

[Home](#) > [About Python](#) > [Learn Python](#)

Python Numpy Array Tutorial

A NumPy tutorial for beginners in which you'll learn how to create a NumPy array, use broadcasting, access values, manipulate arrays, and much more.

Updated Feb 2023 · 45 min read



Karlijn Willems

Former Data Journalist at DataCamp | Manager at NextWave Consulting

TOPICS

Python

Data Science

Data Analysis

NumPy is, just like SciPy, Scikit-Learn, pandas, and similar packages. They are the Python packages that you just can't miss when you're learning data science, mainly because this library provides you with an array data structure that holds some benefits over Python lists, such as being more compact, faster access in reading and writing items, being more convenient and more efficient.

This NumPy will focus precisely on this. It will not only show you what NumPy arrays actually are and how you can install Python, but you'll also learn how to make arrays (even when your data comes from files), how broadcasting works, how you can ask for help, how to manipulate your arrays and how to visualize them.

If you want to know even more about NumPy arrays and the other data structures that you will need in your data science journey, consider taking a look at DataCamp's [Intro to Python for Data Science](#), which has a chapter on NumPy.

What is a Python Numpy Array?

You already read in the introduction that NumPy arrays are a bit like Python lists, but still very much different at the same time. For those of you who are new to the topic, let's clarify what it exactly is and what it's good for.

As the name gives away, a NumPy array is a central data structure of the numpy library. The library's name is short for "Numeric Python" or "Numerical Python."

In other words, NumPy is a Python library that is the core library for scientific computing in Python. It contains a collection of tools and techniques that can be used to solve mathematical models of problems in science and engineering.

One of these tools is a high-performance multidimensional array object that is a powerful data structure for the efficient computation of arrays and matrices. To work with these arrays, there's a vast amount of high-level mathematical functions operate on these matrices and arrays.

But what is an array?



When you look at the print of a couple of arrays, you could see it as a grid that contains values of the same type:

```
import numpy as np
# Define a 1D array
my_array = np.array([[1, 2, 3, 4],
                     [5, 6, 7, 8]],
                    dtype=np.int64)

# Define a 2D array
my_2d_array = np.array([[1, 2, 3, 4],
                        [5, 6, 7, 8]],
                       dtype=np.int64)

# Define a 3D array
my_3d_array = np.array([[[1, 2, 3, 4],
                         [5, 6, 7, 8]],
                        [[1, 2, 3, 4],
                         [9, 10, 11, 12]]],
                       dtype=np.int64)

# Print the 1D array
print("Printing my_array:")
print(my_array)
# Print the 2D array
print("Printing my_2d_array:")
print(my_2d_array)
# Print the 3D array
print("Printing my_3d_array:")
print(my_3d_array)
```



Explain code

OpenAI

You see that, in the example above, the data are integers. The array holds and represents any regular data in a structured way.

However, you should know that, on a structural level, an array is basically nothing but pointers. It's a combination of a memory address, a data type, a shape, and strides:

- The data pointer indicates the memory address of the first byte in the array.
- The data type or dtype pointer describes the kind of elements that are contained within the array.
- The shape indicates the shape of the array.
- The strides are the number of bytes that should be skipped in memory to go to the next element. If your strides are (10,1), you need to proceed one byte to get to the next column and 10 bytes to locate the next row.

In other words, an array contains information about the raw data, how to locate an element, and how to interpret an element.

Enough of the theory. Let's check this out ourselves:

You can easily test this by exploring the numpy array attributes:

```
import numpy as np
my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Print out memory address
print(my_2d_array.data)

# Print out the shape of `my_array`
print(my_2d_array.shape)

# Print out the data type of `my_array`
print(my_2d_array.dtype)
```

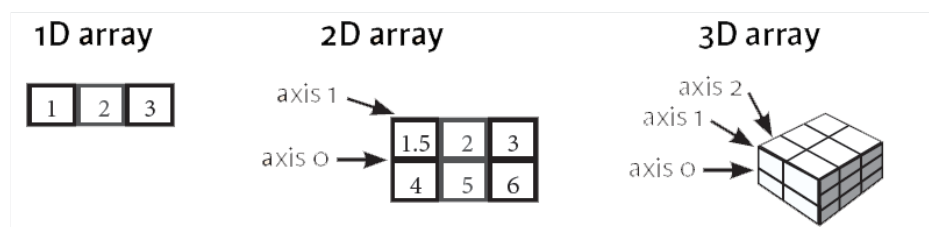


```
# Print out the stride of `my_array`
print(my_2d_array.strides)
```

[🔗 Explain code](#)
[OpenAI](#)

You see that now, you get a lot more information: for example, the data type that is printed out is 'int64' or signed 32-bit integer type; This is a lot more detailed! That also means that the array is stored in memory as 64 bytes (as each integer takes up 8 bytes and you have an array of 8 integers). The strides of the array tell us that you have to skip 8 bytes (one value) to move to the next column, but 32 bytes (4 values) to get to the same position in the next row. As such, the strides for the array will be (32,8).

Note that if you set the data type to int32, the strides tuple that you get back will be (16, 4), as you will still need to move one value to the next column and 4 values to get the same position. The only thing that will have changed is the fact that each integer will take up 4 bytes instead of 8



The array that you see above is, as its name already suggested, a 2-dimensional array: you have rows and columns. The rows are indicated as the “axis 0”, while the columns are the “axis 1”. The number of the axis goes up accordingly with the number of the dimensions: in 3-D arrays, of which you have also seen an example in the previous code chunk, you’ll have an additional “axis 2”. Note that these axes are only valid for arrays that have at least 2 dimensions, as there is no point in having this for 1-D arrays;

These axes will come in handy later when you’re manipulating the shape of your NumPy arrays.

How to Install Numpy

Before you can start to try out these NumPy arrays for yourself, you first have to make sure that you have it installed locally (assuming that you’re working on your pc). If you have the Python library already available, go ahead and skip this section :)

If you still need to set up your environment, you must be aware that there are two major ways of installing NumPy on your pc: with the help of Python wheels or the Anaconda Python distribution.

Install With Python Wheels

Make sure firstly that you have Python installed. You can check out our guide on [how to install Python](#) if you still need to do this :)

If you’re working on Windows, make sure that you have added Python to the PATH environment variable. Then, don’t forget to install a package manager, such as pip, which will ensure that you’re able to use Python’s open-source libraries.

Note that recent versions of Python 3 come with pip, so double-check if you have it, and if you do, upgrade it before you install NumPy:

```
pip install pip --upgrade
pip --version
```

[🔗](#)
[🔗 Explain code](#)
[OpenAI](#)

Download the appropriate NumPy wheel file for your system from the PyPI website. You can find the list of available NumPy wheels on the [Python Package Index](#).

Once you have downloaded the NumPy wheel file, navigate to the directory where the file is saved using the command prompt or terminal window.

Install NumPy using the following command, replacing "numpy-<version>-<architecture>.whl" with the name of the NumPy wheel file you downloaded:

```
pip install numpy-<version>-<architecture>.whl
```



Explain code

OpenAI

For example, if you downloaded NumPy version 1.20.3 for a 64-bit Windows system, the command would be:

```
pip install numpy-1.20.3-cp39-cp39-win_amd64.whl
```



Explain code

OpenAI

After the installation is complete, you can verify that NumPy is installed by opening a Python interpreter and running the following command:

```
import numpy
```



Explain code

OpenAI

If there are no error messages, NumPy is installed and ready to use!

Install With the Anaconda Python Distribution

To get NumPy, you could also download the Anaconda Python distribution. This is easy and will allow you to get started quickly! If you haven't downloaded it already, go to the [official page](#) to get it. Follow the instructions to install, and you're ready to start!

Do you wonder why this might actually be easier?

The good thing about getting this Python distribution is the fact that you don't need to worry too much about separately installing NumPy or any of the major packages that you'll be using for your data analyses, such as pandas, scikit-learn, etc.

Because, especially if you're very new to Python, programming or terminals, it can really come as a relief that Anaconda already includes **100 of the most popular** Python, R, and Scala packages for data science. But also for more seasoned data scientists, Anaconda is the way to go if you want to get started quickly on tackling data science problems.

What's more, Anaconda also includes several open-source development environments, such as Jupyter and Spyder. If you'd like to start working with Jupyter Notebook after this tutorial, go to our [Jupyter notebook tutorial](#).

In short, consider downloading Anaconda to get started on working with numpy and other packages that are relevant to data science!

How to Make NumPy Arrays

So, now that you have set up your environment, it's time for the real work. Admittedly, you have already tried out some stuff with arrays in the code above. However, you haven't really gotten any real hands-on practice with them because you first needed to install NumPy on

your own PC. Now that you have done this, it's time to see what you need to do in order to run the above code chunks on your own.

To make a numpy array, you can just use the `np.array()` function. All you need to do is pass a list to it, and optionally, you can also specify the data type of the data. If you want to know more about the possible data types that you can pick, go to [this guide](#) or consider taking a brief look at DataCamp's [NumPy cheat sheet](#).

There's no need to go and memorize these NumPy data types if you're a new user, but you do have to know and care what data you're dealing with. The data types are there when you need more control over how your data is stored in memory and on disk. Especially in cases where you're working with extensive data, it's good that you know to control the storage type.

Don't forget that, in order to work with the `np.array()` function, you need to make sure that the numpy library is present in your environment.

The NumPy library follows an import convention: when you import this library, you have to make sure that you import it as `np`. By doing this, you'll make sure that other Pythonistas understand your code more easily.

In the following example, you'll create the `my_array` array that you have already played around with above:

```
import numpy as np

# Make the array `my_array`
my_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Print `my_array`
print(my_array)
```

 Explain code

 OpenAI

If you would like to know more about how to make lists, check out our [Python list questions tutorial](#).

However, sometimes you don't know what data you want to put in your array, or you want to import data into a numpy array from another source. In those cases, you'll make use of initial placeholders or functions to load data from text into arrays, respectively.

The following sections will show you how to do this.

How to make an “Empty” NumPy array

What people often mean when they say that they are creating “empty” arrays is that they want to make use of initial placeholders, which you can fill up afterward. You can initialize arrays with ones or zeros, but you can also create arrays that get filled up with evenly spaced values, constant or random values.

However, you can still make a totally empty array, too.

Luckily for us, there are quite a lot of functions to make.

Try it all out below!

```
import numpy as np

# Create an array of ones
ones_array = np.ones((3, 4))
print("Ones Array:")
print(ones_array)
print()

# Create an array of zeros
zeros_array = np.zeros((2, 3, 4), dtype=np.int16)
print("Zeros Array:")
```

```

print(zeros_array)
print()

# Create an array with random values
random_array = np.random.random((2, 2))
print("Random Array:")
print(random_array)
print()

# Create an empty array
empty_array = np.empty((3, 2))
print("Empty Array:")
print(empty_array)
print()

# Create a full array
full_array = np.full((2, 2), 7)
print("Full Array:")
print(full_array)
print()

# Create an array of evenly-spaced values
arange_array = np.arange(10, 25, 5)
print("Arange Array:")
print(arange_array)
print()

# Create an array of evenly-spaced values
np.linspace(0,2,9)

```

[🔗 Explain code](#)
 OpenAI

Tip: play around with the above functions so that you understand how they work!

- For some, such as `np.ones()`, `np.random.random()`, `np.empty()`, `np.full()` or `np.zeros()` the only thing that you need to do in order to make arrays with ones or zeros is pass the shape of the array that you want to make. As an option to `np.ones()` and `np.zeros()`, you can also specify the data type. In the case of `np.full()`, you also have to specify the constant value that you want to insert into the array.
- With `np.linspace()` and `np.arange()` you can make arrays of evenly spaced values. The difference between these two functions is that the last value of the three that are passed in the code chunk above designates either the step value for `np.linspace()` or a number of samples for `np.arange()`. What happens in the first is that you want, for example, an array of 9 values that lie between 0 and 2. For the latter, you specify that you want an array to start at 10 and per steps of 5, generate values for the array that you're creating.

Remember that NumPy also allows you to create an identity array or matrix with `np.eye()` and `np.identity()`. An identity matrix is a square matrix of which all elements in the principal diagonal are ones, and all other elements are zeros. When you multiply a matrix with an identity matrix, the given matrix is left unchanged.

In other words, if you multiply a matrix by an identity matrix, the resulting product will be the same matrix again by the standard conventions of matrix multiplication.

Even though the focus of this tutorial is not on demonstrating how identity matrices work, it suffices to say that identity matrices are useful when you're starting to do matrix calculations: they can simplify mathematical equations, which makes your computations more efficient and robust.

How to Load NumPy Arrays From Text

Creating arrays with the help of initial placeholders or with some example data is an excellent way of getting started with numpy. But when you want to get started with data analysis, you'll need to load data from text files.

With that what you have seen up until now, you won't really be able to do much. Make use of some specific functions to load data from your files, such as `loadtxt()` or `genfromtxt()`.

Let's say you have the following text files with data, you'd want to copy-paste the commented values in the code below, paste them into the text file, and save it as "data.txt." You can then use the code below:

```
# This is your data in the text file
# Value1 Value2 Value3
# 0.2536 0.1008 0.3857
# 0.4839 0.4536 0.3561
# 0.1292 0.6875 0.5929
# 0.1781 0.3049 0.8928
# 0.6253 0.3486 0.8791

# Import your data
x, y, z = np.loadtxt('data.txt',
                    skiprows=1,
                    unpack=True)
```



Explain code

OpenAI

In the code above, you use `loadtxt()` to load the data in your environment. You see that the first argument that both functions take is the text file `data.txt`. Next, there are some specific arguments for each: in the first statement, you skip the first row, and you return the columns as separate arrays with `unpack=True`. This means that the values in column `Value1` will be put in `x`, and so on.

Note that, in case you have comma-delimited data or if you want to specify the data type, there are also the arguments `delimiter` and `dtype` that you can add to the `loadtxt()` arguments.

That's easy and straightforward, right?

Let's take a look at your second file with data:

```
# Your data in the text file
# Value1 Value2 Value3
# 0.4839 0.4536 0.3561
# 0.1292 0.6875 MISSING
# 0.1781 0.3049 0.8928
# MISSING 0.5801 0.2038
# 0.5993 0.4357 0.7410

my_array2 = np.genfromtxt('data2.txt',
                        skip_header=1,
                        filling_values=-999)
```



Explain code

OpenAI

You see that here, you resort to `genfromtxt()` to load the data. In this case, you have to handle some missing values that are indicated by the 'MISSING' strings.

Since the `genfromtxt()` function converts character strings in numeric columns to `nan`, you can convert these values to other ones by specifying the `filling_values` argument. In this case, you choose to set the value of these missing values to `-999`.

If, by any chance, you have values that don't get converted to `nan` by `genfromtxt()`, there's always the `missing_values` argument that allows you to specify what the missing values of your data exactly are.

But this is not all.

Tip: check out [this numpy.genfromtxt page](#) to see what other arguments you can add to import your data successfully.

You now might wonder what the difference between these two functions really is.

The examples indicated this maybe implicitly, but, in general, `genfromtxt()` gives you a little bit more flexibility; It's more robust than `loadtxt()`.

Let's make this difference a little bit more practical: the latter, `loadtxt()`, only works when each row in the text file has the same number of values, so when you want to handle missing values easily, you'll typically find it easier to use `genfromtxt()`.

But this is definitely not the only reason.

A brief look at the number of arguments that `genfromtxt()` has to offer will teach you that there are really many more things that you can specify in your import, such as the maximum number of rows to read or the option to automatically strip white spaces from variables.

How to Save NumPy Arrays

Once you have done everything that you need to do with your arrays, you can also save them to a file. If you want to save the array to a text file, you can use the `savetxt()` function to do this:

```
import numpy as np
x = np.arange(0.0, 5.0, 1.0)
np.savetxt('test.out', x, delimiter=',')
```



Explain code

OpenAI

Remember that `np.arange()` creates a NumPy array of evenly-spaced values. The third value that you pass to this function is the step value.

There are, of course, other ways to save your NumPy arrays to text files. Check out the functions in the table below if you want to get your data to binary files or archives:

<code>save()</code>	Save an array to a binary file in NumPy .npy format
<code>savez()</code>	Save several arrays into an uncompressed .npz archive
<code>savez_compressed()</code>	Save several arrays into a compressed .npz archive

For more information or examples of how you can use the above functions to save your data, check this [NumPy reference page](#) or make use of one of the help functions that NumPy has to offer to get to know more instantly!

Are you not sure what these NumPy help functions are?

No worries! You'll learn more about them in one of the next sections!

How to Inspect your NumPy Arrays

Besides the array attributes that have been mentioned above, namely, data, shape, dtype and strides, there are some more that you can use to easily get to know more about your arrays. The ones that you might find interesting to use when you're just starting out are the following:

```
import numpy as np

# Create a numpy array with shape (2,4) and dtype int64
my_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Print the number of dimensions of `my_array`
print("Number of dimensions of my_array:")
print(my_array.ndim)
print()
```




```
# Print the number of elements in `my_array`
print("Number of elements in my_array:")
print(my_array.size)
print()

# Print information about the memory layout of `my_array`
print("Information about the memory layout of my_array:")
print(my_array.flags)
print()

# Print the length of one array element in bytes
print("Length of one array element in bytes:")
print(my_array.itemsize)
print()

# Print the total consumed bytes by `my_array`'s elements
print("Total consumed bytes by my_array's elements:")
print(my_array.nbytes)
print()
```

 Explain code

 OpenAI

These are almost all the attributes that an array can have.

Don't worry if you don't feel that all of them are useful for you at this point. This is fairly normal because, just like you read in the previous section, you'll only get to worry about memory when you're working with large data sets.

Also note that, besides the attributes, you also have some other ways of gaining more information on and even tweaking your array slightly:

```
import numpy as np
my_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Print the length of `my_array`
print(len(my_array))

# Change the data type of `my_array`
my_array.astype(float)
```

 Explain code

 OpenAI

Now that you have made your array, either by making one yourself with the `np.array()` or one of the initial placeholder functions, or by loading in your data through the `loadtxt()` or `genfromtxt()` functions, it's time to look more closely into the second key element that really defines the NumPy library: scientific computing.

How NumPy Broadcasting Works

Before you go deeper into scientific computing, it might be a good idea to first go over what broadcasting exactly is: it's a mechanism that allows NumPy to work with arrays of different shapes when you're performing arithmetic operations.

The infographic below shows an example of NumPy broadcasting in action:

Food	Fats (g)	Protein (g)	Carbs(g)
Apple	0.8	2.9	3.9
Banana	52.4	23.6	36.5
Raw Almond	55.2	31.7	23.9
Cookies	14.4	11	4.9

[3, 3, 8]

Food	Fats (g)	Protein (g)	Carbs(g)
Apple	2.4	8.7	31.2
Banana	157.2	70.8	292
Raw Almond	165.6	95.1	191.2
Cookies	43.2	33	39.2

[Source](#)

To put it in a more practical context, you often have an array that's somewhat larger and another one that's slightly smaller. Ideally, you want to use the smaller array multiple times to perform an operation (such as a sum, multiplication, etc.) on the larger array.

To do this, you use the broadcasting mechanism.

However, there are some rules if you want to use it. And, before you already sigh, you'll see that these "rules" are very simple and kind of straightforward!

- First off, to make sure that the broadcasting is successful, the dimensions of your arrays need to be compatible. Two dimensions are compatible when they are equal. Consider the following example:

```
# Import the NumPy library and give it an alias of `np`
import numpy as np

# Initialize a 3x4 array of ones and assign it to the variable `x`
x = np.ones((3,4))

# Print the shape of the array `x`
print("Shape of x:", x.shape)

# Initialize a 3x4 array of random numbers between 0 and 1 and assign it to the variable `y`
y = np.random.random((3,4))

# Print the shape of the array `y`
print("Shape of y:", y.shape)

# Add the arrays `x` and `y` element-wise and print the resulting array
z = x + y
print("Result of x + y:\n", z)

# Print the shape of the resulting array
print("Shape of x + y:", z.shape)
```

 Explain code

 OpenAI

- Two dimensions are also compatible when one of them is 1:

```
# Import the NumPy library and give it an alias of `np`
import numpy as np

# Initialize a 3x4 array of ones and assign it to the variable `x`
x = np.ones((3,4))
```

```
# Print the shape of the array `x`
print("Shape of x:", x.shape)

# Initialize a 3x4 array of random numbers between 0 and 1 and assign it to the variable `y`
y = np.random.random((3,4))

# Print the shape of the array `y`
print("Shape of y:", y.shape)

# Add the arrays `x` and `y` element-wise and print the resulting array
z = x + y
print("Result of x + y:\n", z)

# Print the shape of the resulting array
print("Shape of x + y:", z.shape)
```

 Explain code

 OpenAI

Note that if the dimensions are not compatible, you will get a `ValueError`.

Tip: also test what the size of the resulting array is after you have done the computations! You'll see that the size is actually the maximum size along each dimension of the input arrays.

In other words, you see that the result of `x+y` gives an array with shape `(3,4)`: `y` had a shape of `(4,)` and `x` had a shape of `(3,4)`. The maximum size along each dimension of `x` and `y` is taken to make up the shape of the new, resulting array.

- Lastly, the arrays can only be broadcast together if they are compatible in all dimensions. Consider the following example:

```
# Import `numpy` as `np`
import numpy as np

# Initialize `x` and `y`
x = np.ones((3,4))
y = np.random.random((5,1,4))

# Add `x` and `y`
z = x + y
```

 Explain code

 OpenAI

You see that, even though `x` and `y` seem to have somewhat different dimensions, the two can be added together.

That is because they are compatible in all dimensions:

- Array `x` has dimensions `3 X 4`,
- Array `y` has dimensions `5 X 1 X 4`

Since you have seen above that dimensions are also compatible if one of them is equal to 1, you see that these two arrays are indeed a good candidate for broadcasting!

What you will notice is that in the dimension where `y` has size 1, and the other array has a size greater than 1 (that is, 3), the first array behaves as if it were copied along that dimension.

Note that the shape of the resulting array will again be the maximum size along each dimension of `x` and `y`: the dimension of the result will be `(5,3,4)`

In short, if you want to make use of broadcasting, you will rely a lot on the shape and dimensions of the arrays with which you're working.

But what if the dimensions are not compatible?

What if they are not equal or if one of them is not equal to 1?

You'll have to fix this by manipulating your array! You'll see how to do this in one of the next sections.

How do Array Mathematics Work?

You've seen that broadcasting is handy when you're doing arithmetic operations. In this section, you'll discover some of the functions that you can use to do mathematics with arrays.

As such, it probably won't surprise you that you can just use `+`, `-`, `*`, `/` or `%` to add, subtract, multiply, divide or calculate the remainder of two (or more) arrays. However, a big part of why NumPy is so handy, is because it also has functions to do this. The equivalent functions of the operations that you have seen just now are, respectively, `np.add()`, `np.subtract()`, `np.multiply()`, `np.divide()` and `np.remainder()`.

You can also easily do exponentiation and taking the square root of your arrays with `np.exp()` and `np.sqrt()`, or calculate the sines or cosines of your array with `np.sin()` and `np.cos()`. Lastly, it's also useful to mention that there's also a way for you to calculate the natural logarithm with `np.log()` or calculate the dot product by applying the `dot()` to your array.

Try it all out in the code below.


Just a tip: make sure to check out first the arrays that have been loaded for this exercise!


```
# Import `numpy` as `np`
import numpy as np
x = np.array([[1, 2, 3], [3, 4, 5]])
y = np.array([6,7,8])

# Add `x` and `y`
z = np.add(x,y)
print("Addition of x and y:\n", z)

# Subtract `x` and `y`
z = np.subtract(x,y)
print("Subtraction of y from x:\n", z)

# Multiply `x` and `y`
z = np.multiply(x,y)
print("Element-wise multiplication of x and y:\n", z)
```

 Explain code

 OpenAI

Remember how broadcasting works? Check out the dimensions and the shapes of both `x` and `y` in your IPython shell. Are the rules of broadcasting respected?

But there is more.

Check out this small list of aggregate functions:

<code>a.sum()</code>	Array-wise sum
<code>a.min()</code>	Array-wise minimum value
<code>b.max(axis=0)</code>	Maximum value of an array row
<code>b.cumsum(axis=1)</code>	Cumulative sum of the elements
<code>a.mean()</code>	Mean

<code>a.sum()</code>	Array-wise sum
<code>b.median()</code>	Median
<code>a.corrcoef()</code>	Correlation coefficient
<code>np.std(b)</code>	Standard deviation

Besides all of these functions, you might also find it useful to know that there are mechanisms that allow you to compare array elements. For example, if you want to check whether the elements of two arrays are the same, you might use the `==` operator. To check whether the array elements are smaller or bigger, you use the `<` or `>` operators.

This all seems quite straightforward, yes?

However, you can also compare entire arrays with each other! In this case, you use the `np.array_equal()` function. Just pass in the two arrays that you want to compare with each other, and you're done.

Note that, besides comparing, you can also perform logical operations on your arrays. You can start with `np.logical_or()`, `np.logical_not()` and `np.logical_and()`. This basically works like your typical OR, NOT and AND logical operations;

In the simplest example, you use OR to see whether your elements are the same (for example, 1), or if one of the two array elements is 1. If both of them are 0, you'll return FALSE. You would use AND to see whether your second element is also 1 and NOT to see if the second element differs from 1.

Test this out in the code chunk below:

```
# Import `numpy` as `np`
import numpy as np
# Initialize arrays
a = np.array([1, 1, 0, 0], dtype=bool)
b = np.array([1, 0, 1, 0], dtype=bool)

# `a` AND `b`
np.logical_and(a, b)

# `a` OR `b`
np.logical_or(a, b)

# `a` NOT `b`
np.logical_not(a, b)
```



Explain code

OpenAI

How to Subset, Slice, and Index Arrays

Besides mathematical operations, you might also consider taking just a part of the original array (or the resulting array) or just some array elements to use in further analysis or other operations. In such case, you will need to subset, slice and/or index your arrays.

These operations are very similar to when you perform them on Python lists. If you want to check out the similarities for yourself, or if you want a more elaborate explanation, you might consider checking out DataCamp's [Python list tutorial](#).

If you have no clue at all on how these operations work, it suffices for now to know these two basic things:

- You use square brackets `[]` as the index operator, and
- Generally, you pass integers to these square brackets, but you can also put a colon `:` or a combination of the colon with integers in it to designate the elements/rows/columns

you want to select.

Subsetting

Besides from these two points, the easiest way to see how this all fits together is by looking at some examples of subsetting:

```
import numpy as np

# Initialize 1D array
my_array = np.array([1,2,3,4])

# Print subsets
print(my_array[1])
```

 Explain code

 OpenAI

```
import numpy as np

# Initialize 2D array
my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Print subsets
print(my_2d_array[1][2])
print(my_2d_array[1,2])
```

 Explain code

 OpenAI

```
import numpy as np

# Initialize 3D array
my_3d_array = np.array([[[1,2,3,4], [5,6,7,8]], [[1,2,3,4], [9,10,11,12]]], dtype=np.int64)

# Print subset
print(my_3d_array[1,1,2])
```

 Explain code

 OpenAI

Slicing

Something a little bit more advanced than subsetting, if you will, is slicing. Here, you consider not just particular values of your arrays, but you go to the level of rows and columns. You're basically working with "regions" of data instead of pure "locations".

You can see what is meant with this analogy in these code examples:

```
import numpy as np

# Initialize 1D array
my_array = np.array([1,2,3,4])

# Print subsets
print(my_array[0:2])
```

 Explain code

 OpenAI

```
import numpy as np

# Initialize 2D array
my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)
```

```
# Print subsets
print(my_2d_array[0:2,1])
```

 Explain code

 OpenAI

```
import numpy as np

# Initialize 3D array
my_3d_array = np.array([[[1,2,3,4], [5,6,7,8]], [[1,2,3,4], [9,10,11,12]]], dtype=

# Print subset
print(my_3d_array[1,...])
```

 Explain code

 OpenAI

You'll see that, in essence, the following holds:

```
a[start:end] # items start through the end (but the end is not included!)
a[start:]    # items start through the rest of the array
a[:end]      # items from the beginning through the end (but the end is not inclu
```

 Explain code

 OpenAI

In the above code, each array is initialized separately, and subsets are printed in separate code blocks. The `my_array`, `my_2d_array`, and `my_3d_array` are the 1D, 2D, and 3D arrays, respectively, and subsets are printed using the indexing notation.

In the first example, we use slicing to select the items at index 0 and 1 of `my_array`. In the second example, we use slicing to select the items at row 0 and 1, column 1 of `my_2d_array`. In the third example, we use the `...` notation to select all elements along the first and second dimensions, and the second element along the third dimension of `my_3d_array`.

Indexing

Lastly, there's also indexing. When it comes to NumPy, there are boolean indexing and advanced or "fancy" indexing.

(In case you're wondering, this is true NumPy jargon, I didn't make the last one up!)

First up is boolean indexing. Here, instead of selecting elements, rows or columns based on index number, you select those values from your array that fulfill a certain condition.

Putting this into code can be pretty easy:

```
import numpy as np

my_array = np.array([1,2,3,4])
my_3d_array = np.array([[[1,2,3,4], [5,6,7,8]], [[1,2,3,4], [9,10,11,12]]], dtype=

# Try out a simple example
print(my_array[my_array<2])

# Specify a condition
bigger_than_3 = (my_3d_array >= 3)

# Use the condition to index our 3d array
print(my_3d_array[bigger_than_3])
```



Note that to specify a condition, you can also make use of the logical operators `|` (OR) and `&` (AND). If you would want to rewrite the condition above in such a way (which would be inefficient, but I demonstrate it here for educational purposes :)), you would get

```
bigger_than_3 = (my_3d_array > 3) | (my_3d_array == 3).
```

With the arrays that have been loaded in, there aren't too many possibilities, but with arrays that contain, for example, names or capitals, the possibilities could be endless!

When it comes to fancy indexing, what you basically do with it is the following: you pass a list or an array of integers to specify the order of the subset of rows you want to select out of the original array.

Does this sound a little bit abstract to you?

No worries, just try it out in the code chunk below:

```
import numpy as np
my_array = np.array([1,2,3,4])
my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)
my_3d_array = np.array([[[1,2,3,4], [5,6,7,8]], [[1,2,3,4], [9,10,11,12]]], dtype=

# Select elements at (1,0), (0,1), (1,2) and (0,0)
print(my_2d_array[[1, 0, 1, 0], [0, 1, 2, 0]])

# Select a subset of the rows and columns
print(my_2d_array[[1, 0, 1, 0][:, [0,1,2,0]])
```



Now, the second statement might seem to make less sense to you at first sight. This is normal. It might make more sense if you break it down:

- If you just execute `my_2d_array[[1,0,1,0]]`, the result is the following:

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



- What the second part, namely, `[:, [0,1,2,0]]`, is tell you that you want to keep all the rows of this result, but that you want to change the order of the columns around a bit. You want to display the columns 0, 1, and 2 as they are right now, but you want to repeat column 0 as the last column instead of displaying column number 3. This will give you the following result:

```
array([[5, 6, 7, 5],
       [1, 2, 3, 1],
       [5, 6, 7, 5],
       [1, 2, 3, 1]])
```



Advanced indexing clearly holds no secrets for you any more!

How to Ask for Help

As a short intermezzo, you should know that you can always ask for more information about the modules, functions or classes that you're working with, especially because NumPy can be quite something when you first get started on working with it.

Asking for help is fairly easy.

You just make use of the specific help functions that numpy offers to set you on your way:

- Use `lookfor()` to do a keyword search on docstrings. This is specifically handy if you're just starting out, as the 'theory' behind it all might fade in your memory. The one downside is that you have to go through all of the search results if your query is not that specific, as is the case in the code example below. This might make it even less overviewable for you.
- Use `info()` for quick explanations and code examples of functions, classes, or modules. If you're a person that learns by doing, this is the way to go! The only downside about using this function is probably that you need to be aware of the module in which certain attributes or functions are in. If you don't know immediately what is meant by that, check out the code example below.

You see, both functions have their advantages and disadvantages, but you'll see for yourself why both of them can be useful: try them out for yourself in the code chunk below!

```
import numpy as np

# Look up info on `mean` with `np.lookfor()`
print(np.lookfor("mean"))
```



Explain code

OpenAI

```
# Get info on data types with `np.info()`
np.info(np.ndarray.dtype)
```



Explain code

OpenAI

Note that you indeed need to know that `dtype` is an attribute of `ndarray`. Also, make sure that you don't forget to put `np` in front of the modules, classes or terms you're asking information about, otherwise you will get an error message like this:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ndarray' is not defined
```

You now know how to ask for help, and that's a good thing. The next topic that this NumPy tutorial covers is array manipulation.

Not that you can not overcome this topic on your own, quite the contrary!

But some of the functions might raise questions, because, what is the difference between resizing and reshaping?

And what is the difference between stacking your arrays horizontally and vertically?

The next section is all about answering these questions, but if you ever feel in doubt, feel free to use the help functions that you have just seen to quickly get up to speed.

How to Manipulate Arrays

Performing mathematical operations on your arrays is one of the things that you'll be doing, but probably most importantly to make this and the broadcasting work is to know how to manipulate your arrays.

Below are some of the most common manipulations that you'll be doing.

How to Transpose your Arrays

What transposing your arrays actually does is permuting the dimensions of it. Or, in other words, you switch around the shape of the array. Let's take a small example to show you the effect of transposition:

```
import numpy as np
my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Print `my_2d_array`
print(my_2d_array)

# Transpose `my_2d_array`
print(np.transpose(my_2d_array))

# Or use `T` to transpose `my_2d_array`
print(my_2d_array.T)
```

[Explain code](#)

Tip: if the visual comparison between the array and its transposed version is not entirely clear, inspect the shape of the two arrays to make sure that you understand why the dimensions are permuted.

Note that there are two transpose functions. Both do the same; There isn't too much difference. You do have to take into account that `T` seems more of a convenience function and that you have a lot more flexibility with `np.transpose()`. That's why it's recommended to make use of this function if you want to more arguments.

All is well when you transpose arrays that are bigger than one dimension, but what happens when you just have a 1-D array? Will there be any effect, you think?

Try it out for yourself in the code chunk below. Your 1-D array has already been loaded in:

```
import numpy as np
my_array = np.array([1,2,3,4])

# Print `my_2d_array`
print(my_array)

# Transpose `my_2d_array`
print(np.transpose(my_array))

# Or use `T` to transpose `my_2d_array`
print(my_array.T)
```

[Explain code](#)

You're absolutely right! There is no effect when you transpose a 1-D array!

Reshaping Versus Resizing your Arrays

You might have read in the broadcasting section that the dimensions of your arrays need to be compatible if you want them to be good candidates for arithmetic operations. But the question of what you should do when that is not the case, was not answered yet.

Well, this is where you get the answer!

What you can do if the arrays don't have the same dimensions, is resize your array. You will then return a new array that has the shape that you passed to the `np.resize()` function. If you pass your original array together with the new dimensions, and if that new array is larger than the one that you originally had, the new array will be filled with copies of the original array that are repeated as many times as is needed.

However, if you just apply `np.resize()` to the array and you pass the new shape to it, the new array will be filled with zeros.

Let's try this out with an example:

```
import numpy as np
x = np.ones((3,4))

# Print the shape of `x`
print(x.shape)

# Resize `x` to ((6,4))
np.resize(x, (6,4))

# Try out this as well
x.resize((6,4))

# Print out `x`
print(x)
```



Besides resizing, you can also reshape your array. This means that you give a new shape to an array without changing its data. The key to reshaping is to make sure that the total size of the new array is unchanged. If you take the example of array `x` that was used above, which has a size of 3 X 4 or 12, you have to make sure that the new array also has a size of 12.

Psst... If you want to calculate the size of an array with code, make sure to use the size attribute: `x.size` or `x.reshape((2,6)).size`:

```
import numpy as np

# Initialize array with shape (3,4) containing all ones
x = np.ones((3,4))

# Print the shape of `x`
print("Shape of x:", x.shape)

# Resize `x` to ((6,4))
np.resize(x, (6,4))

# Try out this as well
x.resize((6,4))

# Print out `x` before and after resizing
print("Array before transposing:\n", x)
print("Shape of array before transposing:", x.shape)

# Transpose `x`
x = x.T

# Print out `x` after transposing
print("Array after transposing:\n", x)
print("Shape of array after transposing:", x.shape)
```



Output:

```
Shape of x: (3, 4)
Array before transposing:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
```



```
[1. 1. 1. 1.]
[1. 1. 1. 1.]
[1. 1. 1. 1.]]
Shape of array before transposing: (6, 4)
Array after transposing:
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
Shape of array after transposing: (4, 6)
```

[🔗 Explain code](#)
 OpenAI

If all else fails, you can also append an array to your original one or insert or delete array elements to make sure that your dimensions fit with the other array that you want to use for your computations.

Another operation that you might keep handy when you're changing the shape of arrays is `ravel()`. This function allows you to flatten your arrays. This means that if you ever have 2D, 3D or n-D arrays, you can just use this function to flatten it all out to a 1-D array.

Pretty handy, isn't it?

How to Append Arrays

When you append arrays to your original array, they are "glued" to the end of that original array. If you want to make sure that what you append does not come at the end of the array, you might consider inserting it. Go to the next section if you want to know more.

Appending is a pretty easy thing to do thanks to the NumPy library; You can just make use of the `np.append()`.

Check how it's done in the code chunk below. Don't forget that you can always check which arrays are loaded in by typing, for example, `my_array` in the IPython shell and pressing ENTER.

```
import numpy as np

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

# Print `my_array` before appending
print("my_array before appending:", my_array)

# Append a 1D array to `my_array`
new_array = np.append(my_array, [7, 8, 9, 10])

# Print `new_array`
print("new_array:", new_array)

# Print `my_array` after appending
print("my_array after appending:", my_array)
```


[🔗 Explain code](#)
 OpenAI

Output:

```
my_array before appending: [1 2 3 4]
new_array: [ 1  2  3  4  7  8  9 10]
my_array after appending: [1 2 3 4]
```


[🔗 Explain code](#)
 OpenAI

Note that `my_array` remains unchanged after appending because `np.append()` returns a new array rather than modifying the original array in place.

Appending to my_2d_array:

```
import numpy as np

my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Print `my_2d_array` before appending
print("my_2d_array before appending:\n", my_2d_array)

# Append an extra column to `my_2d_array`
new_2d_array = np.append(my_2d_array, [[7], [8]], axis=1)

# Print `new_2d_array`
print("new_2d_array:\n", new_2d_array)

# Print `my_2d_array` after appending
print("my_2d_array after appending:\n", my_2d_array)
```

 Explain code

 OpenAI

Output:

```
my_2d_array before appending:
[[1 2 3 4]
 [5 6 7 8]]
new_2d_array:
[[1 2 3 4 7 8]]
```

[5 6 7 8]]

 Explain code

 OpenAI

Note how, when you append an extra column to my_2d_array, the axis is specified. Remember that axis 1 indicates the columns, while axis 0 indicates the rows in 2-D arrays.

How to Insert and Delete Array Elements

Next to appending, you can also insert and delete array elements. As you might have guessed by now, the functions that will allow you to do these operations are np.insert() and np.delete():

```
import numpy as np
my_array = np.array([1,2,3,4])
my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Insert `5` at index 1
np.insert(my_array, 1, 5)

# Delete the value at index 1
np.delete(my_array, [1])
```

 Explain code

 OpenAI

How to Join and Split Arrays

You can also 'merge' or join your arrays. There are a bunch of functions that you can use for that purpose and most of them are listed below.

Try them out, but also make sure to test out what the shape of the arrays is in the IPython shell. The arrays that have been loaded are x, my_array, my_resized_array and my_2d_array.

```
import numpy as np
x = np.ones((4,))
my_array = np.array([1,2,3,4])
my_resized_array=np.resize(my_array,(2,4))
my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)

# Concatentate `my_array` and `x`
print(np.concatenate((my_array,x)))

# Stack arrays row-wise
print(np.vstack((my_array, my_2d_array)))

# Stack arrays row-wise
print(np.r_[my_resized_array, my_2d_array])

# Stack arrays horizontally
print(np.hstack((my_resized_array, my_2d_array)))

# Stack arrays column-wise
print(np.column_stack((my_resized_array, my_2d_array)))

# Stack arrays column-wise
print(np.c_[my_resized_array, my_2d_array])
```



Explain code



You'll note a few things as you go through the functions:

- The number of dimensions needs to be the same if you want to concatenate two arrays with `np.concatenate()`. As such, if you want to concatenate an array with `my_array`, which is 1-D, you'll need to make sure that the second array that you have, is also 1-D.
- With `np.vstack()`, you effortlessly combine `my_array` with `my_2d_array`. You just have to make sure that, as you're stacking the arrays row-wise, that the number of columns in both arrays is the same. As such, you could also add an array with shape (2,4) or (3,4) to `my_2d_array`, as long as the number of columns matches. Stated differently, the arrays must have the same shape along all but the first axis. The same holds also for when you want to use `np.r_[]`.
- For `np.hstack()`, you have to make sure that the number of dimensions is the same and that the number of rows in both arrays is the same. That means that you could stack arrays such as (2,3) or (2,4) to `my_2d_array`, which itself as a shape of (2,4). Anything is possible as long as you make sure that the number of rows matches. This function is still supported by NumPy, but you should prefer `np.concatenate()` or `np.stack()`.
- With `np.column_stack()`, you have to make sure that the arrays that you input have the same first dimension. In this case, both shapes are the same, but if `my_resized_array` were to be (2,1) or (2,), the arrays still would have been stacked.
- `np.c_[]` is another way to concatenate. Here also, the first dimension of both arrays needs to match.

When you have joined arrays, you might also want to split them at some point. Just like you can stack them horizontally, you can also do the same but then vertically. You use `np.hsplit()` and `np.vsplit()`, respectively:

```
import numpy as np
my_array = np.array([1,2,3,4])
my_resized_array=np.resize(my_array,(2,4))
my_2d_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64)
my_stacked_array = np.hstack((my_resized_array, my_2d_array))

# Split `my_stacked_array` horizontally at the 2nd index
print(np.hsplit(my_stacked_array, 2))
```



```
# Split `my_stacked_array` vertically at the 2nd index
print(np.vsplit(my_stacked_array, 2))
```

 Explain code

 OpenAI

What you need to keep in mind when you're using both of these split functions is probably the shape of your array. Let's take the above case as an example: `my_stacked_array` has a shape of `(2,8)`. If you want to select the index at which you want the split to occur, you have to keep the shape in mind.

How to Visualize NumPy Arrays

Lastly, something that will definitely come in handy is to know how you can plot your arrays. This can especially be handy in data exploration, but also in later stages of the data science workflow, when you want to visualize your arrays.

With `np.histogram()`

Contrary to what the function might suggest, the `np.histogram()` function doesn't draw the histogram but it does compute the occurrences of the array that fall within each bin; This will determine the area that each bar of your histogram takes up.

What you pass to the `np.histogram()` function then is first the input data or the array that you're working with. The array will be flattened when the histogram is computed.

```
# Import `numpy` as `np`
import numpy as np

# Initialize your array
my_3d_array = np.array([[[1,2,3,4], [5,6,7,8]], [[1,2,3,4], [9,10,11,12]]], dtype

# Pass the array to `np.histogram()`
print(np.histogram(my_3d_array))

# Specify the number of bins
print(np.histogram(my_3d_array, bins=range(0,13)))
```


 Explain code

 OpenAI

You'll see that as a result, the histogram will be computed: the first array lists the frequencies for all the elements of your array, while the second array lists the bins that would be used if you don't specify any bins.

If you do specify a number of bins, the result of the computation will be different: the floats will be gone and you'll see all integers for the bins.

There are still some other arguments that you can specify that can influence the histogram that is computed. You can find all of them [here](#).

But what is the point of computing such a histogram if you can't visualize it?

Visualization is a piece of cake with the help of Matplotlib, but you don't need `np.histogram()` to compute the histogram. `plt.hist()` does this for itself when you pass it the (flattened) data and the bins:

```
# Import numpy and matplotlib
import numpy as np
import matplotlib.pyplot as plt

# Construct the histogram with a flattened 3d array and a range of bins
plt.hist(my_3d_array.ravel(), bins=range(0,13))

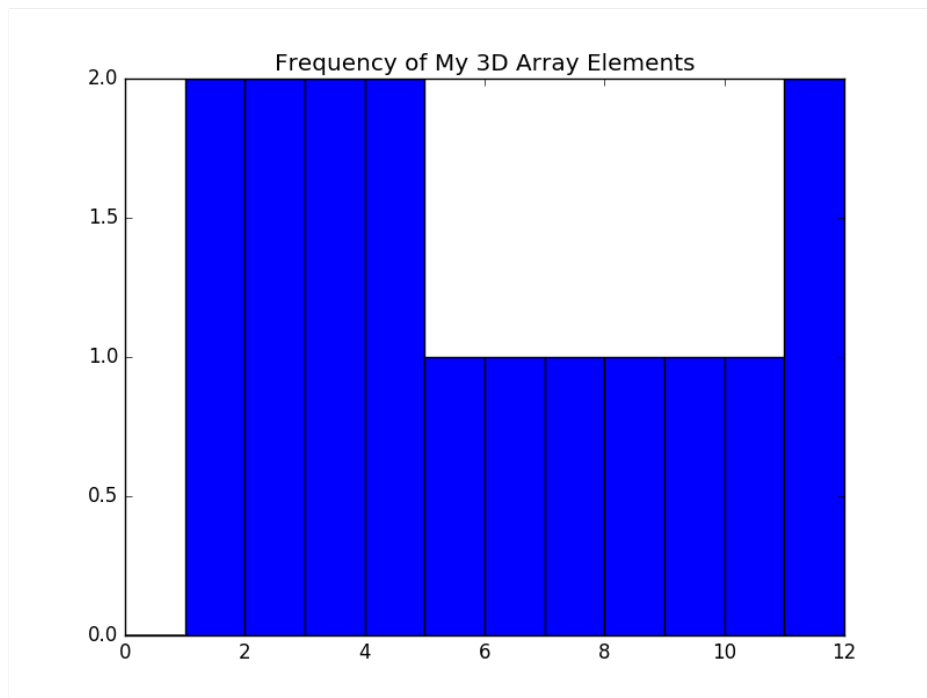
# Add a title to the plot
plt.title('Frequency of My 3D Array Elements')
```



```
# Show the plot
plt.show()
```

[🔗 Explain code](#)
[OpenAI](#)

The above code will then give you the following (basic) histogram:



Using np.meshgrid()

Another way to (indirectly) visualize your array is by using `np.meshgrid()`. The problem that you face with arrays is that you need 2-D arrays of x and y coordinate values. With the above function, you can create a rectangular grid out of an array of x values and an array of y values: the `np.meshgrid()` function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of (x, y) in the two arrays. Then, you can use these matrices to make all sorts of plots.

`np.meshgrid()` is particularly useful if you want to evaluate functions on a grid, as the code below demonstrates:

```
# Import NumPy and Matplotlib
import numpy as np
import matplotlib.pyplot as plt

# Create an array
points = np.arange(-5, 5, 0.01)

# Make a meshgrid
xs, ys = np.meshgrid(points, points)
z = np.sqrt(xs ** 2 + ys ** 2)

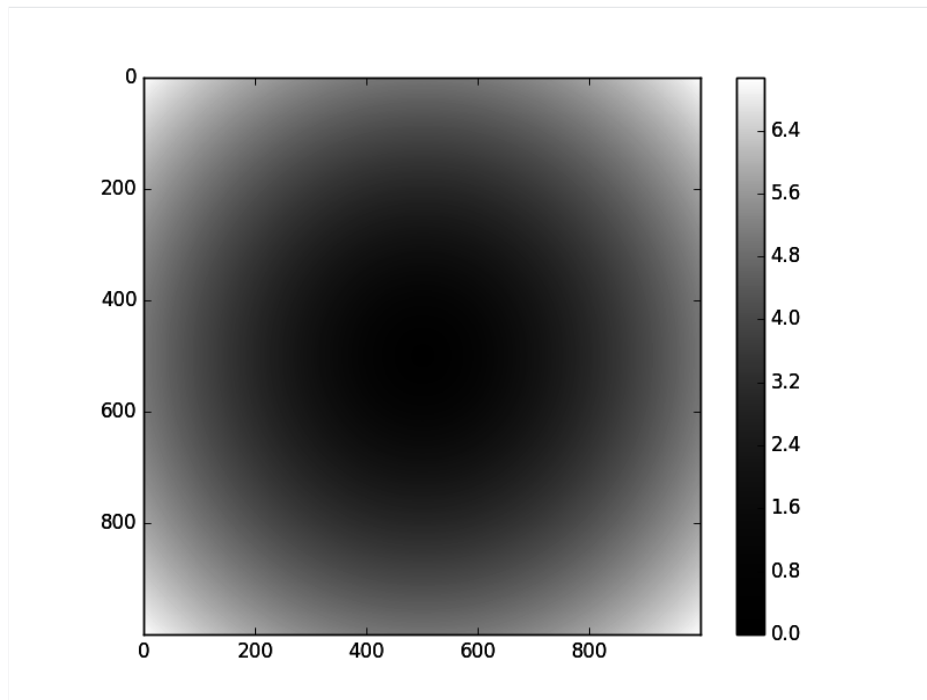
# Display the image on the axes
plt.imshow(z, cmap=plt.cm.gray)

# Draw a color bar
plt.colorbar()

# Show the plot
plt.show()
```

[🔗 Explain code](#)
[OpenAI](#)

The code above gives the following result:



Beyond Data Analysis with NumPy

Congratulations, you have reached the end of the NumPy tutorial!

You have covered a lot of ground, so now you have to make sure to retain the knowledge that you have gained. Don't forget to get your copy of DataCamp's NumPy cheat sheet to support you in doing this!

After all this theory, it's also time to get some more practice with the concepts and techniques that you have learned in this tutorial. One way to do this is to go back to the [scikit-learn tutorial](#) and start experimenting with further with the data arrays that are used to build machine learning models.

If this is not your cup of tea, check again whether you have downloaded Anaconda. Then, get started with NumPy arrays in Jupyter with this [Definitive Guide to Jupyter Notebook](#). Also make sure to check out [this Jupyter Notebook](#), which also guides you through data analysis in Python with NumPy and some other libraries in the interactive data science environment of the Jupyter Notebook.

Lastly, consider checking out DataCamp's courses on data manipulation and visualization. Especially our latest courses in collaboration with Continuum Analytics will definitely interest you! Take a look at the [Manipulating DataFrames with Pandas](#) or the [Pandas Foundations](#) courses.

TOPICS

[Python](#) [Data Science](#) [Data Analysis](#)

Learn more about Python

Introduction to NumPy

Beginner ⌚ 4 hr 👁 25.6K

Master your skills in NumPy by learning how to create, sort, filter, and update arrays using NYC's tree census.

[See Details →](#)[Start Course](#)

S

[See More →](#)

Related

An Introduction to Polars:
Python's Tool for Large-Scale...

Moez Ali



An Introduction to DuckDB:
What is It and Why Should You...

Kurtis Pykes

How Data Science is
Transforming the NBA

Richie Cotton

[See More →](#)

Grow your data skills with DataCamp for Mobile

Make progress on the go with our mobile courses and daily 5-minute coding challenges.



LEARN

[Learn Python](#)[Learn R](#)[Learn AI](#)[Learn SQL](#)[Learn Power BI](#)[Learn Tableau](#)[Assessments](#)[Career Tracks](#)[Skill Tracks](#)[Courses](#)[Data Science Roadmap](#)

DATA COURSES

[Upcoming Courses](#)

[Python Courses](#)

[R Courses](#)

[SQL Courses](#)

[Power BI Courses](#)

[Tableau Courses](#)

[Spreadsheets Courses](#)

[Data Analysis Courses](#)

[Data Visualization Courses](#)

[Machine Learning Courses](#)

[Data Engineering Courses](#)

WORKSPACE

[Get Started](#)

[Templates](#)

[Integrations](#)

[Documentation](#)

CERTIFICATION

[Certifications](#)

[Data Scientist](#)

[Data Analyst](#)

[Data Engineer](#)

[Hire Data Professionals](#)

RESOURCES

[Resource Center](#)

[Upcoming Events](#)

[Blog](#)

[Tutorials](#)

[Open Source](#)

[RDocumentation](#)

[Course Editor](#)

[Book a Demo with DataCamp for Business](#)

[Data Portfolio](#)

[Portfolio Leaderboard](#)

PLANS

[Pricing](#)

[For Business](#)

[For Universities](#)

[Discounts, Promos & Sales](#)

[DataCamp Donates](#)

SUPPORT

[Help Center](#)

[Become an Instructor](#)

[Become an Affiliate](#)

ABOUT

[About Us](#)

[Learner Stories](#)

[Careers](#)

[Press](#)

[Leadership](#)

[Contact Us](#)



[Privacy Policy](#)

[Cookie Notice](#)

[Do Not Sell My Personal Information](#)

[Accessibility](#)

[Security](#)

[Terms of Use](#)

© 2023 DataCamp, Inc. All Rights Reserved.