
Introduction to NumPy

This chapter, along with [Chapter 3](#), outlines techniques for effectively loading, storing, and manipulating in-memory data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats, including collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, images—particularly digital images—can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data are, the first step in making them analyzable will be to transform them into arrays of numbers. (We will discuss some specific examples of this process later in [“Feature Engineering” on page 375](#).)

For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. We’ll now take a look at the specialized tools that Python has for handling such numerical arrays: the NumPy package and the Pandas package (discussed in [Chapter 3](#).)

This chapter will cover NumPy in detail. NumPy (short for *Numerical Python*) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python’s built-in `list` type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice outlined in the preface and installed the Anaconda stack, you already have NumPy installed and ready to go. If you're more the do-it-yourself type, you can go to [the NumPy website](#) and follow the installation instructions found there. Once you do, you can import NumPy and double-check the version:

```
In[1]: import numpy
       numpy.__version__
```

```
Out[1]: '1.11.1'
```

For the pieces of the package discussed here, I'd recommend NumPy version 1.8 or later. By convention, you'll find that most people in the SciPy/PyData world will import NumPy using `np` as an alias:

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

Throughout this chapter, and indeed the rest of the book, you'll find that this is the way we will import and use NumPy.

Reminder About Built-In Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature) as well as the documentation of various functions (using the `?` character). Refer back to ["Help and Documentation in IPython" on page 3](#) if you need a refresher on this.

For example, to display all the contents of the `numpy` namespace, you can type this:

```
In [3]: np.<TAB>
```

And to display NumPy's built-in documentation, you can use this:

```
In [4]: np?
```

More detailed documentation, along with tutorials and other resources, can be found at <http://www.numpy.org>.

Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This section outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn in by its ease of use, one piece of which is dynamic typing. While a statically typed language like C or Java requires each variable to be