Numpy axes explained

This tutorial will explain NumPy axes.

It will explain what a NumPy axis is. The tutorial will also explain how axes work, and how we use them with NumPy functions.

Although it's probably best for you to read the full tutorial, if you want to skip ahead, you can do so by clicking on one of the following links:

Before I get into a detailed explanation of NumPy axes, let me just start by explaining why NumPy axes are problematic.

Numpy axes are hard to understand

NumPy axes are one of the hardest things to understand in the NumPy system. If you're just getting started with NumPy, this is particularly true. Many beginners struggle to understand how NumPy axes work.

Don't worry, it's not you. A *lot* of Python data science beginners struggle with this.

Having said that, this tutorial will explain all the essentials that you need to know about axes in NumPy arrays.

Let's start with the basics. I'll make NumPy axes easier to understand by connecting them to something you already know.

Numpy axes are like axes in a coordinate system

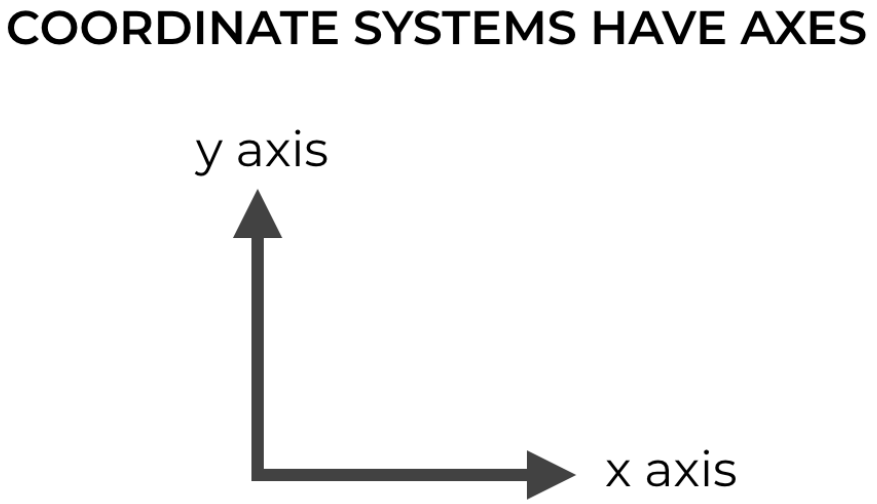
If you're reading this blog post, chances are you've taken more than a couple of math classes.

Think back to early math, when you were first learning about graphs.

You learned about Cartesian coordinates. NumPy axes are very similar to axes in a Cartesian coordinate system.

An analogy: cartesian coordinate systems have axes

You probably remember this, but just so we're clear, let's take a look at a simple Cartesian coordinate system.

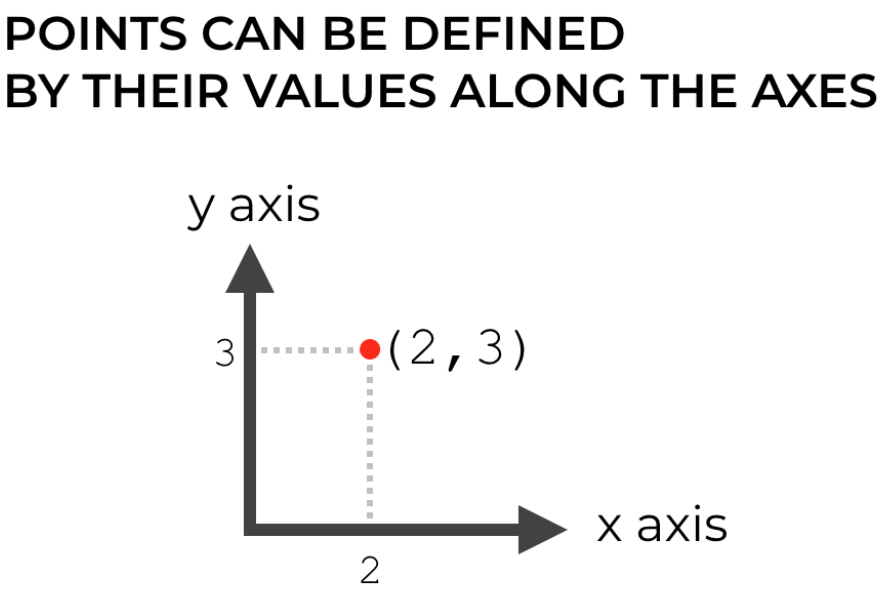


A simple 2-dimensional Cartesian coordinate system has two axes, the x axis and the y axis.

These axes are essentially just directions in a Cartesian space (orthogonal directions).

Moreover, we can identify the position of a point in Cartesian space by it's position along each of the axes.

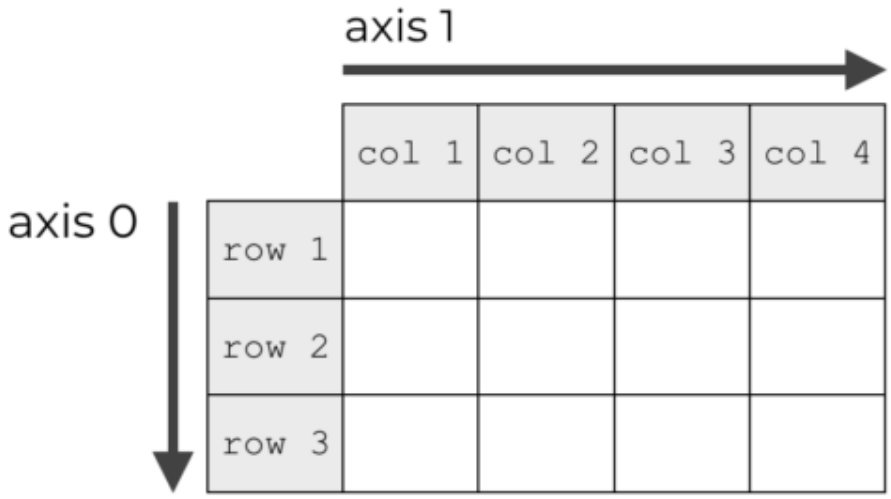
So if we have a point at position `(2, 3)`, we're basically saying that it lies 2 units along the x axis and 3 units along the y axis.



If all of this is familiar to you, good. You're half way there to understanding NumPy axes.

NumPy axes are the directions along the rows and columns

Just like coordinate systems, NumPy arrays also have axes.

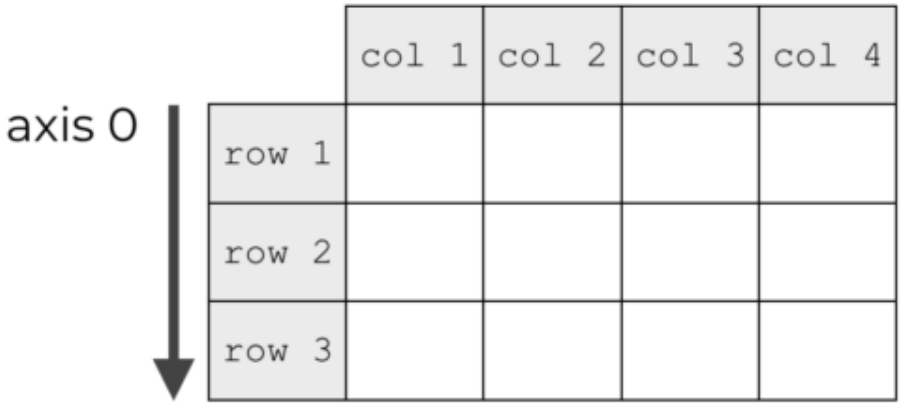


In a 2-dimensional NumPy array, the axes are the *directions* along the rows and columns.

Axis 0 is the direction along the rows

In a NumPy array, axis 0 is the “first” axis.

Assuming that we’re talking about multi-dimensional arrays, axis 0 is the axis that runs downward down the rows.

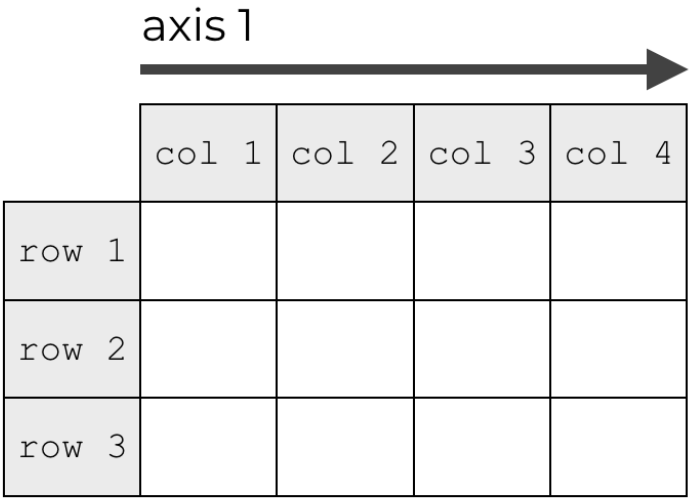


Keep in mind that this really applies to 2-d arrays and multi dimensional arrays. 1-dimensional arrays are a bit of a special case, and I’ll explain those later in the tutorial.

Axis 1 is the direction along the columns

In a multi-dimensional NumPy array, axis 1 is the second axis.

When we’re talking about 2-d and multi-dimensional arrays, axis 1 is the axis that runs horizontally across the columns.



Once again, keep in mind that 1-d arrays work a little differently. Technically, 1-d arrays *don't have an axis 1*. I'll explain more about this later in the tutorial.

NumPy array axes are numbered starting with '0'

It is probably obvious at this point, but I should point out that array axes in NumPy are numbered.

Importantly, they are numbered starting with *0*.

This is just like index values for Python sequences. In Python sequences – like lists and tuples – the values in a the sequence have an index associated with them.

So, let's say that we have a Python list with a few capital letters:

```
alpha_list = ['A','B','C','D']
```

If we retrieve the index value of the first item (' `A` ') …

```
alpha_list.index('A')
```

… we find that ' `A` ' is at index position 0.

Here, `A` is the first item in the list, but the index position is 0.

Essentially all Python sequences work like this. In any Python sequence – like a list, tuple, or string – the index starts at 0.

Numbering of NumPy axes essentially works the same way. They are numbered starting with 0. So the "first" axis is actually "axis 0." The "second" axis is "axis 1," and so on.

The structure of NumPy array axes is important

In the following section, I'm going to show you examples of how NumPy axes are used in NumPy, but before I show you that, you need to remember that the structure of NumPy arrays matters.

The details that I just explained, about axis numbers, and about which axis is which is going to impact your understanding of the NumPy functions we use.

Having said that, before you move on to the examples, make sure you really understand the details that I explained above about NumPy axes.

And if you have any questions or you're still confused about NumPy axes, leave a question in the comments at the bottom of the page.

Ok. Now, let's move on to the examples.

Examples of how Numpy axes are used

Now that we've explained how NumPy axes work in general, let's look at some specific examples of how NumPy axes are used.

These examples are important, because they will help develop your intuition about how NumPy axes work when used with NumPy functions.

Run this code before you start

Before we start working with these examples, you'll need to run a small bit of code:

```
import numpy as np
```

This code will basically import the NumPy package into your environment so you can work with it. Going forward, you'll be able to reference the NumPy package as `np` in our syntax.

A word of advice: pay attention to what the axis parameter controls

Before I show you the following examples, I want to give you a piece of advice.

To understand how to use the axis parameter in the NumPy functions, it's very important to understand what the `axis` parameter actually controls for each function.

This is not always as simple as it sounds. For example, in the `np.sum()` function, the `axis` parameter behaves in a way that many people think is counter intuitive.

I'll explain exactly how it works in a minute, but I need to stress this point: pay very careful attention to what the `axis` parameter actually controls for each function.

Numpy sum

Let's take a look at how NumPy axes work inside of the NumPy sum function.

When trying to understand axes in NumPy sum, you need to know what the `axis` parameter actually controls.

In `np.sum()`, the `axis` parameter controls which axis will be *aggregated*.

Said differently, the `axis` parameter controls which axis will be *collapsed*.

Remember, functions like `sum()`, `mean()`, `min()`, `median()`, and other statistical functions *aggregate* your data.

To explain what I mean by "aggregate," I'll give you a simple example.

Imagine you have a set of 5 numbers. If sum up those 5 numbers, the result will be a *single* number. Summation effectively aggregates your data. It collapses a large number of values into a single value.

Similarly, when you use `np.sum()` on a 2-d array with the `axis` parameter, it is going to collapse your 2-d array down to a 1-d array. It will collapse the data and reduce the number of dimensions.

But which axis will get collapsed?

When you use the NumPy sum function with the `axis` parameter, the axis that you specify is *the axis that gets collapsed*.

Let's take a look at that.

Numpy sum with axis = 0

Here, we're going to use the NumPy sum function with `axis = 0`.

First, we're just going to create a simple NumPy array.

```
np_array_2d = np.arange(0, 6).reshape([2,3])
```

And let's quickly print it out, so you can see the contents.

```
print(np_array_2d)
```

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

The array, `np_array_2d`, is a 2-dimensional array that contains the values from 0 to 5 in a 2-by-3 format.

Next, let's use the NumPy sum function with `axis = 0`.

```
np.sum(np_array_2d, axis = 0)
```

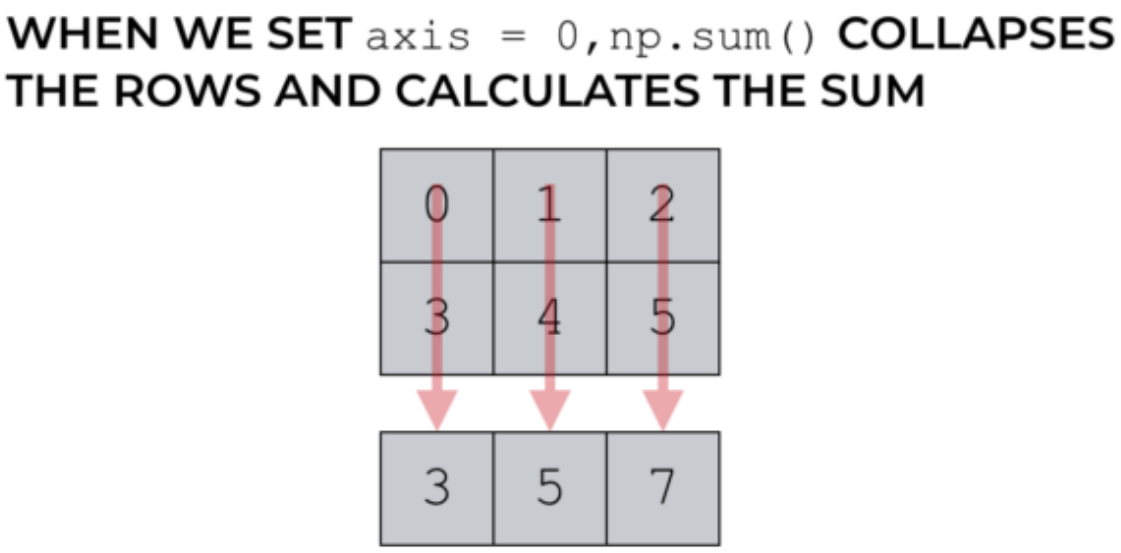
And here's the output.

```
array([3, 5, 7])
```

When we set `axis = 0`, the function actually sums down the columns. The result is a new NumPy array that contains the sum of each column. Why? Doesn't axis 0 refer to the rows?

This confuses many beginners, so let me explain.

As I mentioned earlier, the `axis` parameter indicates which axis gets *collapsed*.



So when we set `axis = 0`, we're not summing across the rows. When we set `axis = 0`, we're aggregating the data such that we *collapse* the rows … we collapse axis 0.

Numpy sum with axis = 1

Now, let's use the NumPy sum function on our array with `axis = 1`.

In this example, we're going to reuse the array that we created earlier, `np_array_2d`.

Remember that it is a simple 2-d array with 6 values arranged in a 2 by 3 form.

```
print(np_array_2d)
```

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

Here, we're going to use the sum function, and we'll set the `axis` parameter to `axis = 1`.

```
np.sum(np_array_2d, axis = 1)
```

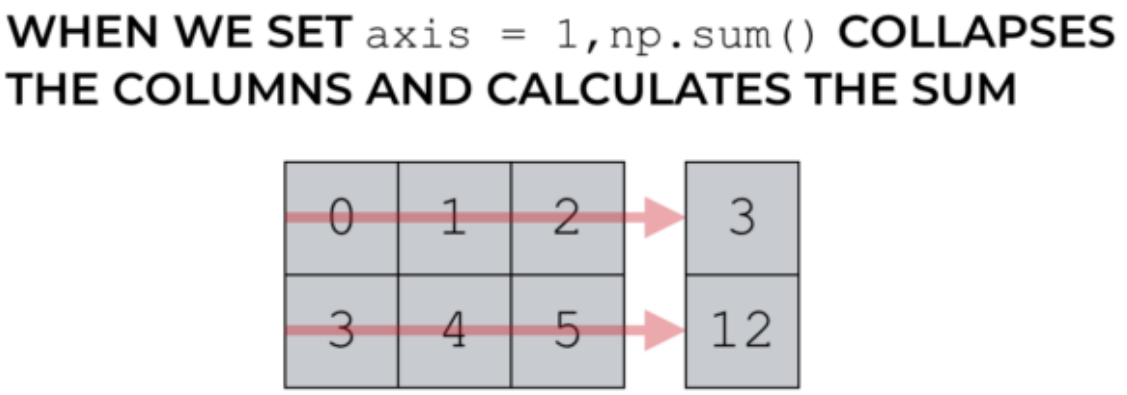
And here's the output:

```
array([3, 12])
```

Let me explain.

Again, with the `sum()` function, the `axis` parameter sets the axis that gets collapsed during the summation process.

Recall from earlier in this tutorial that axis 1 refers to the horizontal direction across the columns. That means that the code `np.sum(np_array_2d, axis = 1)` collapses the columns during the summation.



As I mentioned earlier, this confuses many beginners. They expect that by setting `axis = 1`, NumPy would sum *down* the columns, but that's not how it works.

The code has the effect of summing across the columns. It collapses axis 1.

Numpy concatenate

Now let's take a look at a different example.

Here, we're going to work with the axis parameter in the context of using the NumPy concatenate function.

When we use the `axis` parameter with the `np.concatenate()` function, the `axis` parameter defines the axis along which we stack the arrays. If that doesn't make sense, then work through the examples. It will probably become more clear once you run the code and see the output.

In both of the following examples, we're going to work with two 2-dimensional NumPy arrays:

```
np_array_1s = np.array([[1,1,1],[1,1,1]])
np_array_9s = np.array([[9,9,9],[9,9,9]])
```

Which have the following structure, respectively:

```
array([[1, 1, 1],
       [1, 1, 1]])
```

And:

```
array([[9, 9, 9],
       [9, 9, 9]])
```

Numpy concatenate with axis = 0

First, let's look at how to use NumPy concatenate with axis = 0.

```
np.concatenate([np_array_1s, np_array_9s], axis = 0)
```

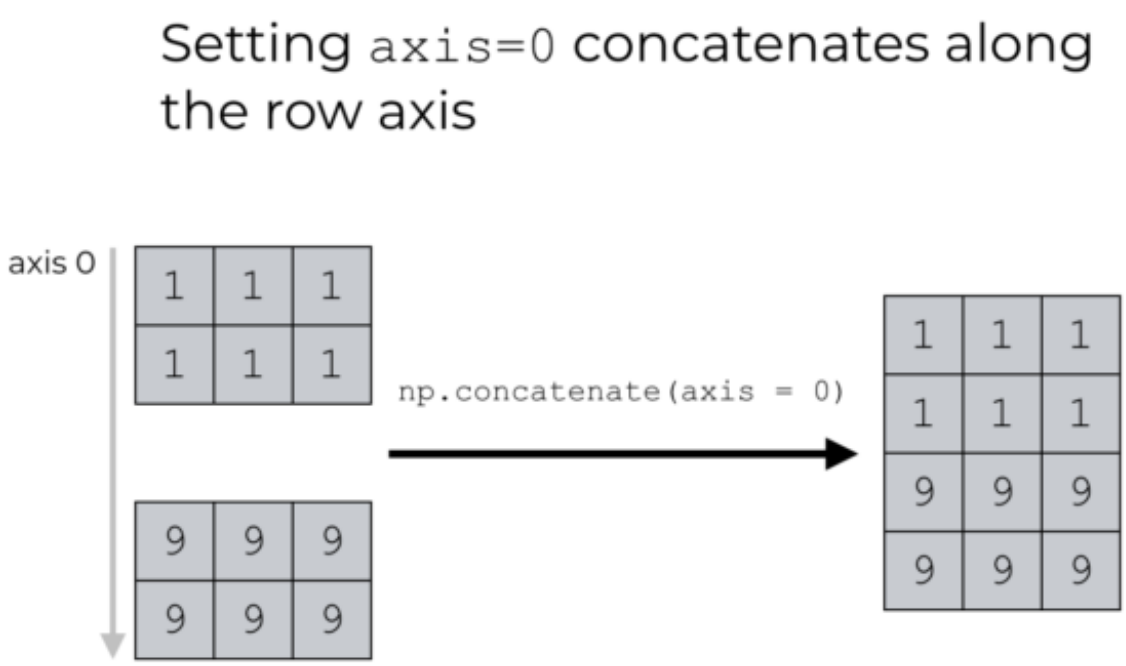
Which produces the following output:

```
array([[1, 1, 1],
       [1, 1, 1],
       [9, 9, 9],
       [9, 9, 9]])
```

Let's carefully evaluate what the syntax did here.

Recall what I mentioned a few paragraphs ago. When we use the concatenate function, the `axis` parameter defines the axis along which we stack the arrays.

So when we set `axis = 0`, we're telling the concatenate function to stack the two arrays along the rows. We're specifying that we want to concatenate the arrays along axis 0.



Numpy concatenate with axis = 1

Now let's take a look at an example of using `np.concatenate()` with `axis = 1`.

Here, we're going to reuse the two 2-dimensional NumPy arrays that we just created, `np_array_1s` and `np_array_9s`.

We're going to use the concatenate function to combine these arrays together horizontally.

```
np.concatenate([np_array_1s, np_array_9s], axis = 1)
```

Which produces the following output:

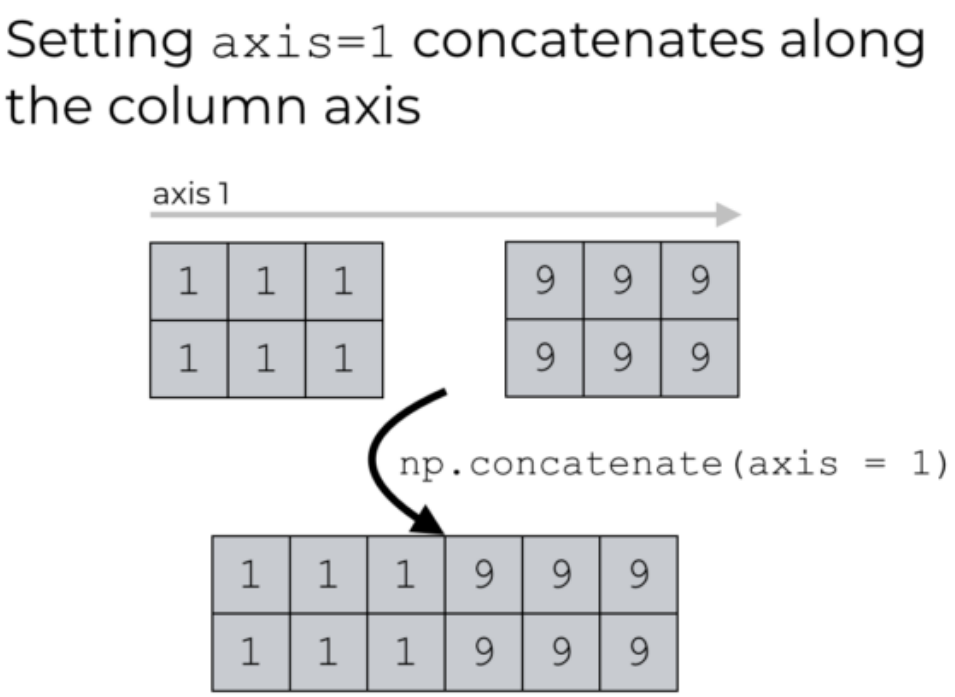
```
array([[1, 1, 1, 9, 9, 9],
       [1, 1, 1, 9, 9, 9]])
```

If you've been reading carefully and you've understood the other examples in this tutorial, this should make sense.

However, let's quickly review what's going on here.

These arrays are 2 dimensional, so they have two axes, axis 0 and axis 1. Axis 1 is the axis that runs *horizontally* across the columns of the NumPy arrays.

When we use NumPy concatenate with `axis = 1`, we are telling the `concatenate()` function to combine these arrays together along axis 1.



That is, we're telling `concatenate()` to combine them together *horizontally*, since axis 1 is the axis that runs horizontally across the columns.

Warning: 1-dimensional arrays work differently

Hopefully this NumPy axis tutorial helped you understand how NumPy axes work.

But before I end the tutorial, I want to give you a warning: 1-dimensional arrays work differently!

Everything that I've said in this post really applies to 2-dimensional arrays (and to some extent, multi-dimensional arrays).

The axes of 1-dimensional NumPy arrays work differently. For beginners, this is likely to cause issues.

Having said all of that, let me quickly explain how axes work in 1-dimensional NumPy arrays.

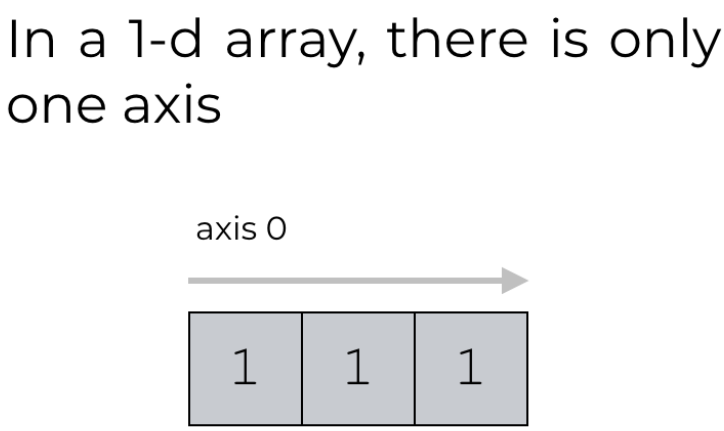
1-dimensional NumPy arrays only have one axis

The important thing to know is that 1-dimensional NumPy arrays only have one axis.

If 1-d arrays only have one axis, can you guess the name of that axis?

Remember, axes are numbered like Python indexes. They start at 0.

So, in a 1-d NumPy array, the first and *only* axis is axis 0.



The fact that 1-d arrays have only one axis can cause some results that confuse NumPy beginners.

Example: concatenating 1-d arrays

Let me show you an example of some of these "confusing" results that can occur when working with 1-d arrays.

We're going to create two simple 1-dimensional arrays.

```
np_array_1s_1dim = np.array([1,1,1])
np_array_9s_1dim = np.array([9,9,9])
```

And we can print them out to see the contents:

```
print(np_array_1s_1dim)
print(np_array_9s_1dim)
```

Output:

```
[1 1 1]
[9 9 9]
```

As you can see, these are two simple 1-d arrays.

Next, let's concatenate them together using `np.concatenate()` with `axis = 0`.

```
np.concatenate([np_array_1s_1dim, np_array_9s_1dim], axis = 0)
```

Output:

```
array([1, 1, 1, 9, 9, 9])
```

This output confuses many beginners. The arrays were concatenated together *horizontally*.

This is different from how the function works on 2-dimensional arrays. If we use `np.concatenate()` with `axis = 0` on `2`-dimensional arrays, the arrays will be concatenated together *vertically*.

What's going on here?

Recall what I just mentioned a few paragraphs ago: 1-dimensional NumPy arrays only have one axis. Axis 0.

The function is working properly in this case. NumPy concatenate *is* concatenating these arrays along axis 0. The issue is that in 1-d arrays, axis 0 doesn't point "downward" like it does in a 2-dimensional array.

Example: an error when concatenating 1-d arrays, with axis = 1

Moreover, you'll also run into problems if you try to concatenate these arrays on axis 1.

Try it:

```
np.concatenate([np_array_1s_1dim, np_array_9s_1dim], axis = 1)
```

This code causes an error:

```
IndexError: axis 1 out of bounds [0, 1)
```

If you've been reading carefully, this error should make sense. `np_array_1s_1dim` and `np_array_9s_1dim` are 1-dimensional arrays. Therefore, *they don't have an axis 1* . We're trying to use `np.concatenate()` on an axis that doesn't exist in these arrays. Therefore, the code generates an error.

Be careful when using axes with 1-d arrays

All of this is to say that you need to be careful when working with 1-dimensional arrays. When you're working with 1-d arrays, and you use some NumPy functions with the `axis` parameter, the code can generate confusing results.

The results make a lot of sense if you really understand how NumPy axes work. But if you don't understand NumPy array axes, the results will probably be confusing.

So make sure that before you start working with NumPy array axes that you really understand them!

As you've seen in this tutorial, NumPy axes can be a little confusing. They are especially confusing to NumPy beginners.

But, in order to use NumPy correctly, you really need to understand how NumPy axes work.