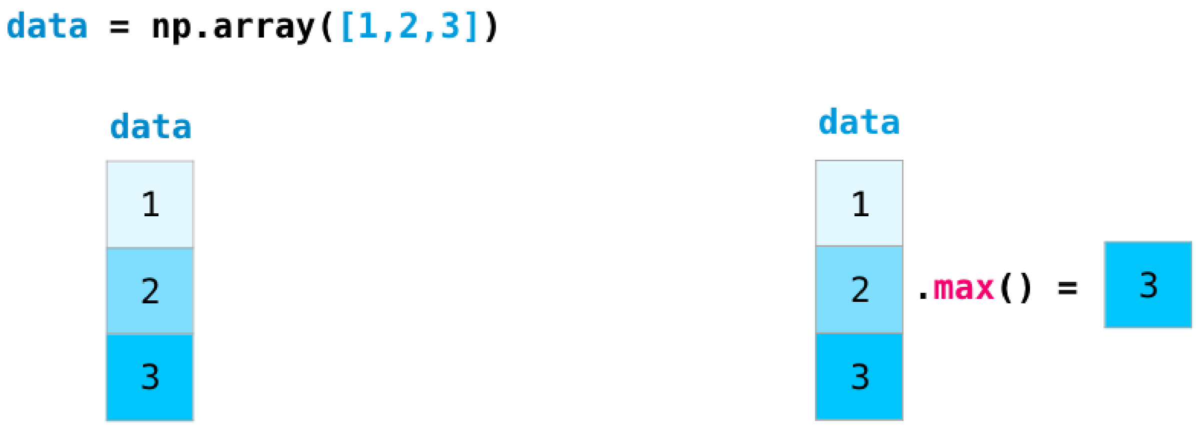
A Visual Intro to NumPy and Data Representation

jalammar.github.io/visual-numpy

Discussions: Hacker News (366 points, 21 comments), Reddit r/MachineLearning (256 points, 18 comments)
Translations: Chinese 1, Chinese 2, Japanese, Korean



```
data = np.array([1,2,3])
```

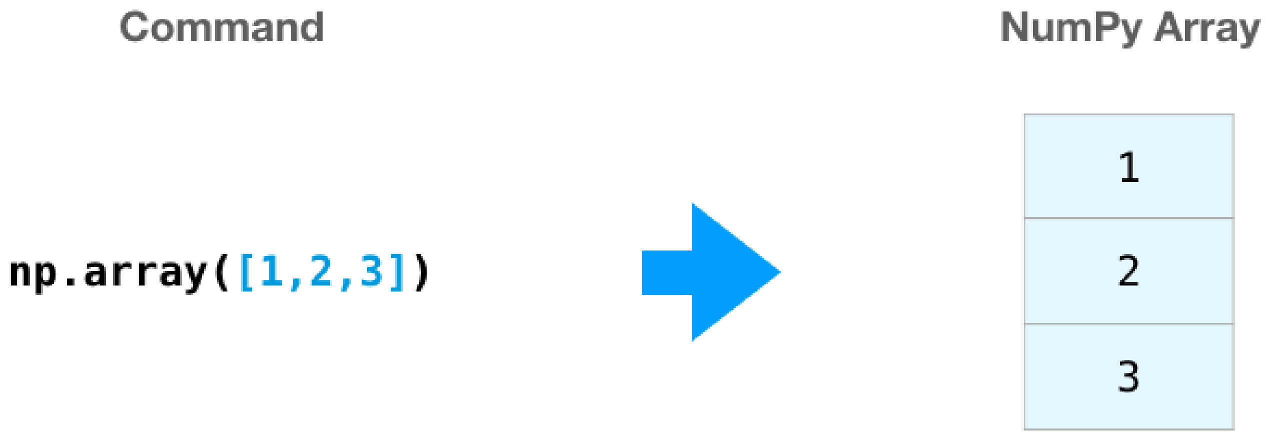
The NumPy package is the workhorse of data analysis, machine learning, and scientific computing in the python ecosystem. It vastly simplifies manipulating and crunching vectors and matrices. Some of python's leading package rely on NumPy as a fundamental piece of their infrastructure (examples include scikit-learn, SciPy, pandas, and tensorflow). Beyond the ability to slice and dice numeric data, mastering numpy will give you an edge when dealing and debugging with advanced usecases in these libraries.

In this post, we'll look at some of the main ways to use NumPy and how it can represent different types of data (tables, images, text…etc) before we can serve them to machine learning models.

```
import numpy as np
```

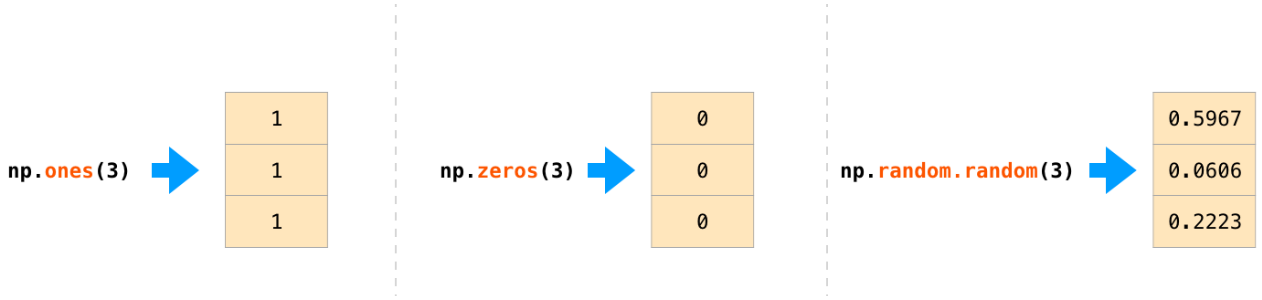
Creating Arrays

We can create a NumPy array (a.k.a. the mighty ndarray) by passing a python list to it and using ` np.array()`. In this case, python creates the array we can see on the right here:



```
np.array([1,2,3])
```

There are often cases when we want NumPy to initialize the values of the array for us. NumPy provides methods like ones(), zeros(), and random.random() for these cases. We just pass them the number of elements we want it to generate:

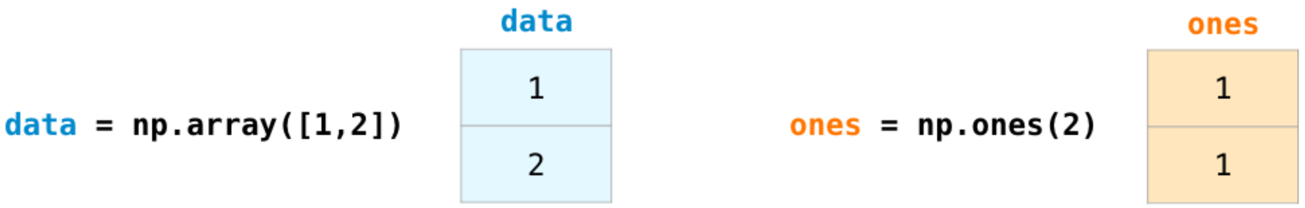


```
np.ones(3)
np.zeros(3)
np.random.random(3)
```

Once we've created our arrays, we can start to manipulate them in interesting ways.

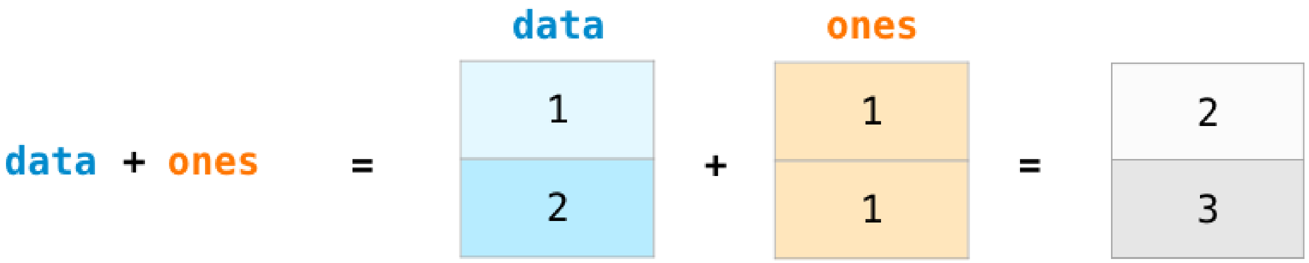
Array Arithmetic

Let's create two NumPy arrays to showcase their usefulness. We'll call them `data` and `ones`:



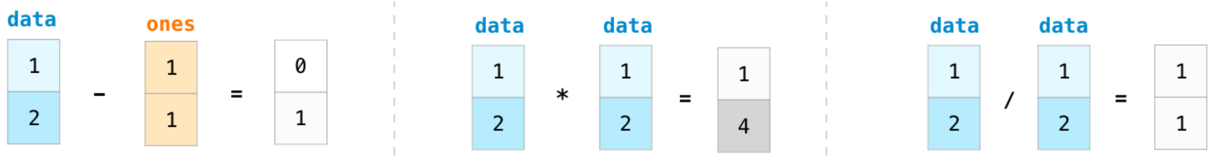
```
data = np.array([1,2])
ones = np.ones(2)
```

Adding them up position-wise (i.e. adding the values of each row) is as simple as typing `data + ones`:

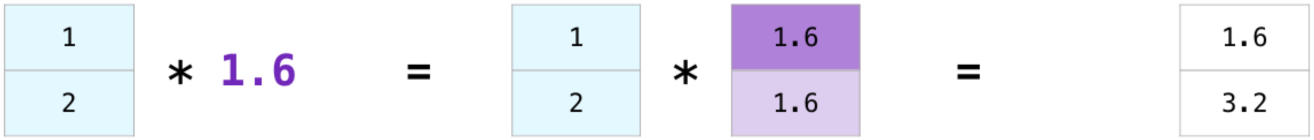


When I started learning such tools, I found it refreshing that an abstraction like this makes me not have to program such a calculation in loops. It’s a wonderful abstraction that allows you to think about problems at a higher level.

And it’s not only addition that we can do this way:



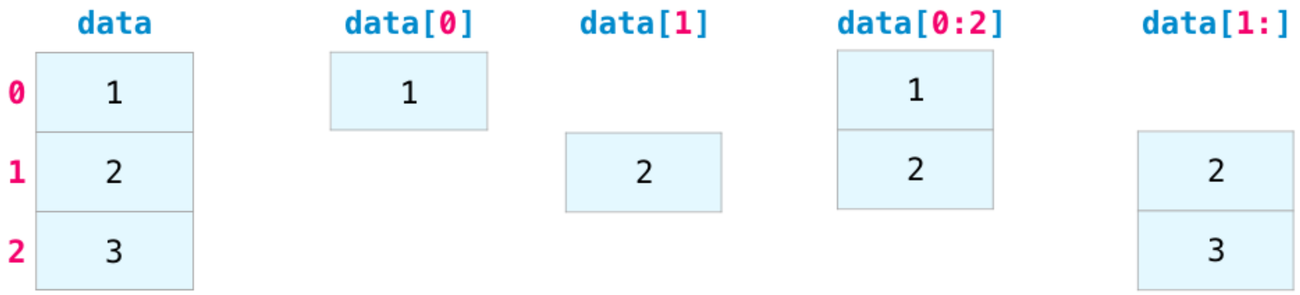
There are often cases when we want to carry out an operation between an array and a single number (we can also call this an operation between a vector and a scalar). Say, for example, our array represents distance in miles, and we want to convert it to kilometers. We simply say `data * 1.6`:



See how NumPy understood that operation to mean that the multiplication should happen with each cell? That concept is called *broadcasting*, and it’s very useful.

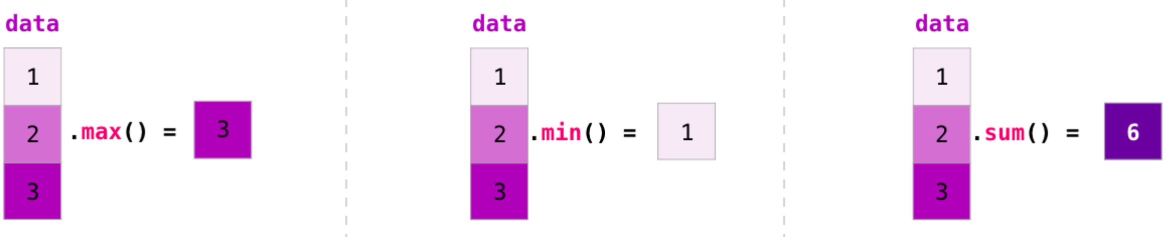
Indexing

We can index and slice NumPy arrays in all the ways we can slice python lists:



Aggregation

Additional benefits NumPy gives us are aggregation functions:



In addition to `min`, `max`, and `sum`, you get all the greats like `mean` to get the average, `prod` to get the result of multiplying all the elements together, `std` to get standard deviation, and plenty of others.

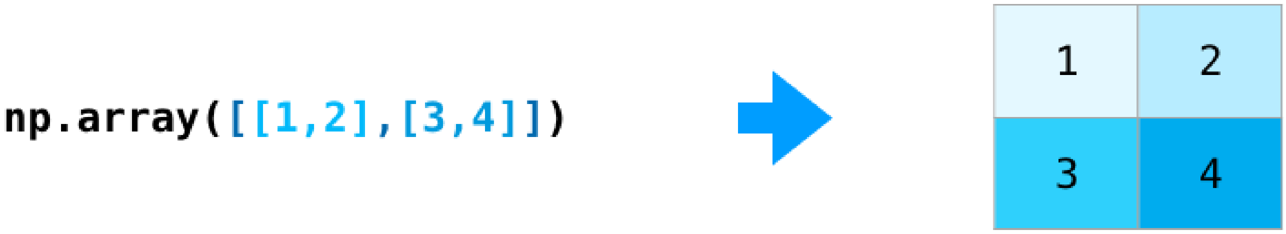
In more dimensions

All the examples we've looked at deal with vectors in one dimension. A key part of the beauty of NumPy is its ability to apply everything we've looked at so far to any number of dimensions.

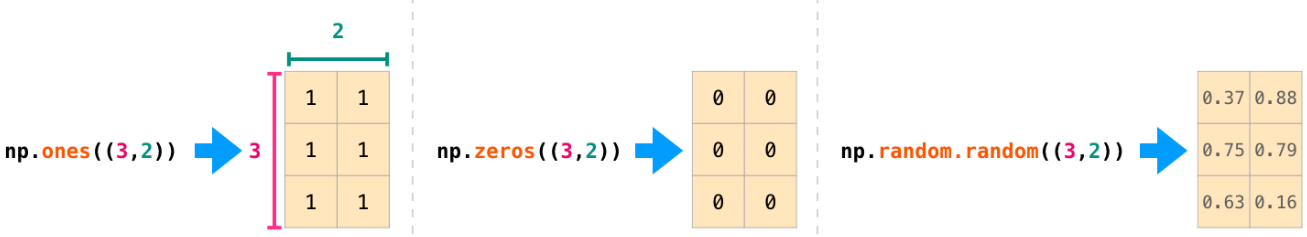
Creating Matrices

We can pass python lists of lists in the following shape to have NumPy create a matrix to represent them:

```
np.array([[1,2],[3,4]])
```

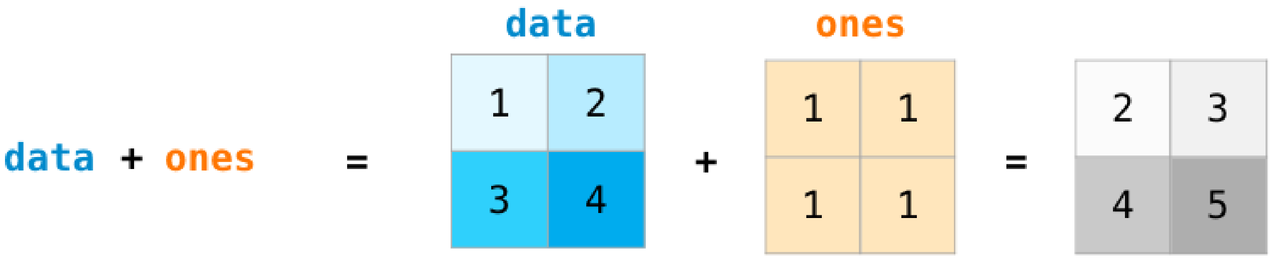


We can also use the same methods we mentioned above (`ones()`, `zeros()`, and `random.random()`) as long as we give them a tuple describing the dimensions of the matrix we are creating:

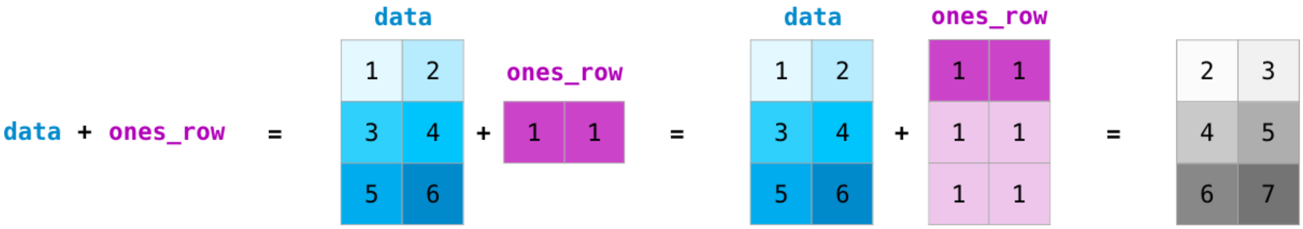


Matrix Arithmetic

We can add and multiply matrices using arithmetic operators (`+-*/`) if the two matrices are the same size. NumPy handles those as position-wise operations:

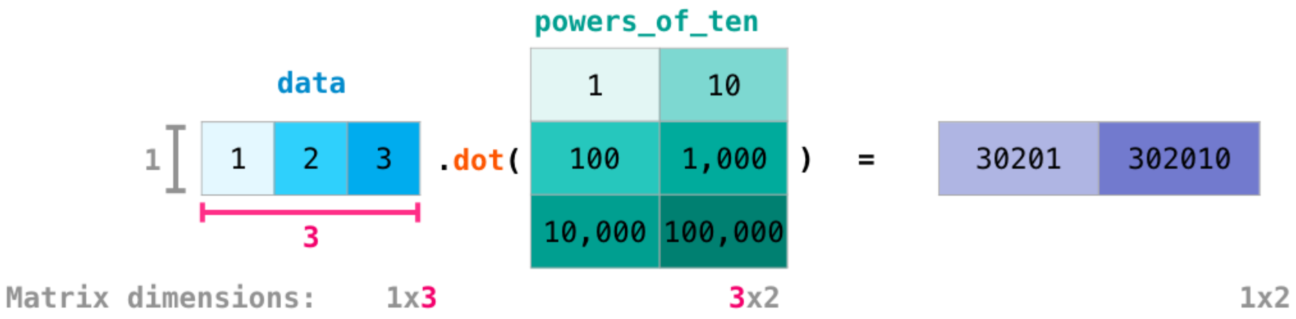


We can get away with doing these arithmetic operations on matrices of different size only if the different dimension is one (e.g. the matrix has only one column or one row), in which case NumPy uses its broadcast rules for that operation:

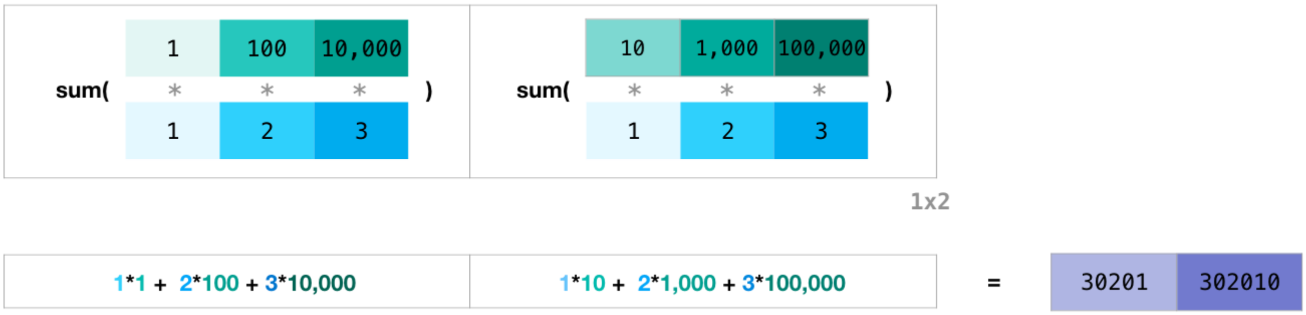


Dot Product

A key distinction to make with arithmetic is the case of matrix multiplication using the dot product. NumPy gives every matrix a `dot()` method we can use to carry-out dot product operations with other matrices:

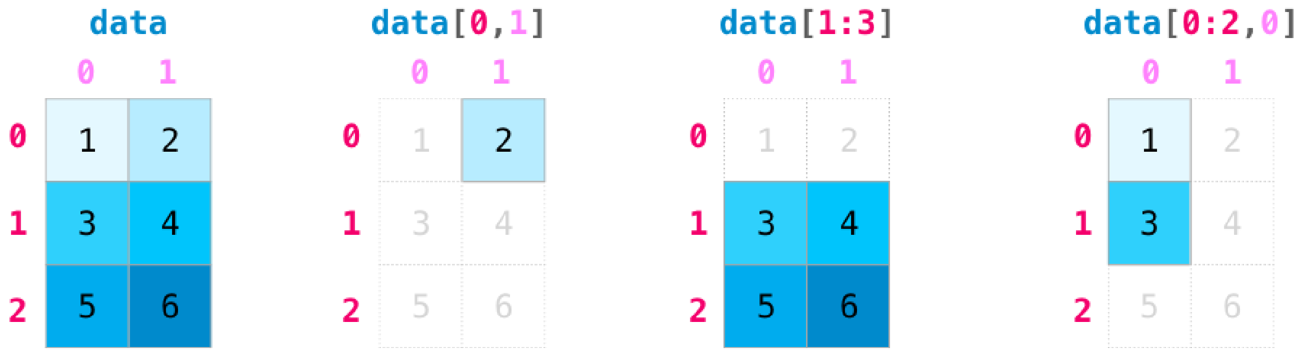


I've added matrix dimensions at the bottom of this figure to stress that the two matrices have to have the same dimension on the side they face each other with. You can visualize this operation as looking like this:



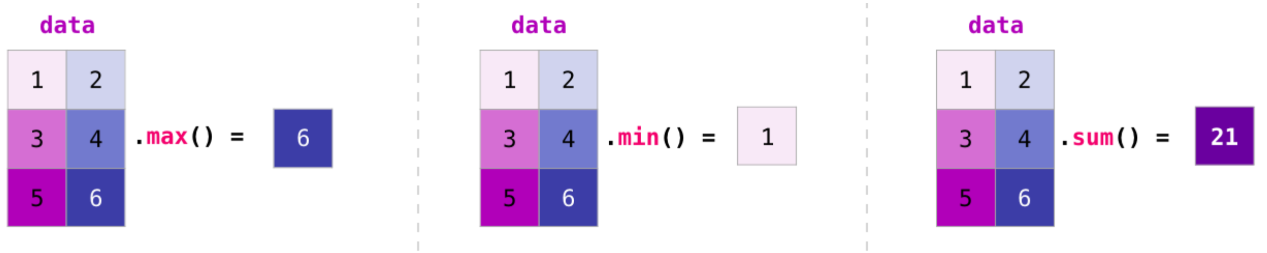
Matrix Indexing

Indexing and slicing operations become even more useful when we're manipulating matrices:

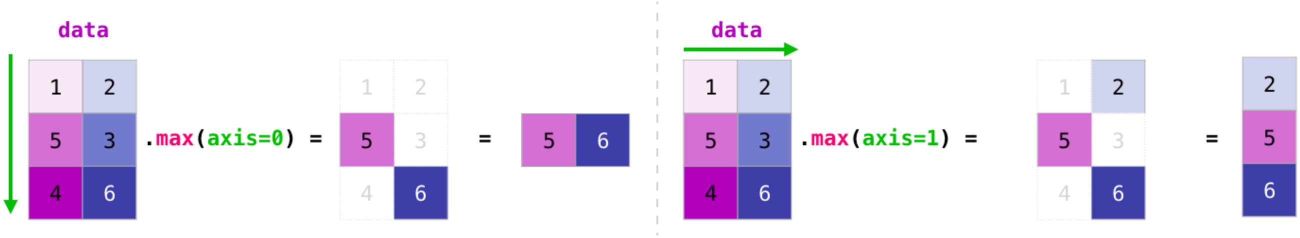


Matrix Aggregation

We can aggregate matrices the same way we aggregated vectors:

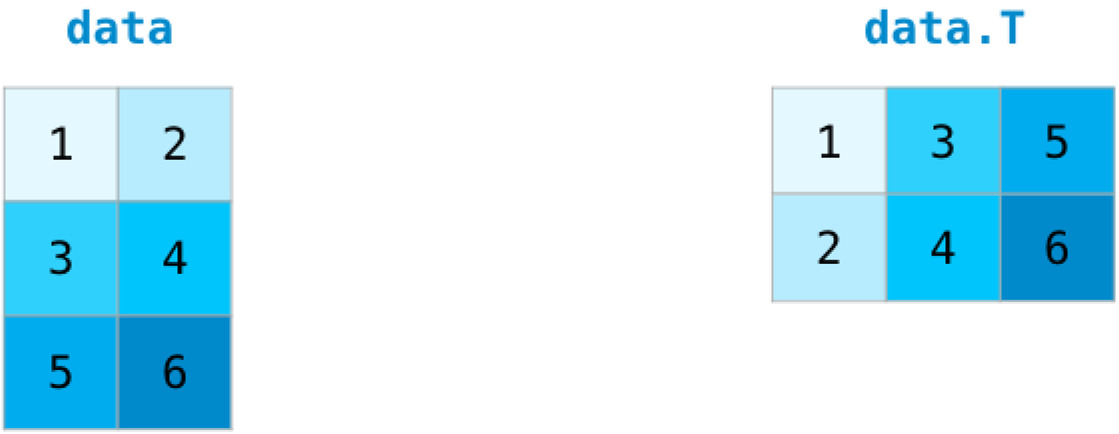


Not only can we aggregate all the values in a matrix, but we can also aggregate across the rows or columns by using the `axis` parameter:

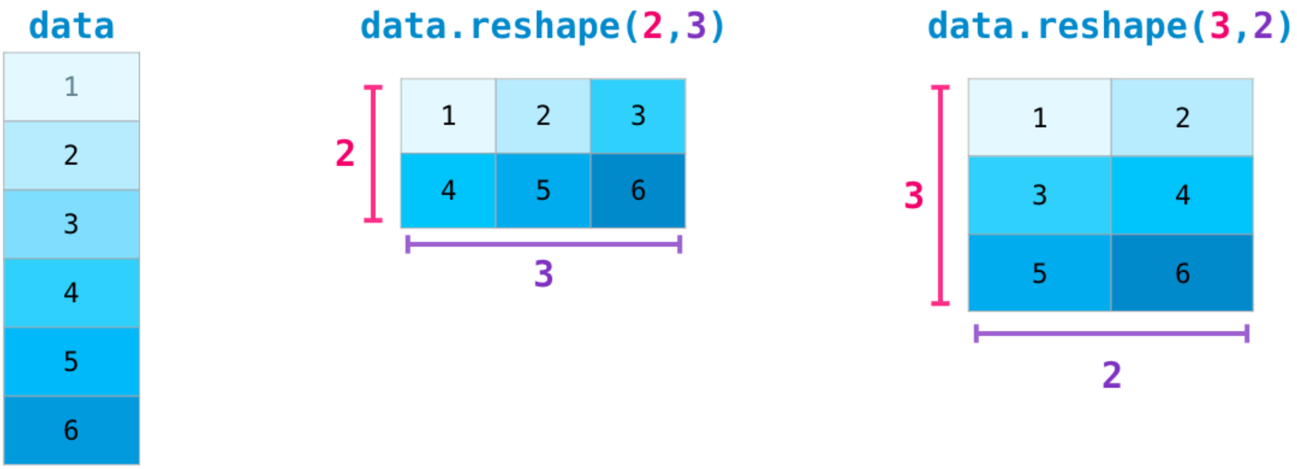


Transposing and Reshaping

A common need when dealing with matrices is the need to rotate them. This is often the case when we need to take the dot product of two matrices and need to align the dimension they share. NumPy arrays have a convenient property called `T` to get the transpose of a matrix:

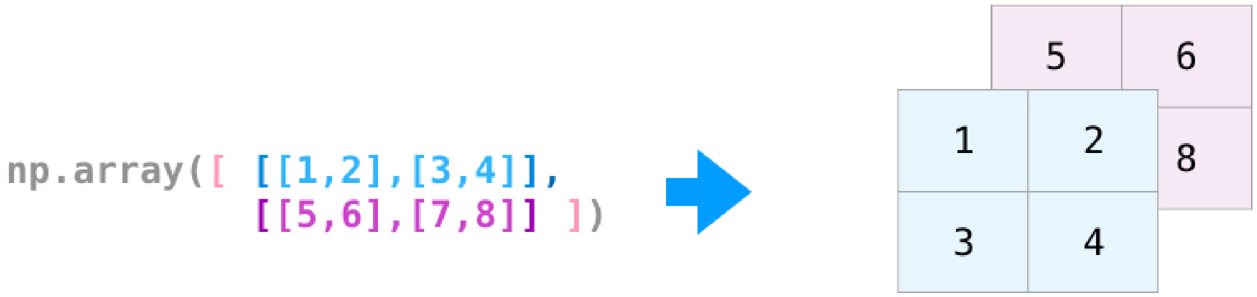


In more advanced use case, you may find yourself needing to switch the dimensions of a certain matrix. This is often the case in machine learning applications where a certain model expects a certain shape for the inputs that is different from your dataset. NumPy's `reshape()` method is useful in these cases. You just pass it the new dimensions you want for the matrix. You can pass -1 for a dimension and NumPy can infer the correct dimension based on your matrix:



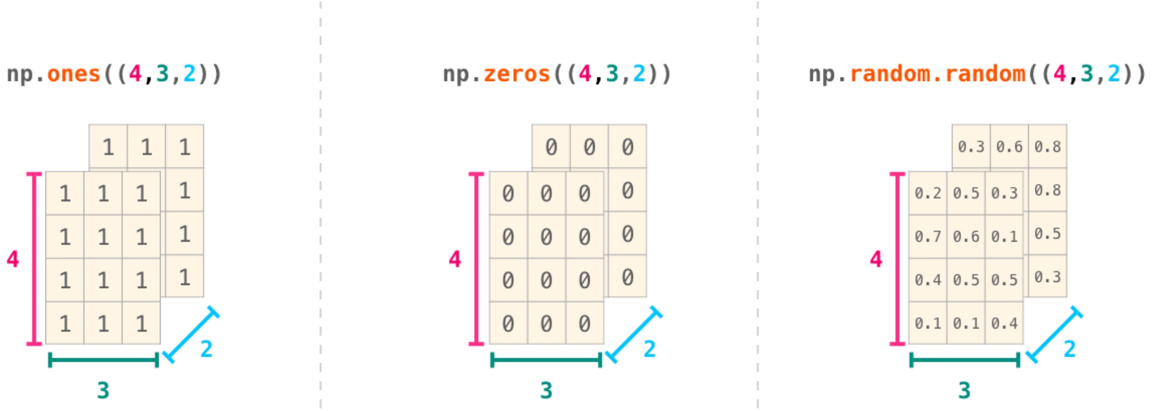
Yet More Dimensions

NumPy can do everything we've mentioned in any number of dimensions. Its central data structure is called ndarray (N-Dimensional Array) for a reason.



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

In a lot of ways, dealing with a new dimension is just adding a comma to the parameters of a NumPy function:



```
np.ones((4,3,2))
```

```
np.zeros((4,3,2))
```

```
np.random.random((4,3,2))
```

Note: Keep in mind that when you print a 3-dimensional NumPy array, the text output visualizes the array differently than shown here. NumPy's order for printing n-dimensional arrays is that the last axis is looped over the fastest, while the first is the slowest. Which means that `np.ones((4,3,2))` would be printed as:

```
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]]])
```

Practical Usage

And now for the payoff. Here are some examples of the useful things NumPy will help you through.

Formulas

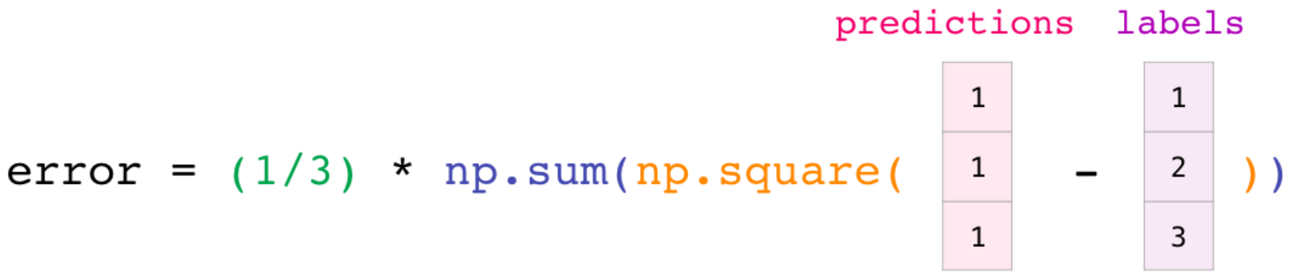
Implementing mathematical formulas that work on matrices and vectors is a key use case to consider NumPy for. It's why NumPy is the darling of the scientific python community. For example, consider the mean square error formula that is central to supervised machine learning models tackling regression problems:

$$MeanSquareError = \frac{1}{n}\sum_{i=1}^{n}(Y\_prediction_i - Y_i)^2$$

Implementing this is a breeze in NumPy:

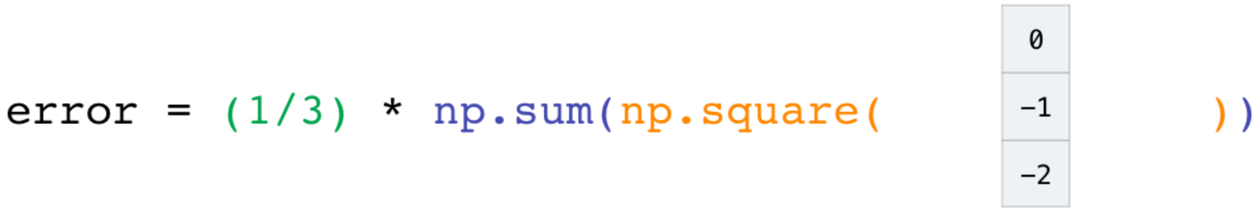
```
error = (1/n) * np.sum(np.square(predictions - labels))
```

The beauty of this is that numpy does not care if `predictions` and `labels` contain one or a thousand values (as long as they're both the same size). We can walk through an example stepping sequentially through the four operations in that line of code:

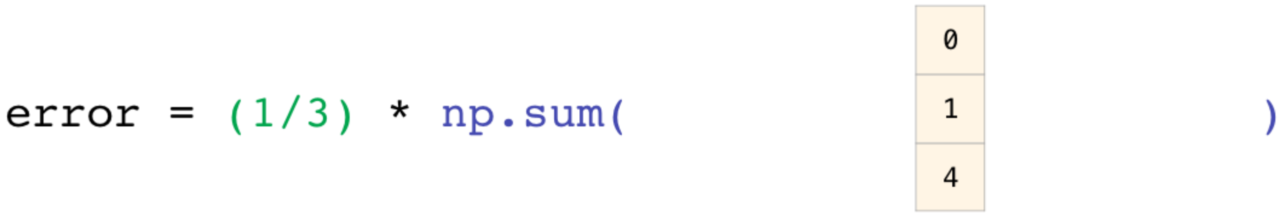


```
error = (1/3) * np.sum(np.square(predictions - labels))
```

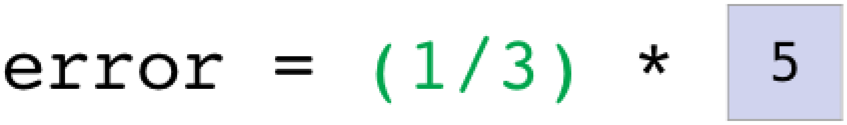
Both the predictions and labels vectors contain three values. Which means n has a value of three. After we carry out the subtraction, we end up with the values looking like this:



Then we can square the values in the vector:



Now we sum these values:



Which results in the error value for that prediction and a score for the quality of the model.

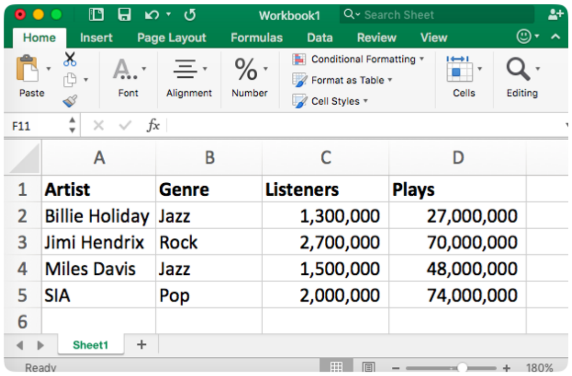
Data Representation

Think of all the data types you'll need to crunch and build models around (spreadsheets, images, audio…etc). So many of them are perfectly suited for representation in an n-dimensional array:

Tables and Spreadsheets

> A spreadsheet or a table of values is a two dimensional matrix. Each sheet in a spreadsheet can be its own variable. The most popular abstraction in python for those is the pandas dataframe, which actually uses NumPy and builds on top of it.

music.csv



	A	B	C	D
1	Artist	Genre	Listeners	Plays
2	Billie Holiday	Jazz	1,300,000	27,000,000
3	Jimi Hendrix	Rock	2,700,000	70,000,000
4	Miles Davis	Jazz	1,500,000	48,000,000
5	SIA	Pop	2,000,000	74,000,000
6				

`pandas.read_csv('music.csv')`

	Artist	Genre	Listeners	Plays
0	Billie Holiday	Jazz	1,300,000	27,000,000
1	Jimi Hendrix	Rock	2,700,000	70,000,000
2	Miles Davis	Jazz	1,500,000	48,000,000
3	SIA	Pop	2,000,000	74,000,000