 3))
M
```

Out[4]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

In [5]:
```
a
```

Out[5]:
```
array([0, 1, 2])
```

In [6]:
```
M + a
```

Out[6]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

Here the one-dimensional array `a` is stretched, or broadcast across the second dimension in order to match the shape of `M`.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

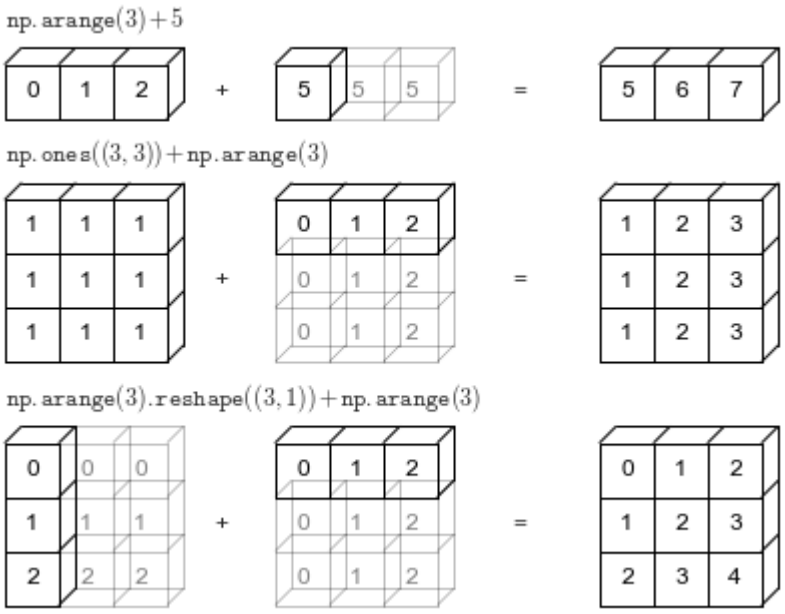
In [7]:
```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
print(a)
print(b)
```

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

In [8]:
```
a + b
```

Out[8]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* `a` and `b` to match a common shape, and the result is a two-dimensional array! The geometry of these examples is visualized in the following figure.



The light boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- 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.

To make these rules clear, let's consider a few examples in detail.

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

In [9]:
```
M = np.ones((2, 3))
a = np.arange(3)
```

In [10]:
```
M.shape
```

Out[10]:
```
(2, 3)
```

In [11]:
```
a.shape
```

Out[11]:
```
(3,)
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `M.shape = (2, 3)`
- `a.shape = (3,)`

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)`:

In [12]:
```
M + a
```

Out[12]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Broadcasting example 2

Let's take a look at an example where both arrays need to be broadcast:

In [13]:
```
a = np.arange(3).reshape((3,1))
b = np.arange(3)
```

In [16]:
```
a.shape
```

Out[16]:
```
(3, 1)
```

In [17]:
```
b.shape
```

Out[17]:
```
(3,)
```

Again, we'll start by writing out the shape of the arrays:

- `a.shape = (3, 1)`
- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (3, 3)`

- `b.shape -> (3, 3)`

Because the result matches, these shapes are compatible. We can see this here:

In [18]:
```
a + b
```

Out[18]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Broadcasting example 3

Now let's take a look at an example in which the two arrays are not compatible:

In [19]:
```
M = np.ones((3,2))
a = np.arange(3)
a.shape
```

Out[19]:
```
(3,)
```

In [20]:
```
M.shape
```

Out[20]:
```
(3, 2)
```

This is just a slightly different situation than in the first example: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

By rule 2, the first dimension of `a` is stretched to match that of `M`:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

Now we hit rule 3–the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

In [21]:
```
M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-8cac1d547906> in <module>
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding `a`'s shape with ones on the right rather than the left. But this is not how

the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array.

In [22]:
```
a.shape
```

Out[22]:
```
(3,)
```

In [25]:
```
a[:, np.newaxis]
```

Out[25]:
```
array([[0],
       [1],
       [2]])
```

In [23]:
```
a[:, np.newaxis].shape
```

Out[23]:
```
(3, 1)
```

In [26]:
```
M + a[:, np.newaxis]
```

Out[26]:
```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

In [27]:
```
M
```

Out[27]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

In [ ]:
```

```