

NUMPY CRASH COURSE

A quick and easy guide to getting
started with Numpy by

SHARP SIGHT

Table of Contents

A quick introduction to Numpy arrays	3
Numpy axes explained	29
How to use the Numpy arange function	57
How to use the Numpy linspace function	70
How to Use the Numpy Sum Function	85
Numpy random seed explained	110
How to use numpy random normal in Python	144
Do you want to master Numpy?	160

A quick introduction to Numpy arrays

One of the cornerstones of the Python data science ecosystem is Numpy, and the foundation of Numpy is the Numpy array.

That being the case, if you want to learn data science in Python, you'll need to learn how to work with Numpy arrays.

Here, I'll explain the essentials of Numpy arrays, including:

- What Numpy arrays are
- How to create a Numpy array
- Attributes of Numpy arrays
- Retrieving individual values from Numpy arrays
- Slicing Numpy arrays
- Creating and working with 2-dimensional Numpy arrays

Let's get started.

What is a Numpy array?

A Numpy array is a collection of elements that have the same data type.

You can think of it like a container that has several compartments that hold data, as long as the data is of the same data type.

Visually, we can represent a simple Numpy array sort of like this:



Let's break this down.

We have a set of integers: 88, 19, 46, 74, 94. These values are all integers; they are all of the same type. You can see that these values are stored in “compartments” of a larger structure. The overall structure is the Numpy array.

Very quickly, I'll explain a little more about some of the properties of a Numpy array.

Numpy arrays must contain data all of the same type

As I mentioned above, Numpy arrays must contain data all of the same type.

That means that if your Numpy array contains integers, all of the values must be integers. If it contains floating point numbers, all of the values must be floats.

I won't write extensively about data types and Numpy data types here. There is a section below in this blog post about how to create a Numpy array of a particular type.

Numpy arrays have an index

Each of the compartments inside of a Numpy array have an "address." We call that address an "index."

Index:	0	1	2	3	4
Value:	88	19	46	74	94

If you're familiar with computing in general, and Python specifically, you're probably familiar with indexes. Many data structures in Python have indexes, and the indexes of a Numpy array essentially work the same.

If you're not familiar with indexes though, let me explain. Again, an index is sort of like an address. These indexes enable you to reference a specific value. We call this indexing.

Index values start at zero

Just like other Python structures that have indexes, the indexes of a Numpy array begin at zero:

Index:	0	1	2	3	4
Value:	88	19	46	74	94

So if you want to reference the value in the very first location, you need to reference location "0". In the example shown here, the value at index 0 is 88.

I'll explain how exactly to use these indexes syntactically, but to do that, I want to give you working examples. To give you working examples, I'll need to explain how to actually create Numpy arrays in Python.

How to create a Numpy array

There are a lot of ways to create a Numpy array. Really. A lot. Off the top of my head, I can think of at least a half dozen techniques and functions that will create a Numpy array (we'll discuss a few in this Crash Course). In fact, the purpose of many of the functions in the Numpy package is to create a Numpy array of one kind or another.

But here, I want to start simple. I'll show you a few very basic ways to do it.

In particular, I'll show you how to use the Numpy `array()` function.

Creating Numpy arrays with the `array()` function

To use the Numpy `array()` function, you call the function and pass in a Python list as the argument.

Let's take a look at some examples. We'll start by creating a 1-dimensional Numpy array.

Create a 1 dimensional Numpy array

Creating a 1-dimensional Numpy array is easy.

You call the function with the syntax `np.array()`. Keep in mind that before you call `np.array()`, you need to import the Numpy package with the code `import numpy as np`.

When you call the `array()` function, you'll need to provide a list of elements as the argument to the function.

```
#import Numpy
import numpy as np

# create a Numpy array from a list of 3 integers
np.array([1,2,3])
```

This isn't complicated, but let's break it down.

We've called the `np.array()` function. The argument to the function is a list of three integers: `[1,2,3]`. It produces a Numpy array of those three integers.

Note that you can also create Numpy arrays with other data types, besides integers. I'll explain how to do that a little later in this Crash Course.

Create a 2 dimensional Numpy array

You can also create 2-dimensional arrays.

To do this using the `np.array()` function, you need to pass in a list of lists.

```
# 2-d array
np.array([[1,2,3],[4,5,6]])
```

Pay attention to what we're doing here, syntactically.

Inside of the call to `np.array()`, there is a list of two lists: `[[1,2,3],[4,5,6]]`. The first list is `[1,2,3]` and the second list is `[4,5,6]`. Those two lists are contained inside of a larger list. The whole input is a list of lists. That list of lists is passed to the array function, which creates a 2-dimensional Numpy array.

This might be a little confusing if you're just getting started with Python and Numpy. In that case, I highly recommend that you review Python lists.

There are also other ways to create a 2-d Numpy array. For example, you can use the `array()` function to create a 1-dimensional Numpy array, and then use the `reshape()` method to reshape the 1-dimensional Numpy array into a 2-dimensional Numpy array.

```
# 2-d array  
np.array([1,2,3,4,5,6]).reshape([2,3])
```

For right now, I don't want to get too "in the weeds" explaining `reshape()`, so I'll leave this as it is. I just want you to understand that there are a few ways to create 2-dimensional Numpy arrays.

I'll write more about how to create and work with 2-dimensional Numpy arrays later in this Crash Course.

N-dimensional Numpy arrays

It's also possible to create 3-dimensional Numpy arrays and N-dimensional Numpy arrays. However, in the interest of simplicity, I'm not going to explain how to create those here.

Create a Numpy array with a specific datatype

Using the Numpy `array()` function, we can also create Numpy arrays with specific data types. Remember that in a Numpy array, all of the elements must be of the same type.

To do this, we need to use the `dtype` parameter inside of the `array()` function.

Here are a couple of examples:

integer

To create a Numpy array with integers, we can use the code `dtype = 'int'`.

```
np.array([1,2,3], dtype = 'int')
```

float

Similarly, to create a Numpy array with floating point number, we can use the code `dtype = 'float'`.

```
np.array([1,2,3], dtype = 'float')
```

These are just a couple of examples. Keep in mind that Numpy supports almost 2 dozen data types ... many more than what I've shown you here.

Having said that, a full explanation of Python data types and Numpy data types is beyond the scope of this book. Just understand that you can specify the data type using the `dtype` parameter.

Common mistakes when creating a Numpy array

I want to point out one common mistake that many beginners make when they try to create a Numpy array with the `np.array()` function.

As I mentioned above, when you create a Numpy array with `np.array()`, you need to provide a list of values.

Many beginners forget to do this and simply provide the values directly to the `np.array()` function, without enclosing them inside of a list. If you attempt to do that it will cause an error:

```
np.array([1,2,3,4,5]) #This works!  
np.array(1,2,3,4,5) #This will cause an error
```

In the two examples above, pay close attention to the syntax. The top example works properly because the integers are contained inside of a Python list. The second example causes an error because the integers are passed directly to `np.array()`, without enclosing them in a list.

Having said that, pay attention! Make sure that when you use `np.array()`, you're passing the values as a list.

Again, if you're confused about this or don't understand Python lists, I strongly recommend that you go back and review lists and other basic "built-in types" in Python.

Attributes of a Numpy array

Numpy arrays have a set of attributes that you can access. These attributes include things like the array's size, shape, number of dimensions, and data type.

Here's an abbreviated list of attributes of Numpy arrays:

Attribute	What it records
<code>shape</code>	The dimensions of the Numpy array
<code>size</code>	The total number of elements in the Numpy array
<code>ndim</code>	The number of dimensions of the array
<code>dtype</code>	The data type of the elements in the array
<code>itemsize</code>	The length of a single array element in bytes

I want to show you a few of these. To illustrate them, let's make a Numpy array and then investigate a few of its attributes.

Here, we'll create a simple Numpy array using `np.random.randint()`.

```
np.random.seed(72)
simple_array = np.random.randint(low = 0, high = 100, size=5)
```

`simple_array` is a Numpy array, and like all Numpy arrays, it has attributes.

You can access those attributes by using a dot after the name of the array, followed by the attribute you want to retrieve.

Here are some examples:

ndim

`ndim` is the number of dimensions.

```
simple_array.ndim
```

Which produces the output:

```
1
```

What this means is that `simple_array` is a 1-dimensional array.

shape

The `shape` attribute tells us the number of elements along each dimension.

```
simple_array.shape
```

With the output:

```
(5,)
```

What this is telling us is that `simple_array` has 5 elements along the first axis.

(And that's the only information provided, because `simple_array` is 1-dimensional.)

size

The `size` attribute tells you the total number of elements in a Numpy array.


```
simple_array.size
```

With the output:

```
5
```

This is telling us that `simple_array` has 5 total elements.

dtype (i.e., data type)

`dtype` tells you the type of data stored in the Numpy array.

Let's take a look. We can access the `dtype` parameter like this:

```
simple_array.dtype
```

Which produces the output:

```
dtype('int64')
```

This is telling us that `simple_array` contains integers.

Also remember: Numpy arrays contain data that are all of the same type.

Although we constructed `simple_array` to contain integers, but we could have created an array with floats or other numeric data types.

For example, we can create a Numpy array with decimal values (i.e., floats):

```
array_float = np.array([1.99,2.99,3.99] )  
array_float.dtype
```

Which gives the output:

```
dtype('float64')
```

When we construct the array with the above input values, you can see that `array_float` contains data of the float64 datatype (i.e., numbers with decimals).

Now that I've explained attributes, let's examine how to index Numpy arrays.

Indexing Numpy arrays

Indexing is very important for accessing and retrieving the elements of a Numpy array.

Recall what I wrote earlier:

A Numpy array is like a container with many compartments. Each of the compartments inside of a Numpy array have an “address.” We call that address an “index.”

Index:	0	1	2	3	4
Value:	88	19	46	74	94

Notice again that the index of the first value is 0.

We can use the index to retrieve specific values in the Numpy array. Let's take a look at how to do that.

First, let's create a Numpy array using the function `np.random.randint()`.

```
np.random.seed(72)
simple_array = np.random.randint(low = 0, high = 100, size=5)
```

You can print out the array with the following code:

```
print(simple_array)
```

And you can see that the array has 5 integers:

```
[88, 19, 46, 74, 94]
```

For the sake of clarity though, here's a visual representation of `simple_array`.

Looking at this will help you understand array indexing:

Index:	0	1	2	3	4
Value:	88	19	46	74	94

In this visual representation, you can see the values stored in the array: 88, 19, 46, 74, 94. But, I've also shown you the index values associated with each of those elements.

These indexes enable us to retrieve values in specific locations.

Let's take a look at how to do that.

Indexing a single element

The simplest form of indexing is retrieving a single value from the array.

To retrieve a single value from particular location in the Numpy array, you need to provide the "index" of that location.

Syntactically, you need to use bracket notation and provide the index inside of the brackets.

Let me show you an example. Above, we created the Numpy array

`simple_array`.

To get the value at index 1 from `simple_array`, you can use the following syntax:

```
# Retrieve the value at index 1
simple_array[1]
```

Which returns the value 19.

Visually though, we can represent this indexing action like this:

Index:	0	1	2	3	4
Value:	88	19	46	74	94

Essentially, we're using a particular index (i.e., the “address” of a particular location in the array) to retrieve the value stored at that location.

So the code `simple_array[1]` is basically saying, “give me the value that’s at index location 1.” The result is 19 ... 19 is the value at that index.

Negative index values

Numpy also supports negative index values. Using a negative index allows you to retrieve or reference locations starting from the end of the array.

Here’s an example:

```
simple_array[-1]
```

This retrieves the value at the very end of the array.

Index:	0	1	2	3	-1
Value:	88	19	46	74	94

We could also retrieve this value by using the index 4 (both will work). But sometimes you won't know exactly how long the array is. This is a convenient way to reference items at the end of a Numpy array.

Indexing multiple elements: AKA array slicing

I just showed you simple examples of array indexing, but array indexing can be quite complex.

It's actually possible to retrieve multiple elements from a Numpy array.

To do this, we still use bracket notation, but we can use a colon to specify a range of values. Here's an example:

```
simple_array[2:4]
```

This code is saying, "retrieve the values stored from index 2, up to but excluding index 4."

Visually, we can represent this as follows:

Index:	0	1	2	3	4
Value:	88	19	46	74	94

Indexing in 2-dimensional Numpy arrays

Now that you've learned how to use indexes in 1-dimensional Numpy arrays, let's review how to use indexes in 2-dimensional Numpy arrays.

Working with 2-d Numpy arrays is very similar to working with 1-d arrays. The major difference (with regard to indexes) is that 2-d arrays have 2 indexes, a row index and a column index.

To retrieve a value from a 2-d array, you need to provide the specific row and column indexes.

Here's an example. We'll create a 2-d Numpy array, and then we'll retrieve a value.

```
np.random.seed(72)
square_array = np.random.randint(low = 0
                                ,high = 100
                                ,size = 25).reshape([5,5])

square_array[2,1]
```

		Second index				
		0	1	2	3	4
First index	0	88	19	46	74	94
	1	69	79	26	7	29
	2	21	45	12	80	72
	3	28	53	65	26	64
	4	71	96	34	61	52

Here, we're essentially retrieving the value at row index 2 and column index 1.

The value at that position is 45.

This is fairly straightforward. The major challenge is that you need to remember that the row index is first and the column index is second.

Slicing 2-d Numpy arrays

Finally, let's review how to retrieve slices from 2-d Numpy arrays. Slicing 2-d arrays is very similar to slicing 1-d arrays. The major difference is that you need to provide 2 ranges, one for the rows and one for the columns.

```
np.random.seed(72)
square_array = np.random.randint(low = 0
                                ,high = 100
                                ,size = 25).reshape([5,5])

square_array[1:3,1:4]
```

		Second index				
		0	1	2	3	4
First index	0	88	19	46	74	94
	1	69	79	26	7	29
	2	21	45	12	80	72
	3	28	53	65	26	64
	4	71	96	34	61	52

Let's break this down.

We've again created a 5×5 square Numpy array called `square_array`.

Then, we took a slice of that array. The slice included the rows from index 1 up-to-and-excluding index 3. It also included the columns from index 1 up-to-and-excluding index 4.

This might seem a little confusing if you're a true beginner. In that case, I recommend working with 1-d arrays first, until you get the hang of them. Then, start working with relatively small 2-d Numpy arrays until you build your intuition about indexing works with 2-d arrays.

Numpy axes explained

This chapter will explain Numpy axes.

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

But 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

I'm going to be honest.

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 chapter will explain all the essentials that you need to know.

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 book, chances are you've taken more than a couple of math classes.

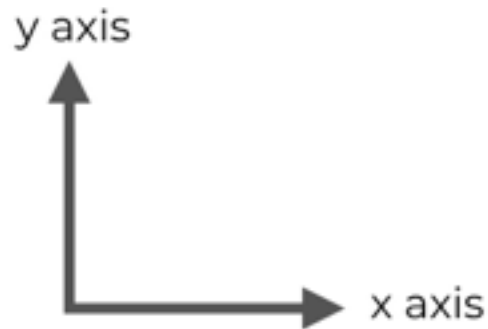
Think back to early math, when you were first learning about graphs.

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.

COORDINATE SYSTEMS HAVE AXES



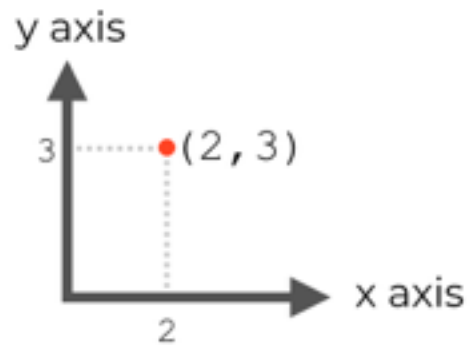
A simple 2-dimensional Cartesian coordinate system has two axes, the x axis and the y axis.

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 its 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.

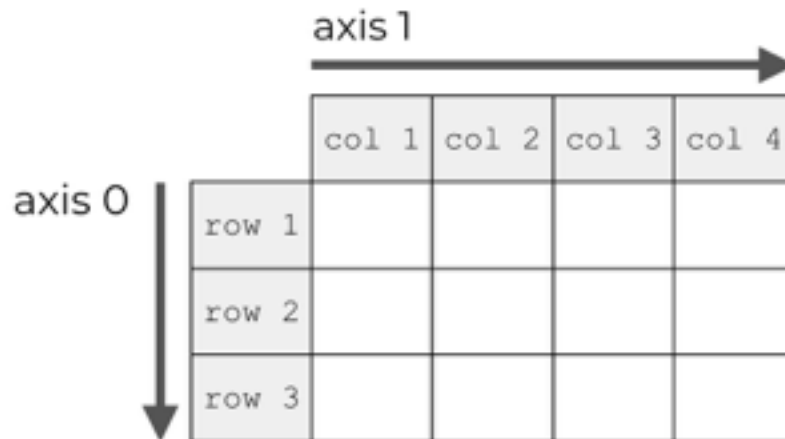
**POINTS CAN BE DEFINED
BY THEIR VALUES ALONG THE AXES**



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.

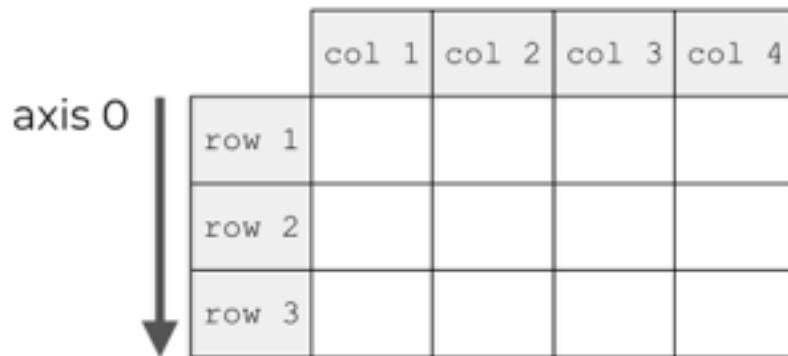


In a 2-dimensional Numpy array, the axes are the directions along the rows and columns.

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.



The diagram shows a 3x4 grid of cells. The first column is labeled 'row 1', 'row 2', and 'row 3' from top to bottom. The top row is labeled 'col 1', 'col 2', 'col 3', and 'col 4' from left to right. To the left of the grid, the text 'axis 0' is positioned above a downward-pointing arrow, indicating the vertical axis.

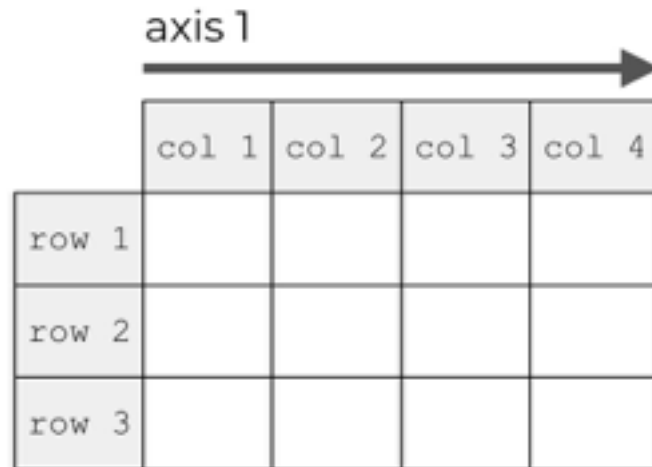
	col 1	col 2	col 3	col 4
row 1				
row 2				
row 3				

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.



The diagram shows a 2D array represented as a table. The columns are labeled 'col 1', 'col 2', 'col 3', and 'col 4'. The rows are labeled 'row 1', 'row 2', and 'row 3'. An arrow labeled 'axis 1' points to the right above the column headers. An arrow labeled 'axis 0' points downwards to the left of the row labels.

	col 1	col 2	col 3	col 4
row 1				
row 2				
row 3				

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.

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.

The axis parameter in 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])
```

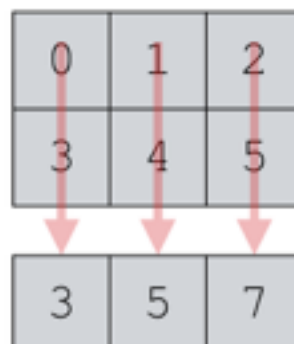
When we set `axis = 0`, the function actually sums down the columns. The result is a new Numpy array that contains the sum of each column. Why?

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*.

WHEN WE SET `axis = 0`, `np.sum()` COLLAPSES THE ROWS AND CALCULATES THE SUM



So when we set `axis = 0`, we're not summing across the rows. When we set `axis = 1`, 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)
```

OUT:

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

Next, 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.

WHEN WE SET `axis = 1`, `np.sum()` COLLAPSES THE COLUMNS AND CALCULATES THE SUM

0	1	2	→	3
3	4	5	→	12

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.

The axis parameter in 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:

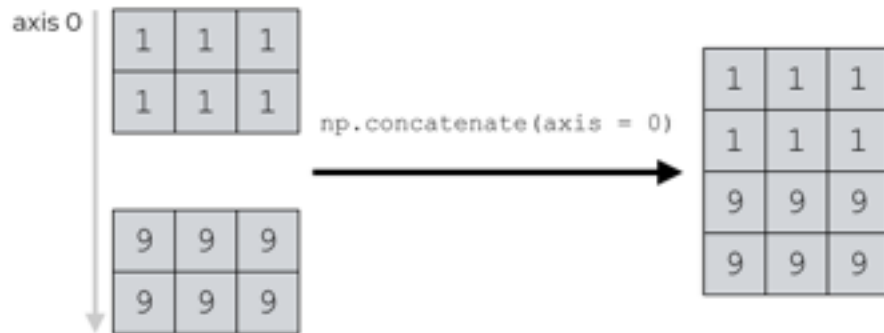
```
array([[1, 1, 1],  
       [1, 1, 1],  
       [9, 9, 9],  
       [9, 9, 9]])
```

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.

Setting `axis=0` concatenates along the row axis



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.

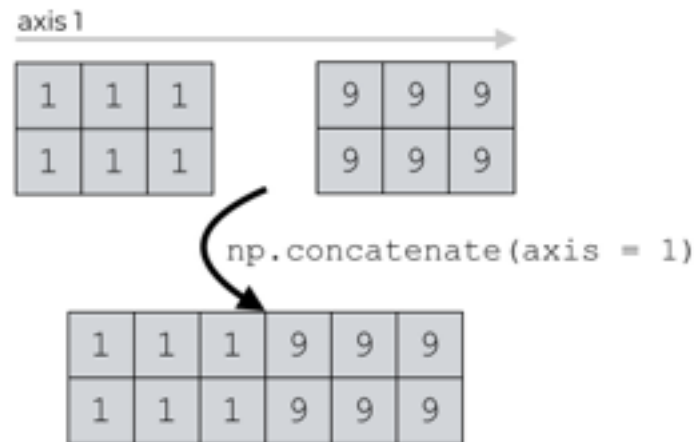
However, let's quickly review what's going on here.

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.

Setting `axis=1` concatenates along the column axis



Warning: 1-dimensional arrays work differently

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

But before I end this section, 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.

In a 1-d array, there is only 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!

How to use the Numpy arange function

The Numpy arange function (sometimes called `np.arange`) is a tool for creating numeric sequences in Python.

If you're learning data science in Python, the Numpy toolkit is important. The Numpy arange function is particularly important because it's very common; you'll see the `np.arange` function in a lot of data science code.

Having said that, this tutorial will show you how to use the Numpy arange function in Python.

It will explain how the syntax works. It will also show you some working examples of the `np.arange` function, so you can play with it and see how it operates.

Numpy arange creates sequences of evenly spaced values

The Numpy arange function returns evenly spaced numeric values within an interval, stored as a Numpy array (i.e., an `ndarray` object).

That might sound a little complicated, so let's look at a quick example.

We can call the `arange()` function like this:

```
numpy.arange(5)
```

Which will produce a Numpy array like this:



What happened here?

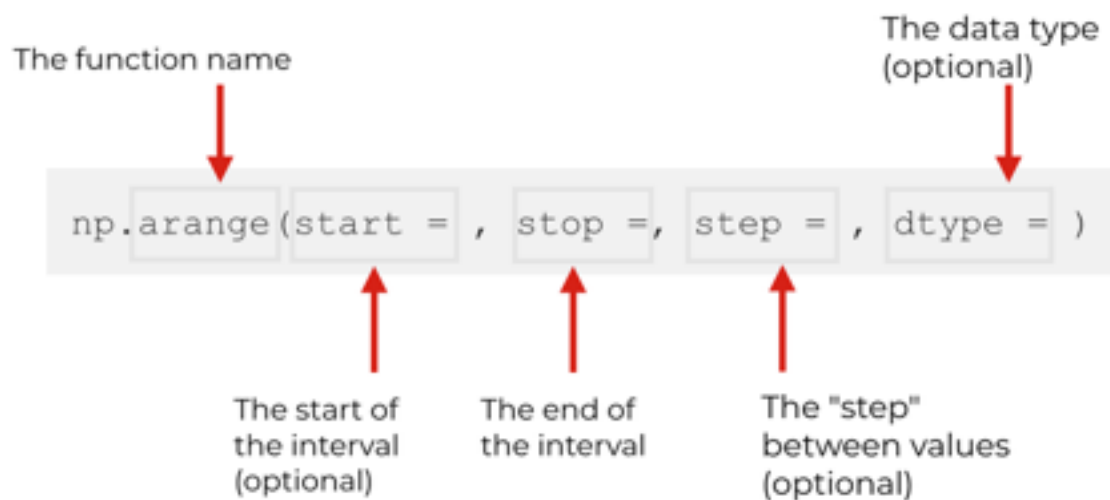
The `np.arange` function produced a sequence of 5 evenly spaced values from 0 to 4, stored as an `ndarray` object (i.e., a Numpy array).

Having said that, what's actually going on here is a little more complicated, so to fully understand the `np.arange` function, we need to examine the syntax. Once we look at the syntax, I'll show you more complicated examples which will make everything more clear.

The syntax of numpy arange

The syntax for Numpy arange is pretty straightforward.

Like essentially all of the Numpy functions, you call the function name and then there are a set of parameters that enable you to specify the exact behavior of the function.



Assuming that you've imported Numpy into your environment as `np`, you call the function with `np.arange()`.

Then inside of the `arange()` function, there are 4 parameters that you can modify:

- `start`
- `stop`
- `step`
- `dtype`

Let's take a look at each of those parameters, so you know what each one does.

`start` (optional)

The `start` parameter indicates the beginning value of the range.

This parameter is optional, so if you omit it, it will automatically default to 0.

`stop` (required)

The `stop` parameter indicates the end of the range. Keep in mind that like all Python indexing, this value will not be included in the resulting range. (The examples below will explain and clarify this point.) So essentially, the sequence of values will extend up to but excluding the stop value.

Additionally, this parameter is required, so you need to provide a stop value.

step (optional)

The `step` parameter specifies the spacing between values in the sequence.

This parameter is optional. If you don't specify a step value, by default the step value will be 1.

dtype (optional)

The `dtype` parameter specifies the data type.

Python and Numpy have a variety of data types that can be used here.

Having said that, if you don't specify a data type, it will be inferred based on the other arguments to the function.

Examples: how to use numpy arange

Now, let's work through some examples of how to use the Numpy `arange` function.

Before you start working through these examples though, make sure that you've imported Numpy into your environment using the following code:

```
# IMPORT NUMPY
import numpy as np
```

Ok, let's get started.

Create a simple Numpy arange sequence

First, let's use a simple case.

Here, we're going to create a simple Numpy array with 5 values.

```
np.arange(stop = 5)
```

Which produces something like the following array:

0	1	2	3	4
---	---	---	---	---

Notice a few things.

First, we didn't specify a start value. Because of this, the sequence starts at "0."

Second, when we used the code `stop = 5`, the "5" serves as the stop position.

This causes Numpy to create a sequence of values starting from 0 (the start value) up to but excluding this stop value.

Next, notice the spacing between values. The values are increasing in "steps" of 1. This is because we did not specify a value for the step parameter. If we don't specify a value for the step parameter, it defaults to 1.

Finally, the data type is integer. We didn't specify a data type with the `dtype` parameter, so Python has inferred the data type from the other arguments to the function.

One last note about this example. In this example, we've explicitly used `stop =` parameter. It's possible to remove the parameter itself, and just leave the argument, like this:

```
np.arange(5)
```

Here, the value “5” is treated a positional argument to the stop parameter.

Python “knows” that the value “5” serves as the stop point of the range. Having said that, I think it’s much clearer to explicitly use the parameter names. It’s clearer if you just type `np.arange(stop = 5)`.

Create a sequence in increments of 2

Now that you’ve seen a basic example, let’s look at something a little more complicated.

Here, we’re going to create a range of values from 0 to 8, in increments of 2.

To do this, we will use the start position of 0 and a stop position of 8. To increment in steps of 2, we’ll set the step parameter to 2.

```
np.arange(start = 0, stop = 8, step = 2)
```


The code creates a `ndarray` object like this:

0	2	4	6
---	---	---	---

Essentially, the code creates the following sequence of values stored as a Numpy array: 0, 2, 4, 6.

Let's take a step back and analyze how this worked. The output range begins at 0. This is because we set `start = 0`.

The output range then consists of values starting from 0 and incrementing in steps of 2: 2, 4, 6.

The range stops at 6. Why? We set the stop parameter to 8. Remember though, `numpy.arange()` will create a sequence up to but excluding the stop value. So once `arange()` gets to 6, the function can't go any further. If it attempts to increment by the step value of 2, it will produce the value of 8, which should be excluded, according to the syntax `stop = 8`. Again, `np.arange` will produce values up to but excluding the stop value.

Specify the data type for np.arange

As noted above, you can also specify the data type of the output array by using the `dtype` parameter.

Here's an example:

```
np.arange(start = 1, stop = 5, dtype = 'float')
```



First of all, notice the decimal point at the end of each number. This essentially indicates that these are floats instead of integers.

How did we create this? This is very straightforward, if you've understood the prior examples.

We've called the `np.arange` function starting from 1 and stopping at 5. And we've set the datatype to float by using the syntax `dtype = 'float'`.

Keep in mind that we used floats here, but we could have one of several different data types. Python and Numpy have a couple dozen different data types. These are all available when manipulating the dtype parameter.

Create a 2-dimensional array with np.arange

It's also possible to create a 2-dimensional Numpy array with `numpy.arange()`, but you need to use it in conjunction with the Numpy `reshape` method.

Ultimately, we're going to create a 3 by 3 array with the values from 1 to 9.

But before we do that, there's an "intermediate" step that you need to understand.

.... let's just create a 1-dimensional array with the values from 1 to 9:

```
np.arange(start = 1, stop = 10, step = 1)
```

This code creates a Numpy array like this:

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Notice how the code worked. The sequence of values starts at 1, which is the start value. The final value is 9. That's because we set the `stop` parameter equal to 10; the range of values will be up to but excluding the stop value. Because we're incrementing in steps of 1, the last value in the range will be 9.

Ok. Let's take this a step further to turn this into a 3 by 3 array.

Here, we're going to use the `reshape` method to re-shape this 1-dimensional array into a 2-dimensional array.

```
np.arange(start = 1, stop = 10, step = 1).reshape((3,3))
```

Notice that this is just a modification of the code we used a moment ago. We're using `np.arange(start = 1, stop = 10, step = 1)` to create a sequence of numbers from 1 to 9.

Then we're calling the `reshape()` method to re-shape the sequence of numbers. We're reshaping it from a 1-dimensional Numpy array, to a 2-dimensional array with 3 values along the rows and 3 values along the columns.

`np.arange` is common, so make sure you master it

In the last few examples, I've given you an overview of how the Numpy `arange` function works.

It's pretty straightforward once you understand the syntax, and it's not that hard to learn.

Having said that, make sure that you study and practice this syntax. Like all programming techniques, the key to mastery is repeated practice. Use the simple examples that I've shown above and try to recall the code. Practice and review this code until you know how the syntax works from memory.

How to use the Numpy linspace function

The Numpy linspace function (sometimes called `np.linspace`) is a tool in Python for creating numeric sequences.

It's somewhat similar to the Numpy arange function, in that it creates sequences of evenly spaced numbers structured as a Numpy array.

There are some differences though. Moreover, some people find the linspace function to be a little tricky to use.

It's not that hard to understand, but you really need to learn how it works.

That being said, this tutorial will explain how the Numpy linspace function works. It will explain the syntax, and it will also show you concrete examples of the function so you can see it in action.

Near the end of the chapter, this will also explain a little more about how `np.linspace` differs from `np.arange`.

Ok, first things first. Let's look a little more closely at what the `np.linspace` function does and how it works.

Numpy linspace creates sequences of evenly spaced values within an interval

The Numpy linspace function creates sequences of evenly spaced values within a defined interval.

Essentially, you specify a starting point and an ending point of an interval, and then specify the total number of breakpoints you want within that interval (including the start and end points). The `np.linspace` function will return a sequence of evenly spaced values on that interval.

To illustrate this, here's a quick example. (We'll look at more examples later, but this is a quick one just to show you what `np.linspace` does.)

```
np.linspace(start = 0, stop = 100, num = 5)
```

This code produces a Numpy array (an `ndarray` object) that looks like the following:

0	25	50	75	100
---	----	----	----	-----

That's the `ndarray` that the code produces, but we can also visualize the output like this:



there are 5 total items within the range
.... which corresponds to the `num` argument

So what's going on here?

Remember: the Numpy `linspace` function produces a evenly spaced observations within a defined interval.

We specified that interval with the `start` and `stop` parameters. In particular, this interval starts at 0 and ends at 100.

We also specified that we wanted 5 observations within that range. So, the `linspace` function returned an `ndarray` with 5 evenly spaced elements. The first element is 0. The last element is 100. The remaining 3 elements are evenly spaced between 0 and 100.

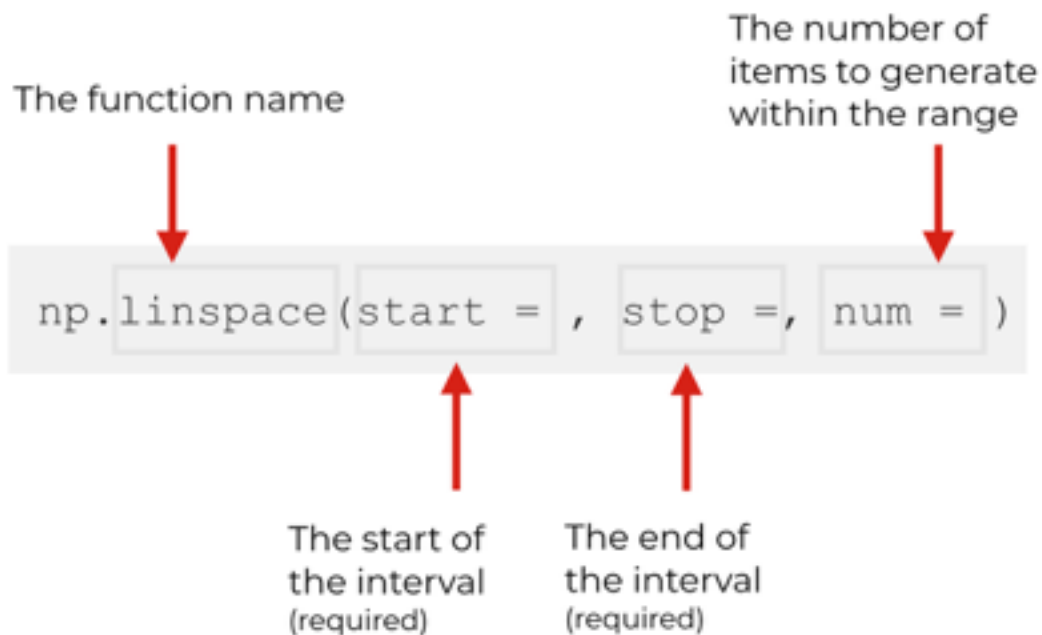
As should be expected, the output array is consistent with the arguments we've used in the syntax.

Having said that, let's look a little more closely at the syntax of the `np.linspace` function so you can understand how it works a little more clearly.

The syntax of Numpy linspace

The syntax of the Numpy `linspace` is very straightforward.

Obviously, when using the function, the first thing you need to do is call the function name itself:



To do this, you use the code `np.linspace` (assuming that you've imported Numpy as `np`).

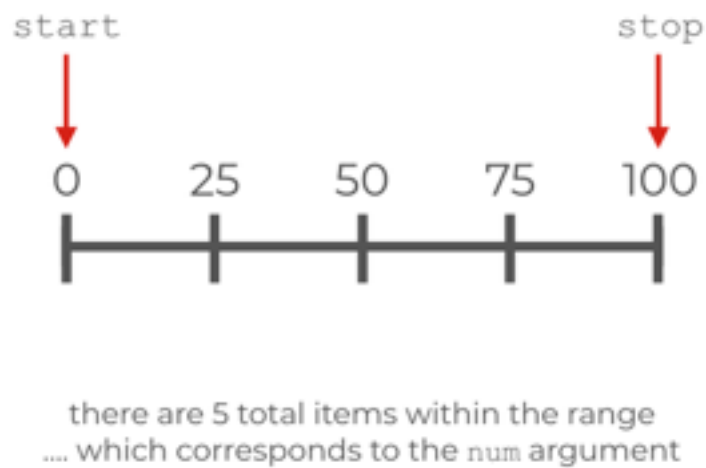
Inside of the `np.linspace` code above, you'll notice 3 parameters: `start`, `stop`, and `num`. These are 3 parameters that you'll use most frequently with the `linspace` function. There are also a few other optional parameters that you can use.

Let's talk about the parameters of `np.linspace`.

The parameters of Numpy linspace

There are several parameters that help you control the linspace function: `start`, `stop`, `num`, `endpoint`, and `dtype`.

To understand these parameters, let's take a look again at the following visual:



start

The `start` parameter is the beginning of the range of numbers.

So if you set `start = 0`, the first number in the new `nd.array` will be 0.

Keep in mind that this parameter is required.

stop

The `stop` parameter is the stopping point of the range of numbers.

In most cases, this will be the last value in the range of numbers. Having said that, if you modify the parameter and set `endpoint = False`, this value will not be included in the output array. (See the examples below to understand how this works.)

num (optional)

The `num` parameter controls how many total items will appear in the output array.

For example, if `num = 5`, then there will be 5 total items in the output array. If, `num = 10`, then there will be 10 total items in the output array, and so on.

This parameter is optional. If you don't provide a value for `num`, then `np.linspace` will use `num = 50` as a default.

endpoint (optional)

The `endpoint` parameter controls whether or not the stop value is included in the output array.

If `endpoint = True`, then the value of the `stop` parameter will be included as the last item in the `nd.array`.

If `endpoint = False`, then the value of the `stop` parameter will not be included.

By default, `endpoint` evaluates as `True`.

dtype (optional)

Just like in many other Numpy functions, with `np.linspace`, the `dtype` parameter controls the data type of the items in the output array.

If you don't specify a data type, Python will infer the data type based on the values of the other parameters.

If you do explicitly use this parameter, however, you can use any of the available data types from Numpy and base Python.

You don't need to use all of the parameters every time

Keep in mind that you won't use all of these parameters every time that you use the `np.linspace` function. Several of these parameters are optional.

Moreover, `start`, `stop`, and `num` are much more commonly used than `endpoint` and `dtype`.

Using the syntax without parameter values

Also keep in mind that you don't need to explicitly use the parameter names.

You can write code without the parameter names themselves; you can add the arguments as “positional arguments” to the function.

Here's an example:

```
np.linspace(0, 100, 5)
```

This code is functionally identical to the code we used in our previous examples:

```
np.linspace(start = 0, stop = 100, num = 5).
```

The main difference is that we did not explicitly use the `start`, `stop`, and `num` parameters. Instead, we provided arguments to those parameters by position. When you don't use the parameter names explicitly, Python knows that the first number (0) is supposed to be the start of the interval. It knows that 100 is supposed to be the stop. And it knows that the third number (5) corresponds to the `num` parameter. Again, when you don't explicitly use the parameter names, Python assigns the argument values to parameters strictly by position; which value appears first, second, third, etc.

You'll see people do this frequently in their code. People will commonly exclude the parameter names in their code and use positional arguments instead.

Although I realize that it's a little faster to write code with positional arguments, I think that it's clearer to actually use the parameter names. As a best practice, you should probably use them.

Examples: how to use numpy linspace

Now that you've learned how the syntax works, and you've learned about each of the parameters, let's work through a few concrete examples.

Create interval between 0 and 1, in breaks of .1

A quick example

```
np.linspace(start = 0, stop = 1, num = 11)
```

Which produces the output array:

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,
        0.9,  1. ])
```

An example like this would be useful if you're working with percents in some way. For example, if you were plotting percentages or plotting "accuracy" metrics for a machine learning classifier, you might use this code to construct part of your plot. Explaining how to do that is beyond the scope of this post, so I'll leave a deeper explanation of that for a future blog post.

Create interval between 0 and 100, in breaks of 10

A very similar example is creating a range of values from 0 to 100, in breaks of 10.

```
np.linspace(start = 0, stop = 100, num = 11)
```

The code for this is almost identical to the prior example, except we're creating values from 0 to 100.

Since it's somewhat common to work with data with a range from 0 to 100, a code snippet like this might be useful.

How to use the `endpoint` parameter

As mentioned earlier in this blog post, the `endpoint` parameter controls whether or not the stop value is included in the output array.

By default (if you don't set any value for endpoint), this parameter will have the default value of `True`. That means that the value of the `stop` parameter will be included in the output array (as the final value).

However, if you set `endpoint = False`, then the value of the `stop` parameter will not be included.

Here's an example.

In the following code, `stop` is set to 5.

```
np.linspace(start = 1, stop = 5, num = 4, endpoint = False)
```

But because we're also setting `endpoint = False`, 5 will not be included as the final value.

On the contrary, the output `nd.array` contains 4 evenly spaced values (i.e., `num = 4`), starting at 1, up to but excluding 5:

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

Personally, I find that it's a little un-intuitive to use `endpoint = False`, so I don't use it often. But if you have a reason to use it, this is how to do it.

Manually specify the data type

As mentioned earlier, the Numpy `linspace` function is supposed to “infer” the data type from the other input arguments. You'll notice that in many cases, the output is an array of floats.

If you want to manually specify the data type, you can use the `dtype` parameter.

This is very straightforward. Using the `dtype` parameter with `np.linspace` is identical to how you specify the data type with `np.array`, specify the data type with `np.arange`, etc.

Essentially, you use the `dtype` parameter and indicate the exact Python or Numpy data type that you want for the output array:

```
np.linspace(start = 0, stop = 100, num = 5, dtype = int)
```

In this case, when we set `dtype = int`, the `linspace` function produces an `nd.array` object with integers instead of floats.

Again, Python and Numpy have a variety of available data types, and you can specify any of these with the `dtype` parameter.

How `np.linspace` is different from `np.arange`

If you're familiar with Numpy, you might have noticed that `np.linspace` is rather similar to the `np.arange` function.

The essential difference between Numpy `linspace` and Numpy `arange` is that `linspace` enables you to control the precise end value, whereas `arange` gives you more direct control over the increments between values in the sequence.

To be clear, if you use them carefully, both `linspace` and `arange` can be used to create evenly spaced sequences. To a large extent, these are two similar different tools for creating sequences, and which you use will be a matter of preference. I personally find `np.arange` to be more intuitive, so I tend to prefer `arange` over `linspace`. Again though, this will mostly be a matter of preference, so try them both and see which you prefer.

How to Use the Numpy Sum Function

This section will show you how to use the Numpy sum function (sometimes called `np.sum`).

In the section, I'll explain what the function does. I'll also explain the syntax of the function step by step. Finally, I'll show you some concrete examples so you can see exactly how `np.sum` works.

Let's jump in.

Numpy sum adds up the values of a Numpy array

Let's very quickly talk about what the Numpy sum function does.

Essentially, the Numpy sum function sums up the elements of an array. It just takes the elements within a Numpy array (an `ndarray` object) and adds them together.

Having said that, it can get a little more complicated. It's possible to also add up the rows or add up the columns of an array. This will produce a new array object (instead of producing a scalar sum of the elements).

Further down in this tutorial, I'll show you examples of all of these cases, but first, let's take a look at the syntax of the `np.sum` function. You need to understand the syntax before you'll be able to understand specific examples.

The syntax of numpy sum

Like many of the functions of Numpy, the `np.sum` function is pretty straightforward syntactically.

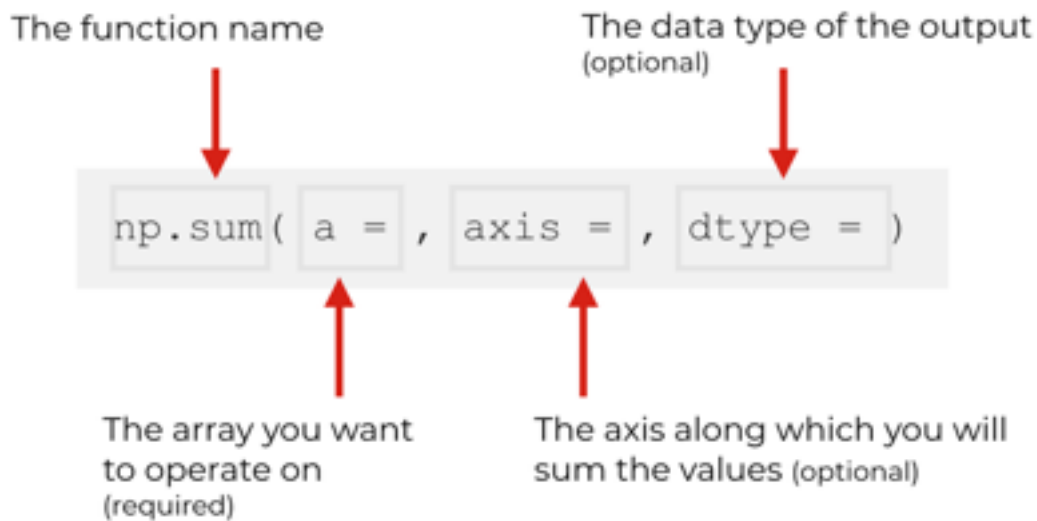
We typically call the function using the syntax `np.sum()`. Note that this assumes that you've imported numpy using the code `import numpy as np`.

Then inside of the `np.sum()` function there are a set of parameters that enable you to precisely control the behavior of the function.

Let's take a look.

The parameters of Numpy sum

The Numpy sum function has several parameters that enable you to control the behavior of the function.



Although technically there are 6 parameters, the ones that you'll use most often are `a`, `axis`, and `dtype`. I've shown those in the image above.

There are also a few others that I'll briefly describe.

Let's quickly discuss each parameter and what it does.

a (required)

The `a =` parameter specifies the input array that the `sum()` function will operate on. It is essentially the array of elements that you want to sum up.

Typically, the argument to this parameter will be a Numpy array (i.e., an `ndarray` object).

Having said that, technically the `np.sum` function will operate on any array like object. That means that in addition to operating on proper Numpy arrays, `np.sum` will also operate on Python tuples, Python lists, and other structures that are “array like.”

axis (optional)

The `axis` parameter specifies the axis or axes upon which the sum will be performed.

Does that sound a little confusing? Don't feel bad. Many people think that array axes are confusing ... particularly Python beginners.

I'll show you some concrete examples below. You can learn about Numpy axes elsewhere in this book, and the examples will clarify what an axis is, but let me very quickly explain.

The simplest example is an example of a 2-dimensional array.

When you're working with an array, each "dimension" can be thought of as an axis. This is sort of like the Cartesian coordinate system, which has an x-axis and a y-axis. The different "directions" – the dimensions – can be called axes.

Array objects have dimensions. For example, in a 2-dimensional Numpy array, the dimensions are the rows and columns. Again, we can call these dimensions, or we can call them axes.

Every axis in a numpy array has a number, starting with 0. In this way, they are similar to Python indexes in that they start at 0, not 1.

So the first axis is axis 0. The second axis (in a 2-d array) is axis 1. For multi-dimensional arrays, the third axis is axis 2. And so on.

Critically, you need to remember that the axis 0 refers to the rows. Axis 1 refers to the columns.

	col 1	col 2	col 3	col 4
row 1				
row 2				
row 3				

Why is this relevant to the Numpy sum function? It matters because when we use the `axis` parameter, we are specifying an axis along which to sum up the values.

So for example, if we set `axis = 0`, we are indicating that we want to sum up the rows. Remember, axis 0 refers to the row axis.

Likewise, if we set `axis = 1`, we are indicating that we want to sum up the columns. Remember, axis 1 refers to the column axis.

If you're still confused about this, don't worry. There is an example further down in this tutorial that will show you how axes work.

dtype (optional)

The `dtype` parameter enables you to specify the data type of the output of `np.sum`.

So for example, if you set `dtype = 'int'`, the `np.sum` function will produce a Numpy array of integers. If you set `dtype = 'float'`, the function will produce a Numpy array of floats as the output.

Python and Numpy have a variety of data types available, so review the documentation to see what the possible arguments are for the `dtype` parameter.

Note as well that the `dtype` parameter is optional.

out (optional)

The `out` parameter enables you to specify an alternative array in which to put the result computed by the `np.sum` function.

Note that the out parameter is optional.

keepdims (optional)

The `keepdims` parameter enables you to keep the number of dimensions of the output the same as the input.

This might sound a little confusing, so think about what `np.sum` is doing. When Numpy sum operates on an `ndarray`, it's taking a multi-dimensional object, and summarizing the values. It either sums up all of the values, in which case it collapses down an array into a single scalar value. Or (if we use the `axis` parameter), it reduces the number of dimensions by summing over one of the dimensions. In some sense, we're collapsing the object down.

More technically, we're reducing the number of dimensions. So by default, when we use the Numpy sum function, the output should have a reduced number of dimensions.

But, it's possible to change that behavior. If we set `keepdims = True`, the axes that are reduced will be kept in the output. So if you use `np.sum` on a 2-dimensional array and set `keepdims = True`, the output will be in the form of a 2-d array.

Still confused by this? Don't worry. I'll show you an example of how `keepdims` works below.

Note that the `keepdims` parameter is optional.

initial (optional)

The `initial` parameter enables you to set an initial value for the sum.

Note that the `initial` parameter is optional.

Examples: how to use the numpy sum function

Ok, now that we've examined the syntax, let's look at some concrete examples. I think that the best way to learn how a function works is to look at and play with very simple examples.

In these examples, we're going to be referring to the Numpy module as `np`, so make sure that you run this code:

```
import numpy as np
```

Sum the elements of a 1-d array with `np.sum`

Let's start with the simplest possible example.

We're going to create a simple 1-dimensional Numpy array using the `np.array` function.

```
np_array_1d = np.array([0,2,4,6,8,10])
```

If we print this out with `print(np_array_1d)`, you can see the contents of this ndarray:

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

Now that we have our 1-dimensional array, let's sum up the values.

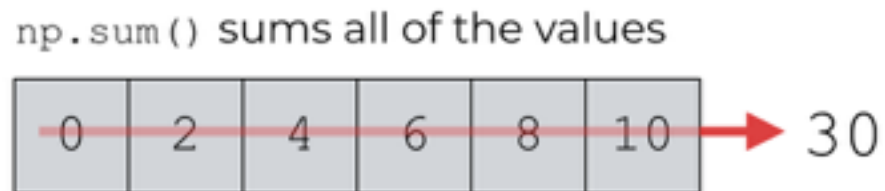
Doing this is very simple. We're going to call the Numpy sum function with the code `np.sum()`. Inside of the function, we'll specify that we want it to operate on the array that we just created, `np_array_1d`:

```
np.sum(np_array_1d)
```

Which will produce the following output:

```
30
```

Because `np.sum` is operating on a 1-dimensional Numpy array, it will just sum up the values. Visually, we can think of it like this:



Notice that we're not using any of the function parameters here. This is as simple as it gets.

When operating on a 1-d array, `np.sum` will basically sum up all of the values and produce a single scalar quantity ... the sum of the values in the input array.

Sum the elements of a 2-d array with `np.sum`

Next, let's sum all of the elements in a 2-dimensional Numpy array.

Syntactically, this is almost exactly the same as summing the elements of a 1-d array.

Basically, we're going to create a 2-dimensional array, and then use the Numpy sum function on that array.

Let's first create the 2-d array using the `np.array` function:

```
np_array_2x3 = np.array([[0,2,4],[1,3,5]])
```

The resulting array, `np_array_2x3`, is a 2 by 3 array; there are 2 rows and 3 columns.

If we print this out using `print(np_array_2x3)`, you can see the contents:


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

Next, we're going to use the `np.sum` function to add up all of the elements of the Numpy array.

This is very straight forward. We're just going to call `np.sum`, and the only argument will be the name of the array that we're going to operate on,

```
np_array_2x3:
```

```
np.sum(np_array_2x3)
```

When we run the code, it produces the following output:

```
15
```

Essentially, the Numpy sum function is adding up all of the values contained within `np_array_2x3`. When you add up all of the values (0, 2, 4, 1, 3, 5), the resulting sum is 15.

This is very straightforward. When you use the Numpy sum function without specifying an axis, it will simply add together all of the values and produce a single scalar value.



Having said that, it's possible to also use the `np.sum` function to add up the rows or add the columns.

Let's take a look at some examples of how to do that.

Sum down the rows with `np.sum`

Here, we're going to sum the rows of a 2-dimensional Numpy array.

First, let's just create the array:

```
np_array_2x3 = np.array([[0,2,4],[1,3,5]])
```

This is a simple 2-d array with 2 rows and 3 columns.

And if we print this out using `print(np_array_2x3)`, it will produce the following output:

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

Next, let's use the `np.sum` function to sum the rows.

```
np.sum(np_array_2x3, axis = 0)
```

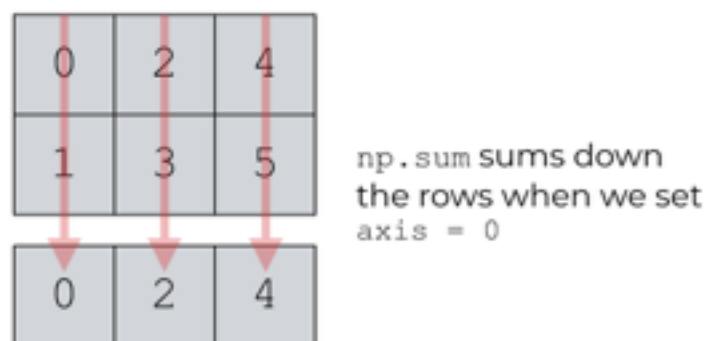
Which produces the following array:

```
array([1, 5, 9])
```

So what happened here?

When we use `np.sum` with the `axis` parameter, the function will sum the values along a particular axis.

In particular, when we use `np.sum` with `axis = 0`, the function will sum over the 0th axis (the rows). It's basically summing up the values row-wise, and producing a new array (with lower dimensions).



To understand this, refer back to the explanation of axes earlier in this book.

Remember: axes are like directions along a Numpy array. They are the dimensions of the array.

Specifically, axis 0 refers to the rows and axis 1 refers to the columns.

	axis 1			
	col 1	col 2	col 3	col 4
axis 0	row 1			
	row 2			
	row 3			

So when we use `np.sum` and set `axis = 0`, we're basically saying, "sum the rows." This is often called a row-wise operation.

Also note that by default, if we use `np.sum` like this on an n-dimensional Numpy array, the output will have the dimensions $n - 1$. So in this example, we used `np.sum` on a 2-d array, and the output is a 1-d array. (For more control over the

dimensions of the output array, see the example that explains the `keepdims` parameter.)

Sum across the columns with `np.sum`

Similar to adding the rows, we can also use `np.sum` to sum across the columns.

It works in a very similar way to our prior example, but here we will modify the `axis` parameter and set `axis = 1`.

First, let's create the array (this is the same array from the prior example, so if you've already run that code, you don't need to run this again):

```
np_array_2x3 = np.array([[0,2,4],[1,3,5]])
```

This code produces a simple 2-d array with 2 rows and 3 columns.

And if we print this out using `print(np_array_2x3)`, it will produce the following output:

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

Next, we're going to use the `np.sum` function to sum the columns.

```
np.sum(np_array_2x3, axis = 1)
```

Which produces the following array:

```
array([6, 9])
```

Essentially, the `np.sum` function has summed across the columns of the input array.

Visually, you can think of it like this:

`np.sum` sums across the columns
when we set `axis = 1`



Once again, remember: the “axes” refer to the different dimensions of a Numpy array. Axis 0 is the rows and axis 1 is the columns. So when we set the parameter `axis = 1`, we’re telling the `np.sum` function to operate on the columns only. Specifically, we’re telling the function to sum up the values across the columns.

How to use the `keepdims` parameter

In the last two examples, we used the `axis` parameter to indicate that we want to sum down the rows or sum across the columns.

Notice that when you do this it actually reduces the number of dimensions.

You can see that by checking the dimensions of the initial array, and the the dimensions of the output of `np.sum`.

So if we check the `ndim` attribute of `np_array_2x3` (which we created in our prior examples), you'll see that it is a 2-dimensional array:

```
np_array_2x3.ndim
```

Which produces the result 2. The array `np_array_2x3` is a 2-dimensional array.

Now, let's use the `np.sum` function to sum across the rows:

```
np_array_colsum = np.sum(np_array_2x3, axis = 1)
```

How many dimensions does the output have? Let's check the `ndim` attribute:

```
np_array_colsum.ndim
```

This produces the following output:

```
1
```

What that means is that the output array, `np_array_colsum`, has only 1 dimension. But the original array that we operated on, `np_array_2x3`, has 2 dimensions.

Why?

When we used `np.sum` with `axis = 1`, the function summed across the columns. Effectively, it collapsed the columns down to a single column!

This is an important point. By default, when we use the `axis` parameter, the `np.sum` function collapses the input from `n` dimensions and produces an output of lower dimensions.

The problem is, there may be situations where you want to keep the number of dimensions the same. If your input is n dimensions, you may want the output to also be n dimensions.

It's possible to create this behavior by using the `keepdims` parameter.

Here's an example. We're going to use `np.sum` to add up the columns by setting `axis = 1`. But we're also going to use the `keepdims` parameter to keep the dimensions of the output the same as the dimensions of the input:

```
np_array_colsum_keepdim = np.sum(np_array_2x3, axis = 1,  
keepdims = True)
```

If you take a look at the `ndim` attribute of the output array you can see that it has 2 dimensions:

```
np_array_colsum_keepdim.ndim
```

This will produce the following:

```
2
```

`np_array_colsum_keepdim` has 2 dimensions. It has the same number of dimensions as the input array, `np_array_2x3`.

To understand this better, you can also print the output array with the code `print(np_array_colsum_keepdim)`, which produces the following output:

```
[[6]
 [9]]
```

Essentially, `np_array_colsum_keepdim` is a 2-d numpy array organized into a single column.

This is a little subtle if you're not well versed in array shapes, so to develop your intuition, print out the array `np_array_colsum`. Remember, when we created `np_array_colsum`, we did not use `keepdims`:

```
print(np_array_colsum)
```

Here's the output of the print statement.

```
[6 9]
```

Do you see that the structure is different?

When we use `np.sum` on an axis without the `keepdims` parameter, it collapses at least one of the axes. But when we set `keepdims = True`, this will cause `np.sum` to produce a result with the same dimensions as the original input array.

Again, this is a little subtle. To understand it, you really need to understand the basics of Numpy arrays, Numpy shapes, and Numpy axes. So if you're a little confused, make sure that you study the basics of Numpy arrays ... it will make it much easier to understand the `keepdims` parameter.

Numpy random seed explained

In this section, I'll explain how to use the Numpy random seed function, which is also called `np.random.seed` or `numpy.random.seed`.

The function itself is extremely easy to use.

However, the reason that we need to use it is a little complicated. To understand why we need to use Numpy random seed, you actually need to know a little bit about pseudo-random numbers.

Numpy random seed is for pseudo-random numbers in Python

So what exactly is Numpy random seed?

Numpy random seed is simply a function that sets the random seed of the Numpy pseudo-random number generator. It provides an essential input that enables Numpy to generate pseudo-random numbers for random processes.

Does that make sense? Probably not.

Unless you have a background in computing and probability, what I just wrote is probably a little confusing.

Honestly, in order to understand “seeding a random number generator” you need to know a little bit about pseudo-random numbers.

That being the case, let me give you a quick introduction to them ...

A quick introduction to pseudo-random numbers

Here, I want to give you a very quick overview of pseudo-random numbers and why we need them.

Once you understand pseudo-random numbers, `numpy.random.seed` will make more sense.

WTF is a pseudo-random number?

At the risk of being a bit of a smart-ass, I think the name “pseudo-random number” is fairly self explanatory, and it gives us some insight into what pseudo-random numbers actually are.

Let's just break down the name a little.

A pseudo-random number is a *number*. A number that's sort-of random.

Pseudo-random.

So essentially, a pseudo-random number is a number that's almost random, but not really random.

It might sound like I'm being a bit sarcastic here, but that's essentially what they are. Pseudo-random numbers are numbers that appear to be random, but are not actually random.

In the interest of clarity though, let's see if we can get a definition that's a little more precise.

A proper definition of psuedo-random numbers

According to the encyclopedia at Wolfram Mathworld, a pseudo-random number is:

... a computer-generated random number.

(Source: <https://mathworld.wolfram.com/PseudorandomNumber.html>)

The definition goes on to explain that

The prefix pseudo- is used to distinguish this type of number from a “truly” random number generated by a random physical process such as radioactive decay.

A separate article at [random.org](https://www.random.org/randomness/) notes that pseudo-random numbers “appear random, but they are really predetermined”. (Source: <https://www.random.org/randomness/>)

Got that? Pseudo-random numbers are computer generated numbers that appear random, but are actually predetermined.

I think that these definitions help quite a bit, and they are a great starting point for understanding why we need them.

Why we need pseudo-random numbers

I swear to god, I'm going to bring this back to Numpy soon.

But, we still need to understand why pseudo-random numbers are required.

Really. Just bear with me. This will make sense soon.

A problem: computers are deterministic, not random

There's a fundamental problem when using computers to simulate or work with random processes.

Computers are completely deterministic, not random.

Setting aside some rare exceptions, computers are deterministic by their very design. To quote an article at MIT's School of Engineering "if you ask the same question you'll get the same answer every time."

(Source: <https://engineering.mit.edu/engage/ask-an-engineer/can-a-computer-generate-a-truly-random-number/>)

Another way of saying this is that if you give a computer a certain input, it will precisely follow instructions to produce an output.



... And if you later give a computer the same input, it will produce the same output.

If the input is the same, then the output will be the same.

THAT'S HOW COMPUTERS WORK.

The behavior of computers is *deterministic* ...

Essentially, the behavior of computers is NOT random.

This introduces a problem: how can you use a non-random machine to produce random numbers?

Pseudo-random numbers are generated by algorithms

Computers solve the problem of generating “random” numbers the same way that they solve essentially everything: with an algorithm.

Computer scientists have created a set of algorithms for creating psuedo random numbers, called “pseudo-random number generators.”

These algorithms can be executed on a computer.

As such, they are completely deterministic. However, the numbers that they produce have properties that approximate the properties of random numbers.

Pseudo-random numbers appear to be random

That is to say, the numbers generated by pseudo-random number generators appear to be random.

Even though the numbers they are completely determined by the algorithm, when you examine them, there is typically no discernible pattern.

For example, here we'll create some pseudo-random numbers with the Numpy randint function:

```
np.random.seed(1)
np.random.randint(low = 1, high = 10, size = 50))
```

OUT:

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

See any pattern here?

Me neither.

I can assure you though, that these numbers are not random, and are in fact completely determined by the algorithm. If you run the same code again, you'll get the exact same numbers.

Pseudo-random numbers can be re-created exactly

Importantly, because pseudo-random number generators are deterministic, they are also repeatable.

What I mean is that if you run the algorithm with the same input, it will produce the same output.

So you can use pseudo-random number generators to create and then re-create the exact same set of pseudo-random numbers.

Let me show you.

Generate pseudo-random integers

Here, we'll create a list of 5 pseudo-random integers between 0 and 9 using

`numpy.random.randint`.

(And notice that we're using `np.random.seed` here)

```
np.random.seed(0)
np.random.randint(10, size = 5)
```

This produces the following output:

```
array([5, 0, 3, 3, 7])
```

Simple. The algorithm produced an array with the values `[5, 0, 3, 3, 7]`.

Generate pseudo-random integers again

Ok.

Now, let's run the same code again.

... and notice that we're using `np.random.seed` in exactly the same way ...

```
np.random.seed(0)
np.random.randint(10, size = 5)
```

OUTPUT:

```
array([5, 0, 3, 3, 7])
```

We'll take a look at that ...

The numbers are the same.

We ran the exact same code, and it produced the exact same output.

I will repeat what I said earlier: pseudo random number generators produce numbers that look random, but are 100% determined.

Determined how though?

Remember what I wrote earlier: computers and algorithms process inputs into outputs. The outputs of computers depend on the inputs.

So just like any output produced by a computer, pseudo-random numbers are dependent on the input.

THIS is where `numpy.random.seed` comes in ...

The `numpy.random.seed` function provides the input (i.e., the seed) to the algorithm that generates pseudo-random numbers in Numpy.

How and why we use Numpy random seed

Ok, you got this far.

You're ready now.

Now you can learn about Numpy random seed.

`numpy.random.seed` provides an input to the pseudo-random number generator

What I wrote in the previous section is critical.

The “random” numbers generated by Numpy are not exactly random. They are pseudo-random ... they approximate random numbers, but are 100% determined by the input and the pseudo-random number algorithm.

The `np.random.seed` function provides an input for the pseudo-random number generator in Python.

That’s all the function does!

It allows you to provide a “seed” value to Numpy’s random number generator.

We use `numpy.random.seed` in conjunction with other numpy functions

Importantly, `numpy.random.seed` doesn't exactly work all on its own.

The `numpy.random.seed` function works in conjunction with other functions from Numpy.

Specifically, `numpy.random.seed` works with other function from the `numpy.random` namespace.

So for example, you might use `numpy.random.seed` along with `numpy.random.randint`. This will enable you to create random integers with Numpy.

You can also use `numpy.random.seed` with `numpy.random.normal` to create normally distributed numbers.

... or you can use it with `numpy.random.choice` to generate a random sample from an input.

In fact, there are several dozen Numpy random functions that enable you to generate random numbers, random samples, and samples from specific probability distributions.

I'll show you a few examples of some of these functions in the examples section of this tutorial.

Numpy random seed is deterministic

Remember what I said earlier in this tutorial pseudo-random number generators are completely deterministic. They operate by algorithm.

What this means is that if you provide the same seed, you will get the same output.

And if you change the seed, you will get a different output.

The output that you get depends on the input that you give it.

I'll show you examples of this behavior in the examples section.

Numpy random seed makes your code repeatable

The important thing about using a seed for a pseudo-random number generator is that it makes the code repeatable.

Remember what I said earlier?

... pseudo-random number generators operate by a deterministic process.

If you give a pseudo-random number generator the same input, you'll get the same output.

This can actually be a good thing!

There are times when you really want your “random” processes to be repeatable.

Code that has well defined, repeatable outputs is good for testing.

Essentially, we use Numpy random seed when we need to generate pseudo-random numbers in a repeatable way.

Numpy random seed makes your code easier to share

The fact that `np.random.seed` makes your code repeatable also makes it easier to share.

Take for example the tutorials that I post at sharpsightlabs.com.

I post detailed tutorials about how to perform various data science tasks, and I show how code works, step by step.

When I do this, it's important that people who read the tutorials and run the code get the same result. If a student reads the tutorial, and copy-and-pastes the code exactly, I want them to get the exact same result. This just helps them check their work! If they type in the code exactly as I show it in a tutorial, getting the exact same result gives them confidence that they ran the code properly.

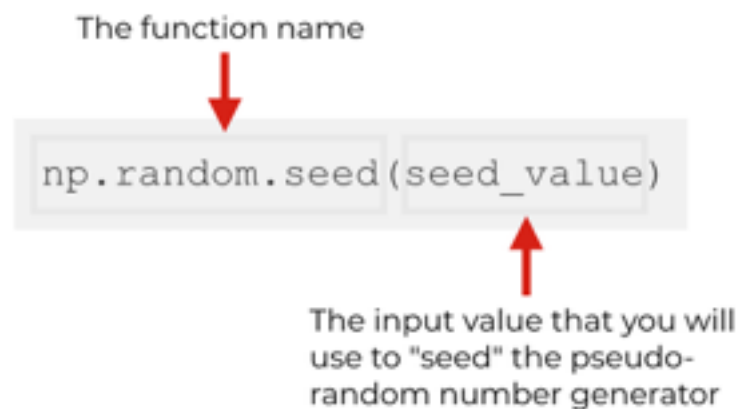
Again, in order to get repeatable results when we are using “random” functions in Numpy, we need to use `numpy.random.seed`.

Ok ... now that you understand what Numpy random seed is (and why we use it), let's take a look at the actual syntax.

The syntax of Numpy random seed

The syntax of Numpy random seed is extremely simple.

There's essentially only one parameter, and that is the seed value.



So essentially, to use the function, you just call the function by name and then pass in a “seed” value inside the parenthesis.

Note that in this syntax explanation, I'm using the abbreviation “np” to refer to Numpy. This is a common convention, but it requires you to import Numpy with the code “import numpy as np.” I'll explain more about this soon in the examples section.

Examples of `numpy.random.seed`

Let's take a look at some examples of how and when we use

`numpy.random.seed`.

Before we look at the examples though, you'll have to run some code.

Run this code first

To get the following examples to run properly, you'll need to import Numpy with the appropriate "nickname."

You can do that by executing the following code:

```
import numpy as np
```

Running this code will enable us to use the alias `np` in our syntax to refer to

`numpy`.

This is a common convention in Numpy. When you read Numpy code, it is extremely common to see Numpy referred to as `np`. If you're a beginner you might not realize that you need to import Numpy with the code `import numpy as np`, otherwise the examples won't work properly!

Now that we've imported Numpy properly, let's start with a simple example. We'll generate a single random number between 0 and 1 using Numpy random random.

Generate a random number with `numpy.random.random`

Here, we're going to use Numpy to generate a random number between zero and one. To do this, we're going to use the Numpy random random function (AKA, `np.random.random`).

Ok, here's the code:

```
np.random.seed(0)
np.random.random()
```

OUTPUT:

```
0.5488135039273248
```

Note that the output is a float. It's a decimal number between 0 and 1.

For the record, we can essentially treat this number as a probability. We can think of the `np.random.random` function as a tool for generating probabilities.

Rerun the code

Now that I've shown you how to use `np.random.random`, let's just run it again with the same seed.

Here, I just want to show you what happens when you use `np.random.seed` before running `np.random.random`.

```
np.random.seed(0)
np.random.random()
```

OUTPUT:

```
0.5488135039273248
```

Notice that the number is exactly the same as the first time we ran the code.

Essentially, if you execute a Numpy function with the same seed, you'll get the same result.

Generate a random integer with `numpy.random.randint`

Next, we're going to use `np.random.seed` to set the number generator before using Numpy random `randint`.

Essentially, we're going to use Numpy to generate 5 random integers between 0 and 99.

```
np.random.seed(74)
np.random.randint(low = 0, high = 100, size = 5)
```

OUTPUT:

```
array([30, 91,  9, 73, 62])
```

This is pretty simple.

Numpy random seed sets the seed for the pseudo-random number generator, and then Numpy random randint selects 5 numbers between 0 and 99.

Run the code again

Let's just run the code so you can see that it reproduces the same output if you have the same seed.

```
np.random.seed(74)
np.random.randint(low = 0, high = 100, size = 5)
```

OUTPUT:

```
array([30, 91,  9, 73, 62])
```

Once again, as you can see, the code produces the same integers if we use the same seed. As noted previously in the tutorial, Numpy random randint doesn't exactly produce "random" integers. It produces pseudo-random integers that are completely determined by `numpy.random.seed`.

Select a random sample from an input array

It's also common to use the NP random seed function when you're doing random sampling.

Specifically, if you need to generate a reproducible random sample from an input array, you'll need to use `numpy.random.seed`.

Let's take a look.

Here, we're going to use `numpy.random.seed` before we use `numpy.random.choice`. The Numpy random choice function will then create a random sample from a list of elements.

```
np.random.seed(0)
np.random.choice(a = [1,2,3,4,5,6], size = 5)
```

OUTPUT:

```
array([5, 6, 1, 4, 4])
```

As you can see, we've basically generated a random sample from the list of input elements ... the numbers 1 to 6.

In the output, you can see that some of the numbers are repeated. This is because `np.random.choice` is using random sampling with replacement.

Rerun the code

Let's quickly re-run the code.

I want to re-run the code just so you can see, once again, that the primary reason we use Numpy random seed is to create results that are completely repeatable.

Ok, here is the exact same code that we just ran (with the same seed).

```
np.random.seed(0)
np.random.choice(a = [1,2,3,4,5,6], size = 5)
```

OUTPUT:

```
array([5, 6, 1, 4, 4])
```

Once again, we used the same seed, and this produced the same output.

Frequently asked questions about `np.random.seed`

Now that we've taken a look at some examples of using Numpy random seed to set a random seed in Python, I want to address some frequently asked questions.

What does `np.random.seed(0)` do?

Dude. I just wrote 2000 words explaining what the `np.random.seed` function does ... which basically explains what `np.random.seed(0)` does.

Ok, ok ... I get it. You might still have questions.

I'll summarize.

We use `np.random.seed` when we need to generate random numbers or mimic random processes in Numpy.

Computers are generally deterministic, so it's very difficult to create truly "random" numbers on a computer. Computers get around this by using pseudo-random number generators.

These pseudo-random number generators are algorithms that produce numbers that appear random, but are not really random.

In order to work properly, pseudo-random number generators require a starting input. We call this starting input a “seed.”

The code `np.random.seed(0)` enables you to provide a seed (i.e., the starting input) for Numpy’s pseudo-random number generator.

Numpy then uses the seed and the pseudo-random number generator in conjunction with other functions from the `numpy.random` namespace to produce certain types of random outputs.

Ultimately, creating pseudo-random numbers this way leads to repeatable output, which is good for testing and code sharing.

Having said all of that, to really understand `numpy.random.seed`, you need to have some understanding of pseudo-random number generators.

... so if what I just wrote doesn’t make sense, please return to the top of the page and read the f*#^ing tutorial.

What number should I use in numpy random seed?

Basically, it doesn't matter.

You can use `numpy.random.seed(0)`, or `numpy.random.seed(42)`, or any other number.

For the most part, the number that you use inside of the function doesn't really make a difference.

You just need to understand that using different seeds will cause Numpy to produce different pseudo-random numbers. The output of a `numpy.random` function will depend on the seed that you use.

Here's a quick example. We're going to use Numpy random seed in conjunction with Numpy random randint to create a set of integers between 0 and 99.

In the first example, we'll set the seed value to 0.

```
np.random.seed(0)
np.random.randint(99, size = 5)
```

Which produces the following output:

```
array([44, 47, 64, 67, 67])
```

Basically, `np.random.randint` generated an array of 5 integers between 0 and 99. Note that if you run this code again with the exact same seed (i.e. 0), you'll get the same integers from `np.random.randint`.

Next, let's run the code with a different seed.

```
np.random.seed(1)
np.random.randint(99, size = 5)
```

OUTPUT:

```
array([37, 12, 72,  9, 75])
```

Here, the code for `np.random.randint` is exactly the same ... we only changed the seed value. Here, the seed is 1.

With a different seed, Numpy random randint created a different set of integers. Everything else is the same. The code for `np.random.randint` is the same. But with a different seed, it produces a different output.

Ultimately, I want you to understand that the output of a `numpy.random` function ultimately depends on the value of `np.random.seed`, but the choice of seed value is sort of arbitrary.

Do I always need to use numpy random seed?

The short answer is, no.

If you use a function from the `numpy.random` namespace (like `np.random.randint`, `np.random.normal`, etc) without using Numpy random seed first, Python will actually still use `numpy.random.seed` in the background. Numpy will generate a seed value from a part of your computer system (like `/urandom` on a Unix or Linux machine).

So essentially, if you don't set a seed with `numpy.random.seed`, Numpy will set one for you.

However, this has a disadvantage!

If you don't explicitly set a seed, your code will not have repeatable outputs. Numpy will generate a seed on its own, but that seed might change moment to moment. This will make your outputs different every time you run it.

So to summarize: you don't absolutely have to use `numpy.random.seed`, but you should use it if you want your code to have repeatable outputs.

What's the difference between `np.random.seed` and `np.random.RandomState`?

Ok.

We're really getting into the weeds here.

Essentially, `numpy.random.seed` sets a seed value for the global instance of the `numpy.random` namespace.

On the other hand, `np.random.RandomState` returns one instance of the `RandomState` and does not effect the global `RandomState`.

Confused?

That's okay this answer is a little technical and it requires you to know a little about how Numpy is structured on the back end. It also requires you to know a little bit about programming concepts like "global variables." If you're a relative data science beginner, the details that you need to know might be over your head.

The important thing is that Numpy random seed is probably sufficient if you're just using Numpy for some data science or scientific computing.

However, if you're building software systems that need to be secure, Numpy random seed is probably not the right tool.

To summarize, `np.random.seed` is probably fine if you're just doing simple analytics, data science, and scientific computing, but you need to learn more about `RandomState` if you want to use the Numpy pseudo-random number generator in systems where security is a consideration.

How to use numpy random normal in Python

This tutorial will cover the Numpy random normal function (AKA, `np.random.normal`).

If you're doing any sort of statistics or data science in Python, you'll often need to work with random numbers. And in particular, you'll often need to work with normally distributed numbers.

The Numpy random normal function generates a sample of numbers drawn from the normal distribution, otherwise called the Gaussian distribution.

This tutorial will show you how the function works, and will show you how to use the function.

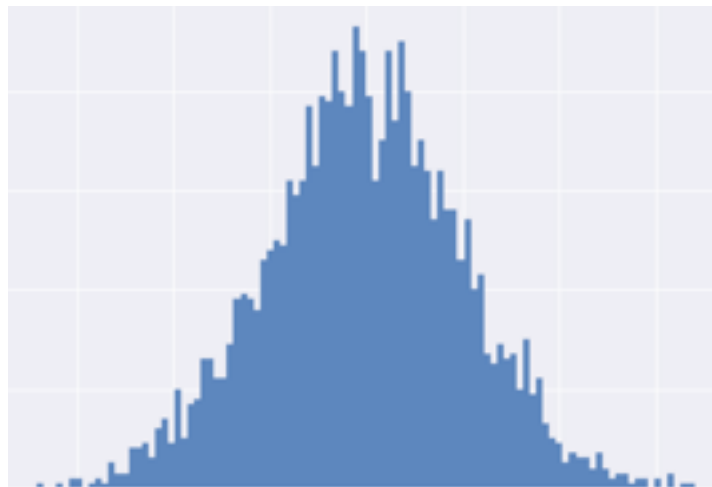
Numpy random normal generates normally distributed numbers

As you've learned in this book, Numpy is a package for working with numerical data. Where does `np.random.normal` fit in?

As I mentioned previously, Numpy has a variety of tools for working with numerical data. In most cases, Numpy's tools enable you to do one of two things: create numerical data (structured as a Numpy array), or perform some calculation on a Numpy array.

The Numpy random normal function enables you to create a Numpy array that contains normally distributed data.

Hopefully you're familiar with normally distributed data, but just as a refresher, here's what it looks like when we plot it in a histogram:



Normally distributed data is shaped sort of like a bell, so it's often called the "bell curve."

Now that I've explained what the `np.random.normal` function does at a high level, let's take a look at the syntax.

The syntax of numpy random normal

The syntax of the Numpy random normal function is fairly straightforward.

Note that in the following illustration and throughout this blog post, we will assume that you've imported Numpy with the following code: `import numpy as np`. That code will enable you to refer to Numpy as `np`.

Let's take a quick look at the syntax.

An explanation of the syntax of the numpy random normal function.

Let me explain this. Typically, we will call the function with the name `np.random.normal()`. As I mentioned earlier, this assumes that we've imported Numpy with the code `import numpy as np`.

Inside of the function, you'll notice 3 parameters: `loc`, `scale`, and `size`.

Let's talk about each of those parameters.

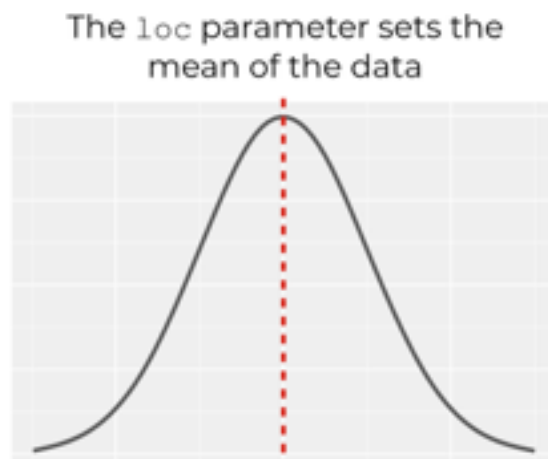
The parameters of the `np.random.normal` function

The `np.random.normal` function has three primary parameters that control the output: `loc`, `scale`, and `size`.

I'll explain each of those parameters separately.

`loc`

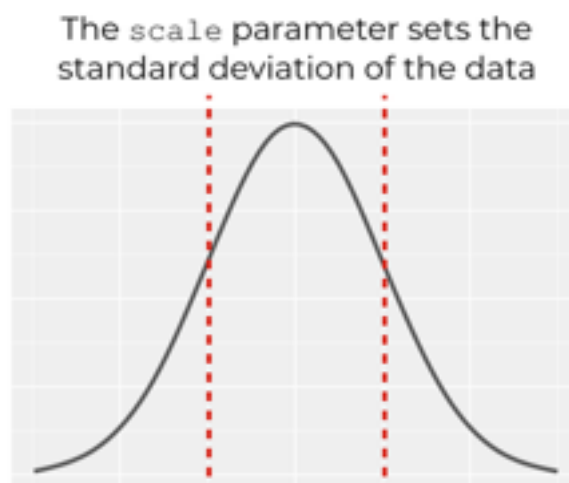
The `loc` parameter controls the mean of the function.



This parameter defaults to 0, so if you don't use this parameter to specify the mean of the distribution, the mean will be at 0.

scale

The `scale` parameter controls the standard deviation of the normal distribution.



By default, the `scale` parameter is set to 1.

size

The `size` parameter controls the size and shape of the output.

Remember that the output will be a Numpy array. Numpy arrays can be 1-dimensional, 2-dimensional, or multi-dimensional (i.e., 2 or more).

The argument that you provide to the `size` parameter will dictate the size and shape of the output array.

If you provide a single integer, `x`, `np.random.normal` will provide `x` random normal values in a 1-dimensional Numpy array.

You can also specify a more complex output.

For example, if you specify `size = (2, 3)`, `np.random.normal` will produce a numpy array with 2 rows and 3 columns. It will be filled with numbers drawn from a random normal distribution.

Keep in mind that you can create output arrays with more than 2 dimensions, but in the interest of simplicity, I will leave that to another tutorial.

The `np.random.randn` function

There's another function that's similar to `np.random.normal`. It's called `np.random.randn`.

Just like `np.random.normal`, the `np.random.randn` function produces numbers that are drawn from a normal distribution.

The major difference is that `np.random.randn` is like a special case of `np.random.normal`.

`np.random.randn` operates like `np.random.normal` with `loc = 0` and `scale = 1`.

So this code:

```
np.random.seed(1)
np.random.normal(loc = 0, scale = 1, size = (3,3))
```

Operates effectively the same as this code:

```
np.random.seed(1)
np.random.randn(3, 3)
```

Examples: how to use the numpy random normal function

Now that I've shown you the syntax the numpy random normal function, let's take a look at some examples of how it works.

Run this code before you run the examples

Before you work with any of the following examples, make sure that you run the following code:

```
import numpy as np
```

Ok, now let's work with some examples.

Draw a single number from the normal distribution

First, let's take a look at a very simple example.

Here, we're going to use `np.random.normal` to generate a single observation from the normal distribution.

```
np.random.normal(1)
```

This code will generate a single number drawn from the normal distribution with a mean of 0 and a standard deviation of 1.

Essentially, this code works the same as `np.random.normal(size = 1, loc = 0, scale = 1)`. Remember, if we don't specify values for the `loc` and `scale` parameters, they will default to `loc = 0` and `scale = 1`.

Draw 5 numbers from the normal distribution

Now, let's draw 5 numbers from the normal distribution.

This code will look almost exactly the same as the code in the previous example.


```
np.random.normal(5)
```

Here, the value 5 is the value that's being passed to the size parameter. It essentially indicates that we want to produce a Numpy array of 5 values, drawn from the normal distribution.

Note as well that because we have not explicitly specified values for `loc` and `scale`, they will default to `loc = 0` and `scale = 1`.

Create a 2-dimensional Numpy array of normally distributed values

Now, we'll create a 2-dimensional array of normally distributed values.

To do this, we need to provide a tuple of values to the `size` parameter.

```
np.random.seed(42)
np.random.normal(size = (2, 3))
```

Which produces the output:

```
array([[ 1.62434536, -0.61175641, -0.52817175],  
       [-1.07296862,  0.86540763, -2.3015387 ]])
```

So we've used the `size` parameter with the `size = (2, 3)`. This has generated a 2-dimensional Numpy array with 6 values. This output array has 2 rows and 3 columns.

To be clear, you can use the `size` parameter to create arrays with even higher dimensional shapes.

Generate normally distributed values with a specific mean

Now, let's generate normally distributed values with a specific mean.

To do this, we'll use the `loc` parameter. Recall from earlier in the tutorial that the `loc` parameter controls the mean of the distribution from which we draw the numbers with `np.random.normal`.

Here, we're going to set the mean of the data to 50 with the syntax `loc = 50`.

```
np.random.seed(42)
np.random.normal(size = 1000, loc = 50)
```

The full array of values is too large to show here, but here are the first several values of the output:

```
array([ 50.49671415,  49.8617357 ,  50.64768854,  51.52302986,
        49.76584663,  49.76586304,  51.57921282,  50.76743473,
        49.53052561,  50.54256004,  49.53658231,  49.53427025
        ...
```

You can see at a glance that these values are roughly centered around 50. If you were to calculate the average using the numpy mean function, you would see that the mean of the observations is in fact 50.

Generate normally distributed values with a specific standard deviation

Next, we'll generate an array of values with a specific standard deviation.

As noted earlier in the blog post, we can modify the standard deviation by using the `scale` parameter.

In this example, we'll generate 1000 values with a standard deviation of 100.

```
np.random.seed(42)
np.random.normal(size = 1000, scale = 100)
```

And here is a truncated output that shows the first few values:

```
array([ 4.96714153e+01, -1.38264301e+01,  6.47688538e+01,
        1.52302986e+02, -2.34153375e+01, -2.34136957e+01,
        1.57921282e+02,  7.67434729e+01, -4.69474386e+01,
        ...])
```

Notice that we set `size = 1000`, so the code will generate 1000 values. I've only shown the first few values for the sake of brevity.

It's a little difficult to see how the data are distributed here, but we can use the `std()` method to calculate the standard deviation:

```
np.random.seed(42)
np.random.normal(size = 1000, scale = 100).std()
```

Which produces the following:

```
99.695552529463015
```

If we round this up, it's essentially 100.

Notice that in this example, we have not used the `loc` parameter. Remember that by default, the `loc` parameter is set to `loc = 0`, so by default, this data is centered around 0. We could modify the `loc` parameter here as well, but for the sake of simplicity, I've left it at the default.

How to use the `loc` and `scale` parameter in `np.random.normal`

Let's do one more example to put all of the pieces together.

Here, we'll create an array of values with a mean of 50 and a standard deviation of 100.

```
np.random.seed(42)
np.random.normal(size = 1000, loc = 50, scale = 100)
```

I won't show the output of this operation I'll leave it for you to run it yourself.

Let's quickly discuss the code. If you've read the previous examples in this tutorial, you should understand this.

We're defining the mean of the data with the `loc` parameter. The mean of the data is set to 50 with `loc = 50`.

We're defining the standard deviation of the data with the `scale` parameter.

We've done that with the code `scale = 100`.

The code `size = 1000` indicates that we're creating a Numpy array with 1000 values.

That's it. That's how you can use the Numpy random normal function to create normally distributed data in Python.

Do you want to master Numpy?

In this book, I've shown you the basics of how to use Numpy.

But there's really a lot more to learn.

... we've just scratched the surface on Numpy.

Moreover, there's a lot more to learn if you want to master data science in Python, more generally.

If and when you're ready to take the next step, you should enroll in one of our premium, online video courses.

You can find the full list of courses here:

<https://www.sharpsightlabs.com/course-directory/>