**Numpy**

1. This chapter along with the chapter 3(Pandas) outlines techniques for effectively storing loading and manuplating 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 numerical measurement, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of number
2. For example, images-particulary digital images can be thought of as simply two-dimensional arrays of numbers representing pixel brightness across area.

🡪Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical represntations, 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**

1. For this reason, efficient storage and manipulation of numerical of numerical arrays is absolutely fundamental to the process of doing data science. We will now take a look at the specialized tools that python has for handling such numerical arrays: the numpy package and the pandas package
2. 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 lists type, but numpy arrays provides much more efficient storage and data operations as the arrays grow larger in size.

🡪Numpy arrays from the core of nearly the entire ecosystem of data science tools in python, so time spent learning to use Numpy effectively will be valueable no matter what aspect of data science interests you

1. By installing the anaconda stack, it will help us such that all the required packages for the machine learning will be installed automatically and than we have to only import them in our projects, and not to be in the stuff of installing different packages
2. And than by installing the anaconda the numpy package will be installed automatically and than we only remained to use that, And for the usage of the package we have to write the following statement as shown below, We can also import it by different name instead of the np, but in most of the projects it is used as the np only, so it is recommended to use the np



1. 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

1. A python integer is more than just a integer

🡪The standard python implementations is written in the c language. This means that every python object is simply a cleverly disguised c structure, which contains not only its value, but other information as well.

🡪For example, when we define an integer in python, such as the x=10000, x is not just a “raw” integer. It’s actually a pointer to a compound C structure, which contains several values.

🡪A single integer in python 3.4 actually contains four piece:

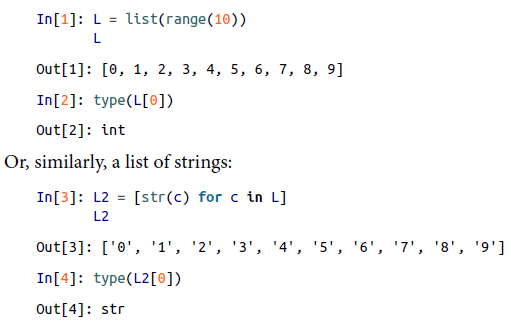
1. ob\_refcnt , a reference count that helps python silently handle memory allocation and deallocation
2. ob\_type , which encodes the type of the variable
3. ob\_size , which specifies the size of the following data members
4. ob\_digit , which contains the actual integer value that we except the python variable to represent

🡪­Notice the difference here: a C integer is essentially a label for a position in memory whose bytes encode an integer value. A python integer is a pointer to a position in memory containing all the python object information, including the bytes that contain the integer value

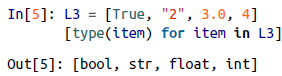
🡪This extra information in the python integer structure is what allows python to be coded so freely and dynamically. All this additional information in python types comes at a cost, however, which becomes especially apparent in structure that combine many of these objects

1. A python is more than just a list

🡪Lets consider now what happens when we use a python data structure that holds many python objects. The standard mutable multielement container in python is the list. We can create a list of integers as follows:



🡪Because of python’s dynamic typing, we can even create heterogenous lists:

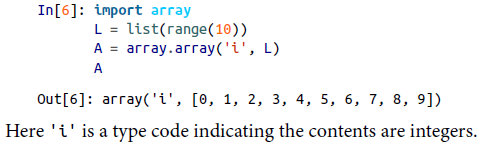


🡪But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information-that is, each item is a complete python object.

🡪In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed type array

🡪The advantage of the list is flexibility: because each list element is a full structure containing both data and type of information, the list can be filled with the data of any desired type. Fixed type Numpy-style arrays lack this flexibility, but are much more efficient for sorting and manipulating data

1. Fixed-Type arrays in python:
   1. Python offers several different options for storing data in efficient, fixed-type data buffers. The built-in array module can be used to create dense arrays of a uniform type:

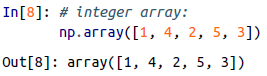


* 1. While python’s array object provides efficient storage of array-based data, Numpy adds to this efficient operations on that data. We will explore these operations in later sections; here we will demonstrate several ways of creating a Numpy array

🡪We will start with the standard Numpy import, under the alias np:

1. Creating Array from python lists:

🡪First, we can use np.array to create arrays from Python lists:



🡪Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are upcast to floating point):

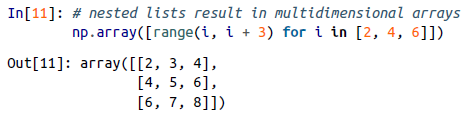


🡪If we want to explicitly set the data type of the resulting array, we can use the dtype keyword:



🡪Finally, unlike Python lists, NumPy arrays can explicitly be multidimensional; here’s

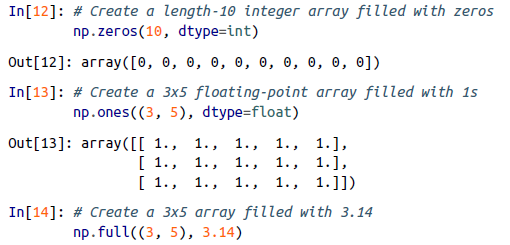
one way of initializing a multidimensional array using a list of lists:

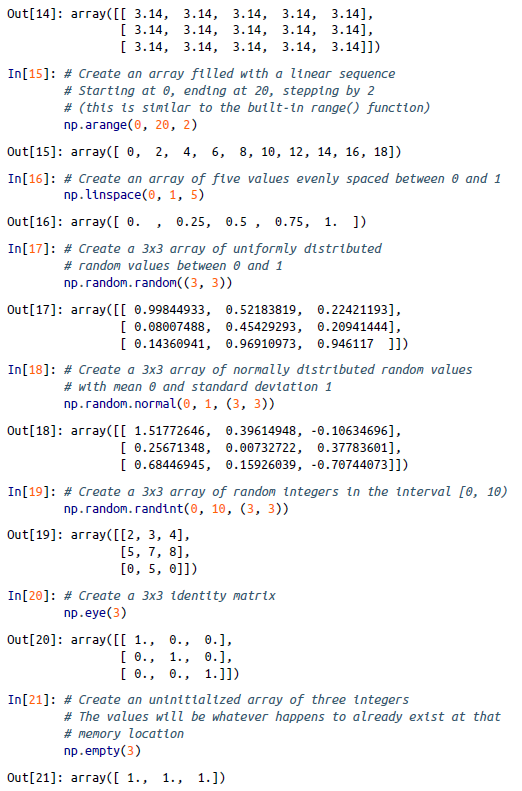


🡪The inner lists are treated as rows of the resulting two-dimensional array.

1. Creating Arrays from scratch:

🡪Especially for larger arrays, it is more efficient to create arrays from scratch using routines built into NumPy. Here are several example:





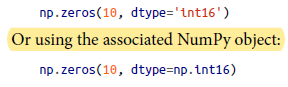
1. Numpy Standard data types:

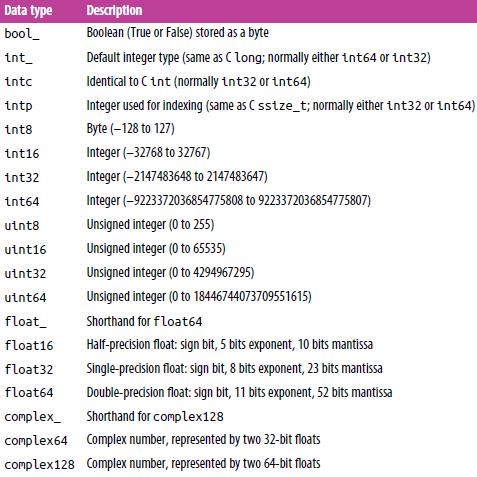
🡪NumPy arrays contain values of a single type, so it is important to have detailed

knowledge of those types and their limitations. Because NumPy is built in C, the

types will be familiar to users of C, Fortran, and other related languages.

🡪The standard Numpy data types are listed in table, Note that when constructing an array, you can specify them using a string:





🡪More advanced type is possible, such as specifying big or little endian numbers numpy also supports compound data types

1. The basics of numpy Arrays

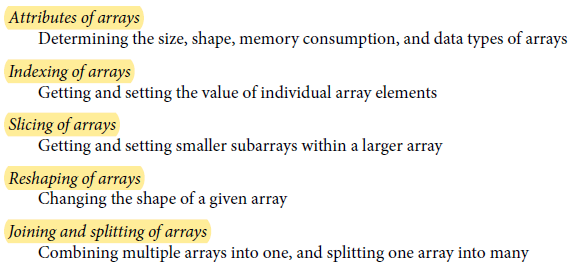
🡪Data manipulation in Python is nearly synonymous with NumPy array manipulation:

even newer tools like Pandas (Chapter 3) are built around the NumPy array.

🡪This section will present several examples using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays.

🡪While the types of operations shown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the book.

🡪We will cover a few categories of basic array manipulations here:

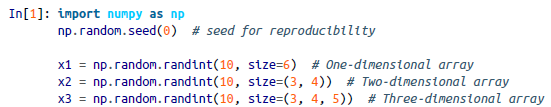


1. Numpy array attributes

🡪First let’s discuss some useful array attributes. We’ll start by defining three random

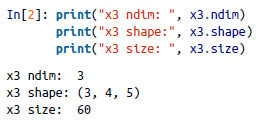
arrays: a one-dimensional, two-dimensional, and three-dimensional array.

🡪We’ll use NumPy’s random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:



🡪Each array has attributes ndim (the number of dimensions), shape (the size of each

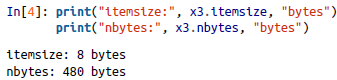
dimension), and size (the total size of the array):



🡪Another useful attribute is the dtype, the data type of the array,



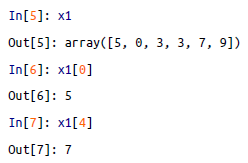
🡪Other attributes include itemsize, which lists the size (in bytes) of each array element, and nbytes, which lists the total size (in bytes) of the array:

****

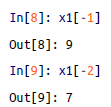
1. **Array Indexing: Accessing Single Elements**

**🡪**If you are familiar with Python’s standard list indexing, indexing in NumPy will feel

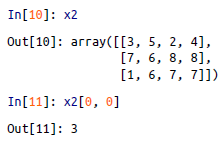
quite familiar. In a one-dimensional array, you can access the *i*th value (counting from zero) by specifying the desired index in square brackets, just as with Python lists:



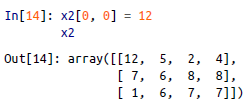
🡪To index from the end of the array, you can use negative indeices:



🡪In a multidimensional array, you access items using a comma-seprated tuple of indices:



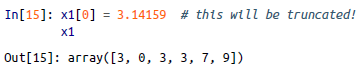
🡪You can also modify values using any of the above index notation:



🡪Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means,

for example, that if you attempt to insert a floating-point value to an integer array, the

value will be silently truncated.



1. Array Slicing: Accessing Subarrays

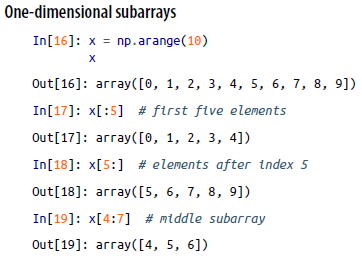
🡪Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character.

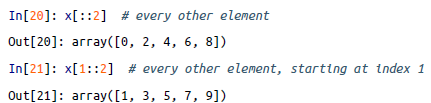
🡪The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:



🡪If any of these are unspecified, they default to the values start=0, stop=*size of*

*dimension*, step=1.We will take look at accessing subarrays in one dimension and in multiple dimensions

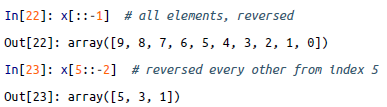


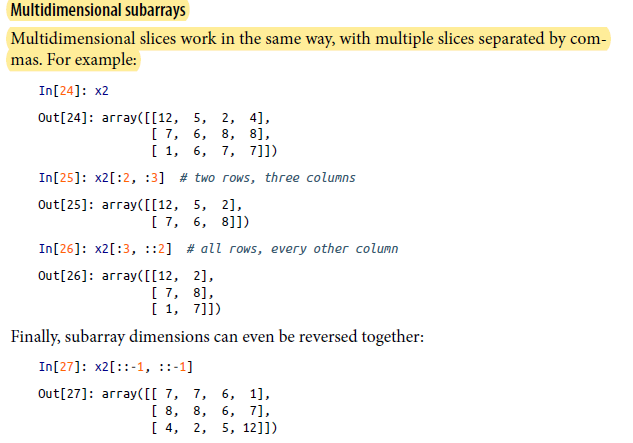


*🡪*A potentially confusing case is when the step value is negative. In this case, the

defaults for start and stop are swapped. This becomes a convenient way to reverse

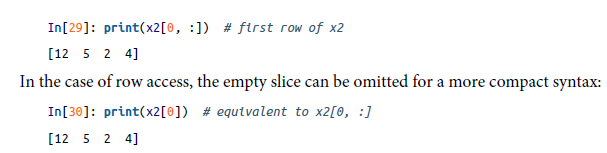
an array:





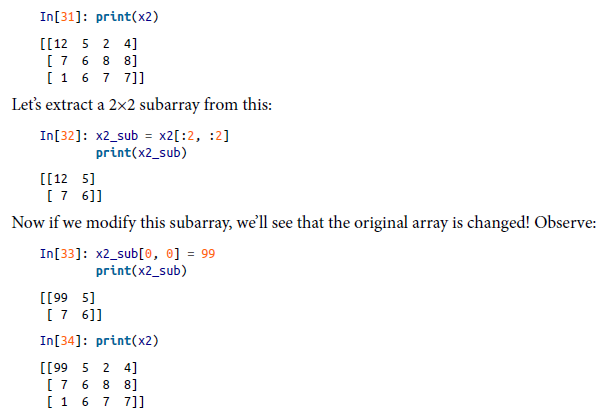
1. Accessing array row and columns: One commonly needed routine is accessing single row or columns of an array. You can do this by combining indexing and slicing, using an empty slice marked by a single colon (:):





1. Subarrays as no-copy views(I.M.P.)

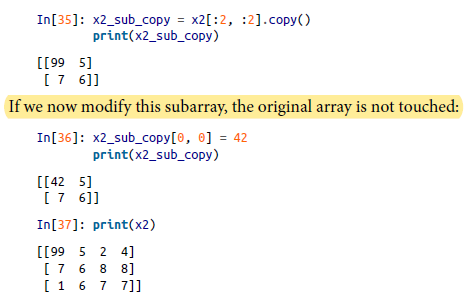
🡪One important—and extremely useful—thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:



🡪This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

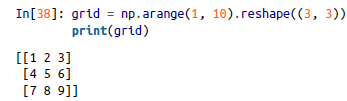
1. Creating copies of arrays(I.M.P.)

🡪Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the copy() method:



1. Reshaping of Arrays

🡪Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the reshape() method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

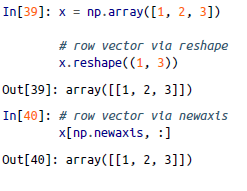


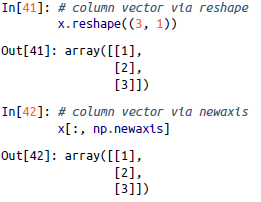
🡪Note that for this to work, the size of the initial array must match the size of the

reshaped array. Where possible, the reshape method will use a no-copy view of the

initial array, but with noncontiguous memory buffers this is not always the case.

🡪Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. You can do this with the reshape method, or more easily by making use of the newaxis keyword within a slice operation:





1. Array Concatenation and Splitting

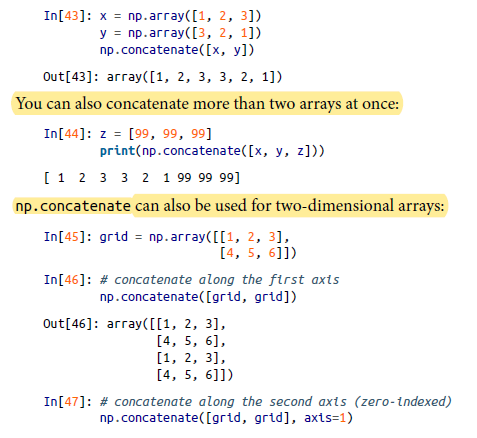
🡪All of the preceding routines worked on single arrays. It’s also possible to combine

multiple arrays into one, and to conversely split a single array into multiple arrays.

We’ll take a look at those operations here.

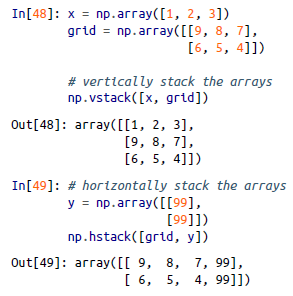
1. Concatenation Of Arrays:

🡪Concatenation, or joining of two arrays in NumPy, is primarily zaccomplished through the routines np.concatenate, np.vstack, and np.hstack. np.concatenate takes a tuple or list of arrays as its first argument, as we can see here:





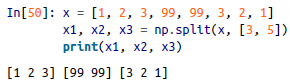
🡪For working with arrays of mixed dimensions, it can be clearer to use the np.vstack(vertical stack) and np.hstack (horizontal stack) functions:



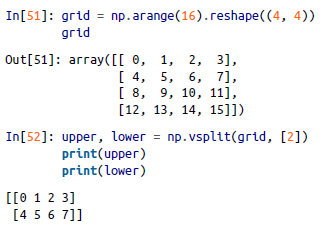
🡪Similarly, np.dstack will stack arrays along the third axis

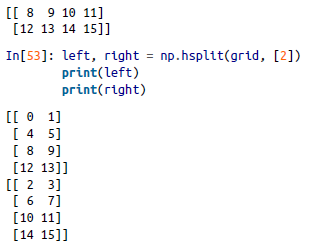
1. Splitting Of Arrays:

🡪The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points:



🡪Notice that *N* split points lead to *N + 1* subarrays. The related functions np.hsplit and np.vsplit are similar:





🡪Similarly, np.dsplit will split arrays along the third axis.

1. **Computation on Numpy Arrays: Universal Functions**

**🡪Numpy** provides an easy and flexible interface to optimized computation with arrays of data.

**🡪**Computation on NumPy arrays can be very fast, or it can be very slow.

🡪The key to making it fast is to use *vectorized* operations, generally implemented through NumPy’s *universal functions* (ufuncs). This section motivates the need for NumPy’s ufuncs, which can be used to make repeated calculations on array elements much more efficient.

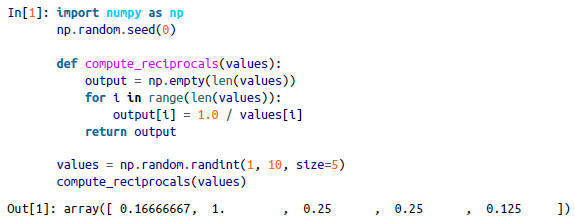
🡪It then introduces many of the most common and useful arithmetic ufuncs

available in the NumPy package.

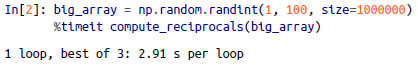
1. The Slowness of Loops

🡪Python’s default implementation (known as CPython) does some operations very slowly. This is in part due to the dynamic, interpreted nature of the language: the fact that types are flexible, so that sequences of operations cannot be compiled down to efficient machine code as in languages like C and Fortran.

🡪The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated—for instance, looping over arrays to operate on each element. For example, imagine we have an array of values and we’d like to compute the reciprocal of each. A straightforward approach might look like this:



🡪This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow, perhaps surprisingly so! We’ll benchmark this with IPython’s %timeit magic

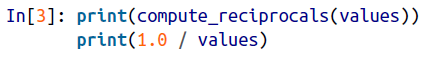


🡪It takes several seconds to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e., billions of numerical operations per second), this seems almost absurdly slow. It turns out that the bottleneck here is not the operations themselves, but the type-checking and function dispatches that CPython must do at each cycle of the loop. Each time the reciprocal is computed, Python first examines the object’s type and does a dynamic lookup of the correct function to use for that type. If we were working in compiled code instead, this type specification would be known before the code executes and the result could be computed much more efficiently.

1. Introducing UFuncs

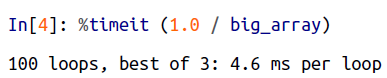
🡪 For many types of operations, NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a *vectorized* operation. You can accomplish this by simply performing an operation on the array, which will then be applied to each element. This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

🡪Compare the results of following two:

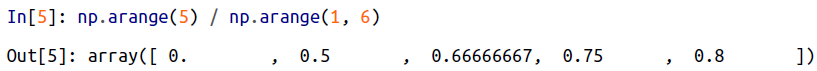




🡪 Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

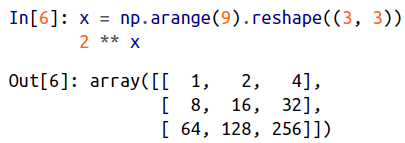


🡪 Vectorized operations in NumPy are implemented via *ufuncs*, whose main purpose is to quickly execute repeated operations on values in NumPy arrays. Ufuncs are extremely flexible—before we saw an operation between a scalar and an array, but we can also operate between two arrays:



🡪 And ufunc operations are not limited to one-dimensional arrays—they can act on

multidimensional arrays as well:

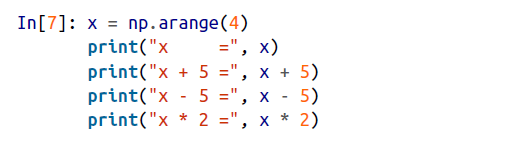


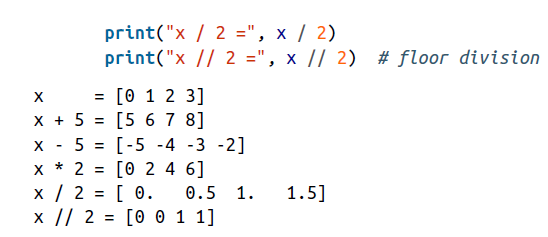
🡪 Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented through Python loops, especially as the arrays grow in size. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

1. Exploring NumPy’s UFuncs

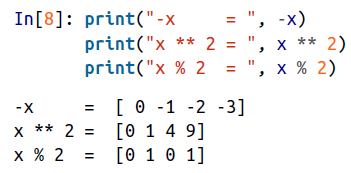
🡪 Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs.

*🡺* Array arithmetic: NumPy’s ufuncs feel very natural to use because they make use of Python’s native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

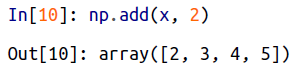




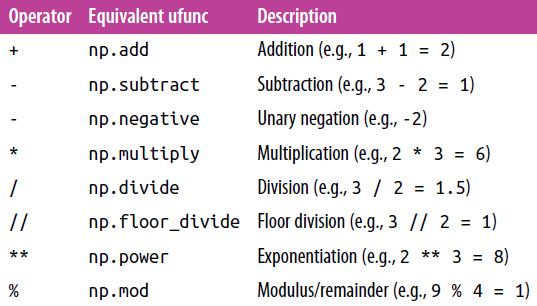
🡪 There is also a unary ufunc for negation, a \*\* operator for exponentiation, and a % operator for modulus:



🡪 All of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the + operator is a wrapper for the add function:



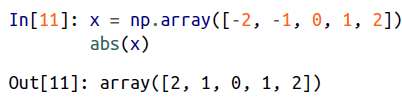
🡪List of arithmetic operators implemented in Numpy



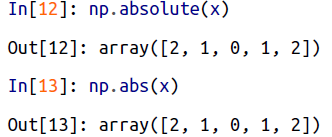
🡪 Additionally there are Boolean/bitwise operators

🡺Absolute Value

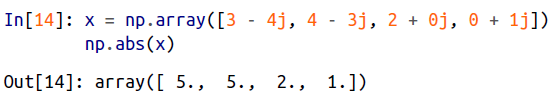
🡪 Just as NumPy understands Python’s built-in arithmetic operators, it also understands Python’s built-in absolute value function:



🡪 The corresponding NumPy ufunc is np.absolute, which is also available under the alias np.abs:



🡪 This ufunc can also handle complex data, in which the absolute value returns the magnitude:

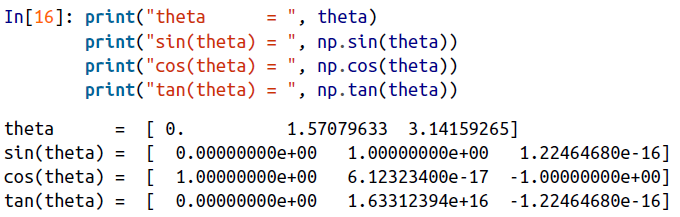


🡺Trigonometric function

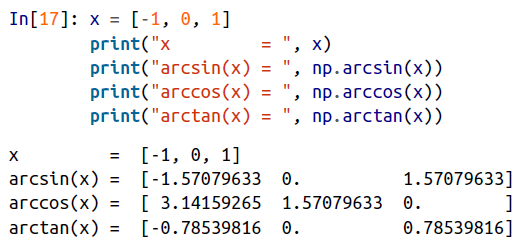
🡪 NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We’ll start by defining an array of angles:



🡪 Now we can compute some trigonometric functions on these values:

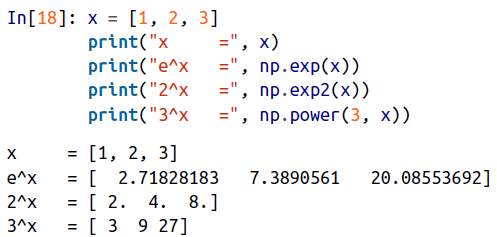


🡪 The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

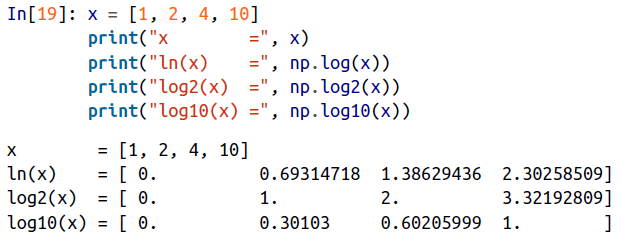


🡺Exponents And Logarithms

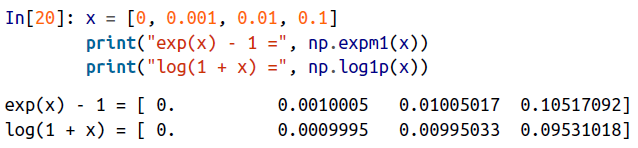
🡪 Another common type of operation available in a NumPy ufunc are the exponentials:



🡪 The inverse of the exponentials, the logarithms, are also available. The basic np.log gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:



🡪 There are also some specialized versions that are useful for maintaining precision with very small input:



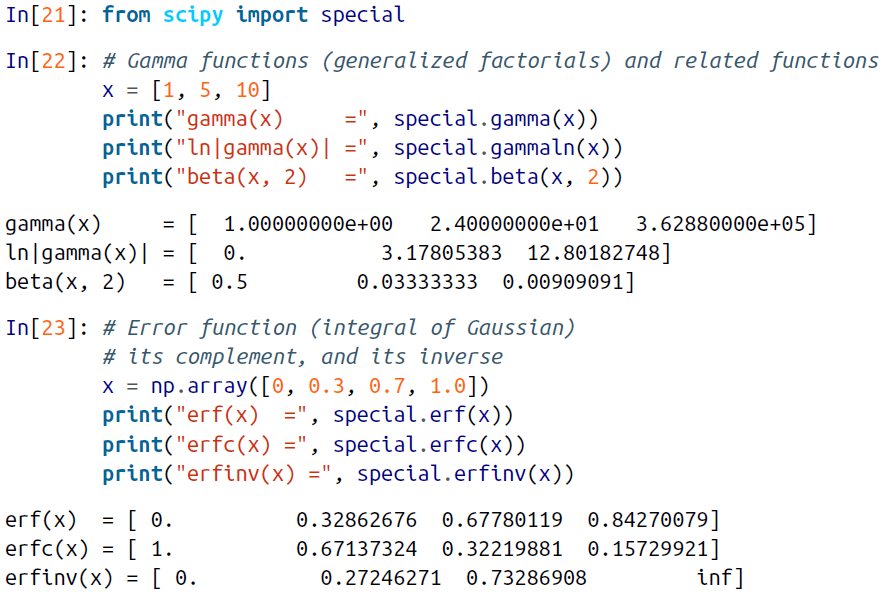
🡪 When x is very small, these functions give more precise values than if the raw np.log or np.exp were used.

**🡺Specialized unfuncs**

🡪 NumPy has many more ufuncs available, including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the NumPy documentation reveals a lot of interesting functionality.

🡪 Another excellent source for more specialized and obscure ufuncs is the submodule scipy.special.

🡪 If you want to compute some obscure mathematical function on your data, chances are it is implemented in scipy.special. There are far too many functions to list them all, but the following snippet shows a couple that might come up in a statistics context:



🡪 There are many, many more ufuncs available in both NumPy and scipy.special. Because the documentation of these packages is available online, a web search along the lines of “gamma function python” will generally find the relevant information.

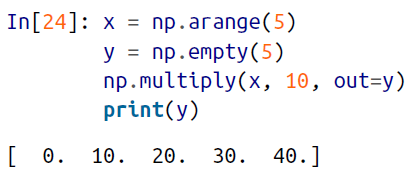
1. Advanced Ufunc Features

🡪 Many NumPy users make use of ufuncs without ever learning their full set of features. We’ll outline a few specialized features of ufuncs here.

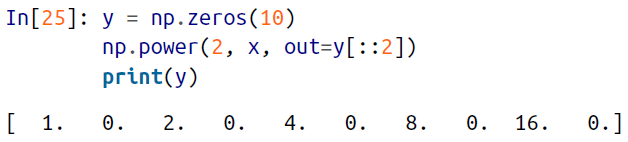
**🡺Specifying Output**

🡪 For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored.

🡪 Rather than creating a temporary array, you can use this to write computation results directly to the memory location where you’d like them to be. For all ufuncs, you can do this using the out argument of the function:



🡪 This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:



🡪 If we had instead written y[::2] = 2 \*\* x, this would have resulted in the creation of a temporary array to hold the results of 2 \*\* x, followed by a second operation copying those values into the y array.

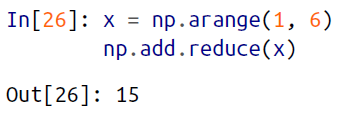
🡪 This doesn’t make much of a difference for such

a small computation, but for very large arrays the memory savings from careful use of the out argument can be significant.

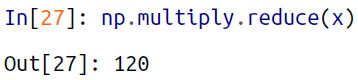
🡺Aggregates

🡪 For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we’d like to *reduce* an array with a particular operation, we can use the reduce method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

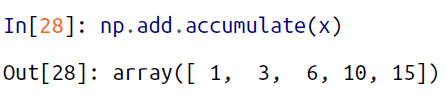
🡪 For example, calling reduce on the add ufunc returns the sum of all elements in the array:

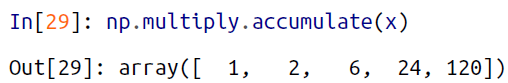


🡪 Similarly, calling reduce on the multiply ufunc results in the product of all array elements:



🡪 If we’d like to store all the intermediate results of the computation, we can instead use accumulate:

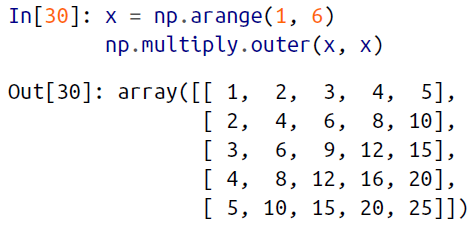




🡪 Note that for these particular cases, there are dedicated NumPy functions to compute the results (np.sum, np.prod, np.cumsum, np.cumprod)

🡺 **Outer products**

🡪 Finally, any ufunc can compute the output of all pairs of two different inputs using the outer method. This allows you, in one line, to do things like create a multiplication table:



🡪 Another extremely useful feature of ufuncs is the ability to operate between arrays of different sizes and shapes, a set of operations known as *broadcasting*.

🡪More about the ufuncs can be learned from the official docuementation of Numpy and we can also get the information about that by help of the **<tab>** and **?** operator

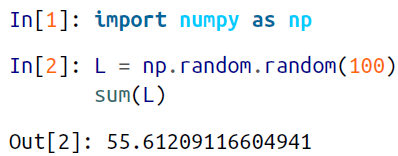
1. Aggregations: Min, Max, and Everything in Between

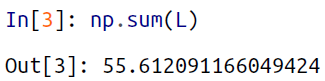
🡪 Often when you are faced with a large amount of data, a first step is to compute summary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the “typical” values in a dataset, but other aggregates are useful as well (the sum, product, median,minimum and maximum, quantiles, etc.).

🡪 NumPy has fast built-in aggregation functions for working on arrays; we’ll discuss and demonstrate some of them here.

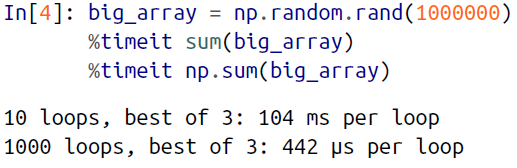
🡺**Summing the values in an array**

🡪 As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in sum function:

  
🡪 The syntax is quite similar to that of NumPy’s sum function, and the result is the same in the simplest case:



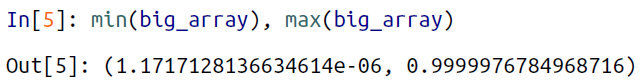
🡪 However, because it executes the operation in compiled code, NumPy’s version of the operation is computed much more quickly:



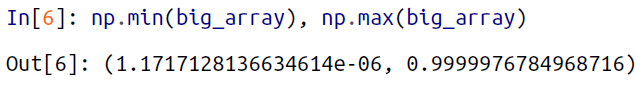
🡪 Be careful, though: the sum function and the np.sum function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings, and np.sum is aware of multiple array dimensions, as we will see in the following section.

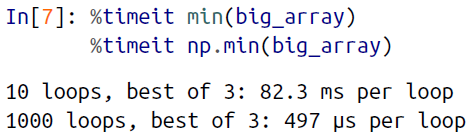
**🡺 Minimum and Maximum**

🡪 Similarly, Python has built-in min and max functions, used to find the minimum value and maximum value of any given array:

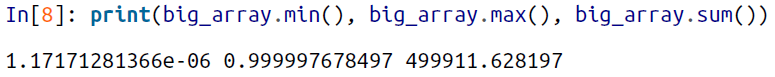


🡪 NumPy’s corresponding functions have similar syntax, and again operate much more quickly:





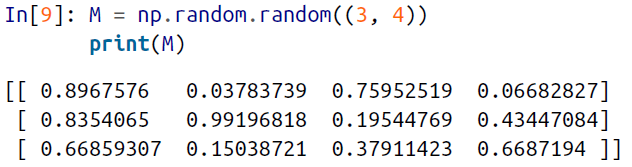
🡪 For min, max, sum, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:



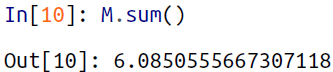
🡪 Whenever possible, make sure that you are using the NumPy version of these aggregates when operating on NumPy arrays!

**🡺 Multidimensional aggregates**

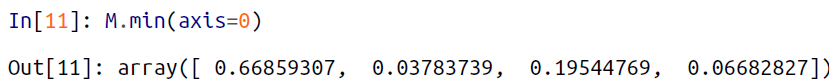
🡪 One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:



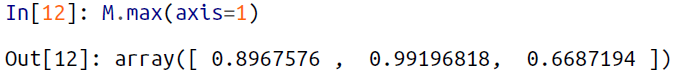
🡪 By default, each NumPy aggregation function will return the aggregate over the entire array:



🡪 Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within each column by specifying axis=0:



🡪 The function returns four values, corresponding to the four columns of numbers. Similarly, we can find the maximum value within each row:



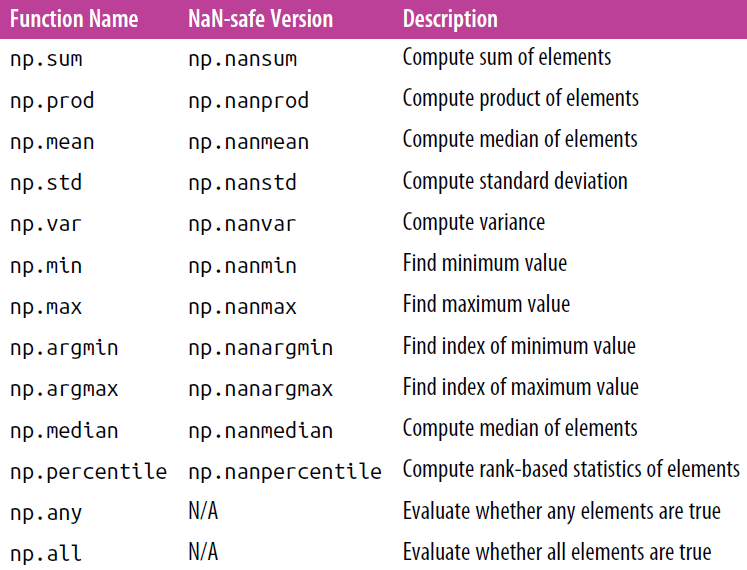
**🡪In the axis keyword as the parameter 0 means the column and 1 means the row**

**🡺Other aggregation function**

**🡪** NumPy provides many other aggregation functions, but we won’t discuss them in detail here.

🡪Additionally, most aggregates have a NaN-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point NaN value

🡪List of the usefull aggregation functions

****

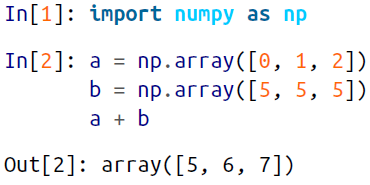
1. Computation on Arrays: Broadcasting

🡪We saw in the previous section how NumPy’s universal functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizingoperations is to use NumPy’s *broadcasting* functionality.

🡪 Broadcasting is simply a set of rules for applying binary ufuncs (addition, subtraction, multiplication, etc.) on arrays of different sizes.

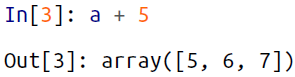
🡺 Introducing Broadcasting

🡪Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

****

🡪 Broadcasting allows these types of binary operations to be performed on arrays of different sizes—for example, we can just as easily add a scalar (think of it as a zero dimensional

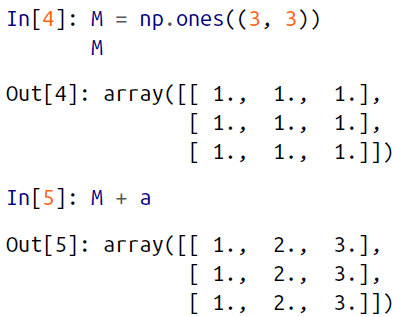
array) to an array:

****

🡪 We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5], and adds the results.

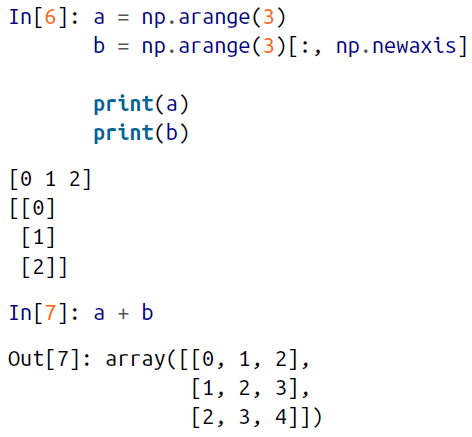
🡪The advantage of NumPy’s broadcasting is that this duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

🡪 We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

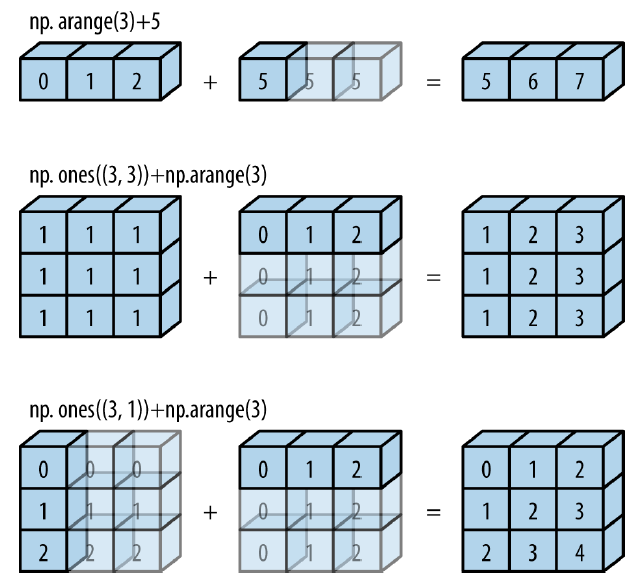


🡪 Here the one-dimensional array a is stretched, or broadcast, across the second dimension in order to match the shape of M.

🡪 While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider following example:



🡪 Just as before we stretched or broadcasted one value to match the shape of the other, here we’ve stretched *both* a and b to match a common shape, and the result is a two dimensional array! The geometry of these examples is visualized in figure bellow:

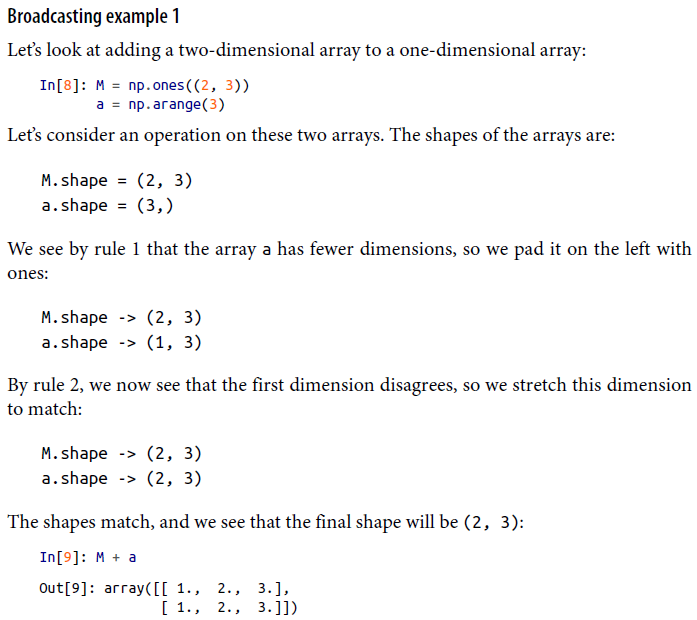


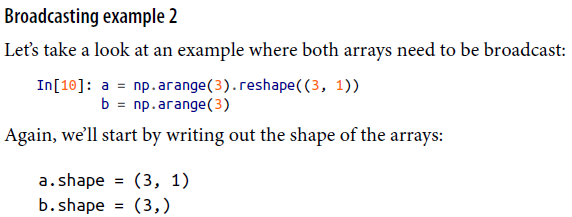
🡪 The light boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

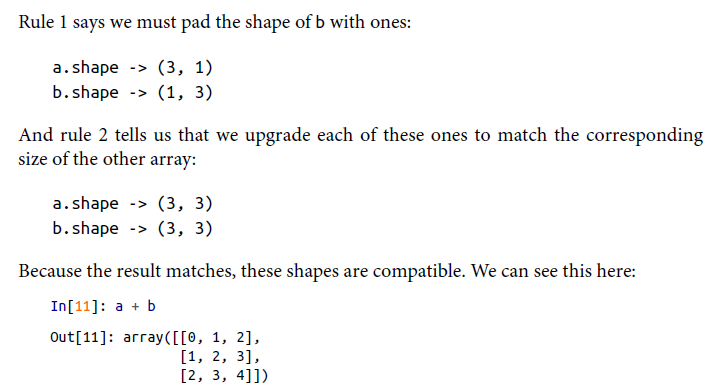
**🡺Rules Of Broadcasting**

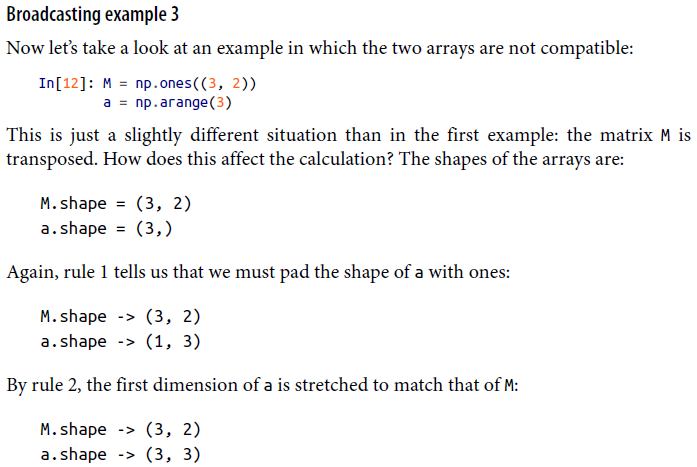
**🡪** Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

1. **Rule 1**: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
2. **Rule 2**: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape
3. **Rule 3**: If in any dimension the sizes disagree and neither is equal to 1, an error is raised





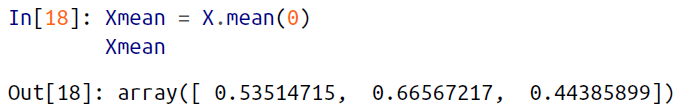




🡪Now we hit by rule-3, the final shapes do not match, so these two arrays are incompatiable, as we can observe by attempting this operation:

🡺 **Broadcasting in Practice**

🡪 We can compute the mean of each feature using the mean aggregate across the first dimension:

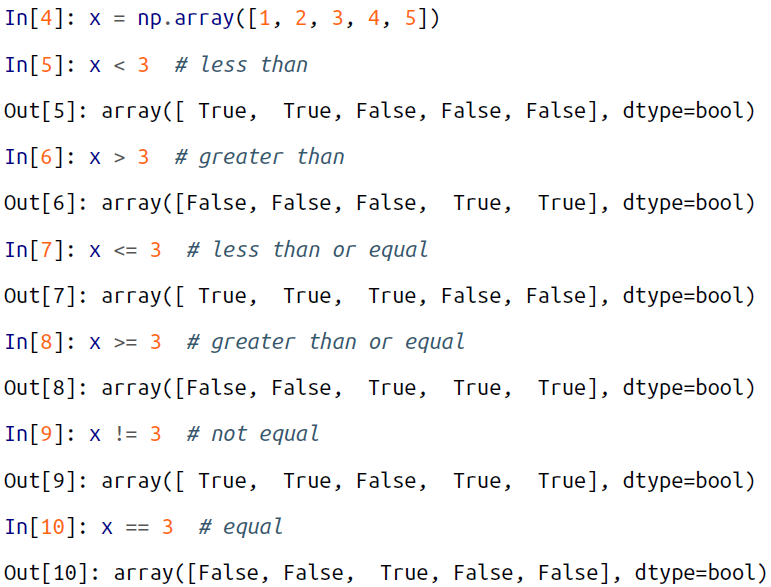


1. **Comparisons, Masks, and Boolean Logic**

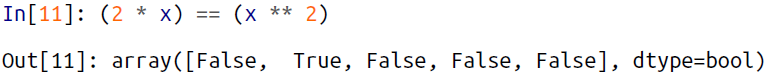
🡪This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

**🡺Comparison Operators as ufuncs**

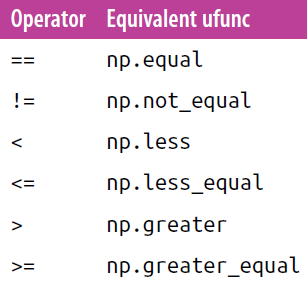
🡪Till now we are introduced to ufuncs, and focused in particular on arithmetic operators. We saw that using +, -, \*, /, and others on arrays leads to element-wise operations. NumPy also implements comparison operators such as < (less than) and > (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:



🡪 It is also possible to do an element-by-element comparison of two arrays, and to include compound expressions:



🡪 As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write x < 3, internally NumPy uses np.less(x, 3). A summary of the comparison operators and their equivalent ufunc is shown here:

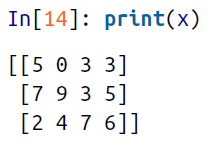


🡪 Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape.

🡪 In each case, the result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

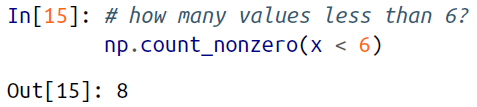
**🡺Working with Boolean Arrays**

**🡪** Given a Boolean array, there are a host of useful operations you can do. We’ll work with x, the two-dimensional array we created earlier:

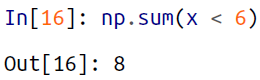


**🡺 Counting entries**

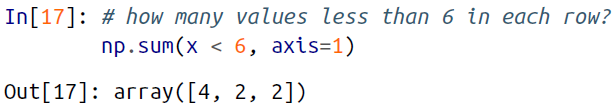
🡪 To count the number of True entries in a Boolean array, np.count\_nonzero is useful:



🡪 We see that there are eight array entries that are less than 6. Another way to get at this information is to use np.sum; in this case, False is interpreted as 0, and True is interpreted as 1:

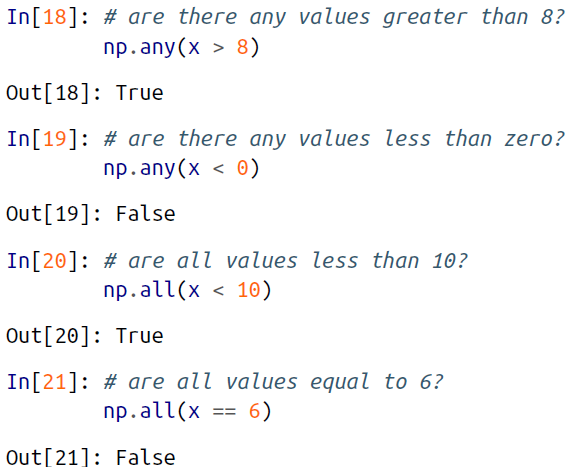


🡪 The benefit of sum() is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

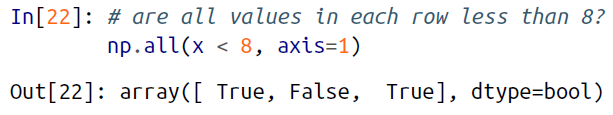


**🡪 This counts the number of values less than 6 in each row of the matrix.**

🡪 If we’re interested in quickly checking whether any or all the values are true, we can use (you guessed it) np.any() or np.all():



🡪 np.all() and np.any() can be used along particular axes as well. For example:



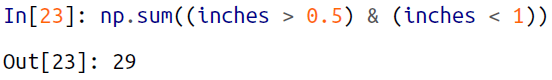
**🡪 Here all the elements in the first and third rows are less than 8, while this is not the case for the second row.**

🡪 Python has built-in sum(), any(), and all() functions, and in particular will fail or produce unintended results when used on multidimensional arrays. Be sure that you are using np.sum(), np.any(), and np.all() for these examples!

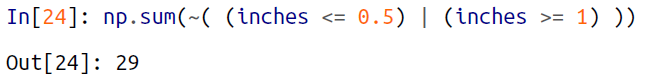
**🡺 Boolean operators**

🡪In python there are bitwise operators such as &, |, ^, ~

🡪 For example, we can address this sort of compound question as follows:

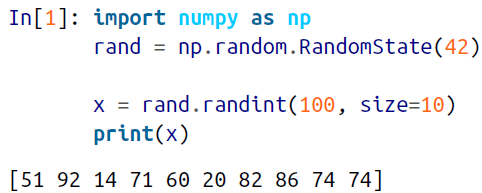


🡪 Using the equivalence of *A AND B* and *NOT (A OR B)* (which you may remember if you’ve taken an introductory logic course), we can compute the same result in a different manner:

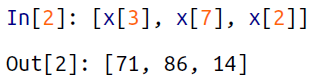


1. Exploring Fancy Indexing

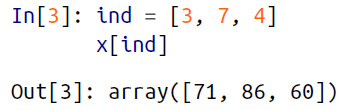
🡪 Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once. For example, consider the following array:



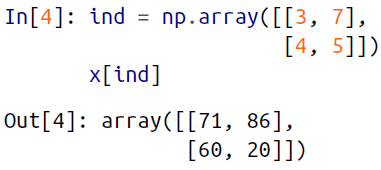
🡪 Suppose we want to access three different elements. We could do it like this:



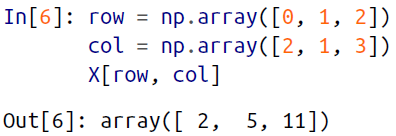
🡪 Alternatively, we can pass a single list or array of indices to obtain the same result:



🡪 With fancy indexing, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

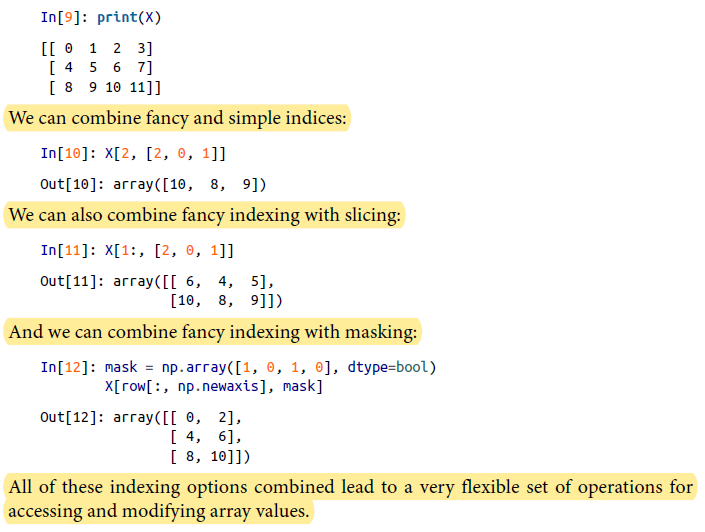
**

🡪 Like with standard indexing, the first index refers to the row, and the second to the column:

**

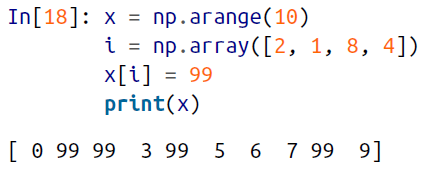
1. *Combined Indexing*

*🡪* For even more powerful operations, fancy indexing can be combined with the other indexing schemes we’ve seen:

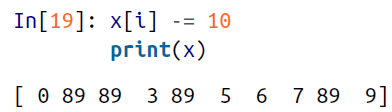
**

***🡺* Modifying Values with Fancy Indexing**

*🡪* Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array. For example, imagine we have an array of indices and we’d like to set the corresponding items in an array to some value:



🡪 We can use any assignment-type operator for this. For example:



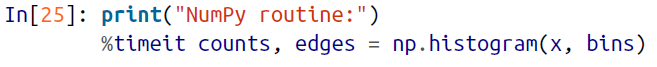
1. **Example: Binning Data**

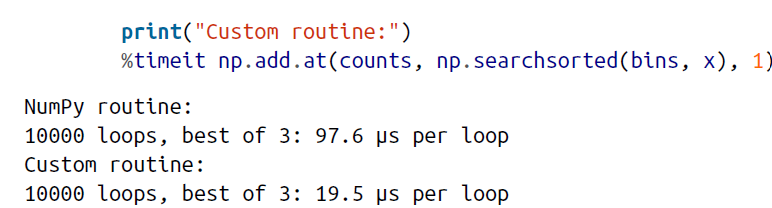
🡪You can use these ideas to efficiently bin data to create a histogram by hand.

🡪 Of course, it would be silly to have to do this each time you want to plot histogram. This is why Matplotlib provides the plt.hist() routine, which does the same in a single line:

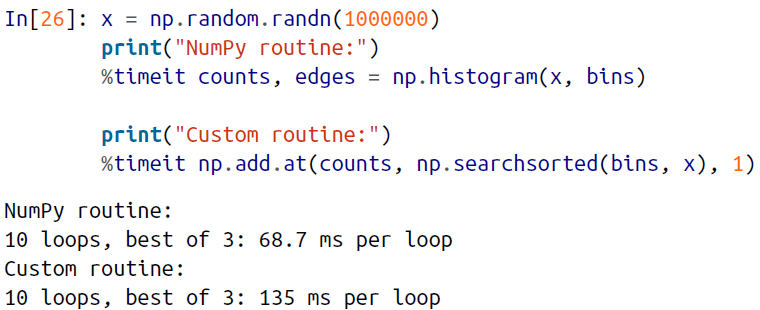


🡪 This function will create a nearly identical plot to the one seen here. To compute the binning, Matplotlib uses the np.histogram function, which does a very similar computation to what we did before. Let’s compare the two here:





🡪 Our own one-line algorithm is several times faster than the optimized algorithm in NumPy! How can this be? If you dig into the np.histogram source code (you can do this in IPython by typing **np.histogram??**), you’ll see that it’s quite a bit more involved than the simple search and-count that we’ve done; this is because NumPy’s algorithm is more flexible, and particularly is designed for better performance when the number of data points becomes large:

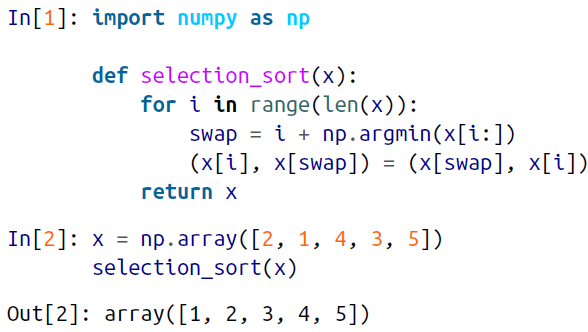


🡪 What this comparison shows is that algorithmic efficiency is almost never a simple question. An algorithm efficient for large datasets will not always be the best choice for small datasets, and vice versa

1. Sorting Arrays

🡪 This section covers algorithms related to sorting values in NumPy arrays.

🡪 For example, a simple *selection sort* repeatedly finds the minimum value from a list, and makes swaps until the list is sorted. We can code this in just a few lines of Python:



🡪 Fortunately, Python contains built-in sorting algorithms that are *much* more efficient than either of the simplistic algorithms

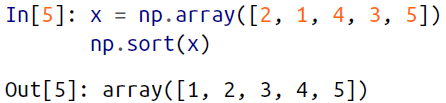
🡪 We’ll start by looking at the Python built-ins, and then take a look at the routines included in NumPy and optimized for NumPy arrays.

**🡺 Fast Sorting in NumPy: np.sort and np.argsort**

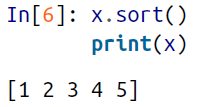
**🡪** Although Python has built-in sort and sorted functions to work with lists, we won’tdiscuss them here because NumPy’s np.sort function turns out to be much more

🡪 efficient and useful for our purposes. By default np.sort uses an *N* log *N* , *quicksort* algorithm, though *mergesort* and *heapsort* are also available. For most applications,the default quicksort is more than sufficient.

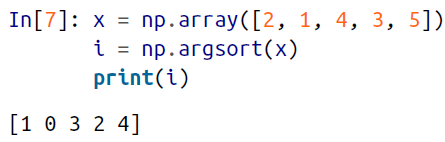
🡪 To return a sorted version of the array without modifying the input, you can use np.sort:

****

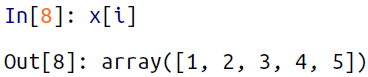
🡪 If you prefer to sort the array in-place, you can instead use the sort method of arrays:

****

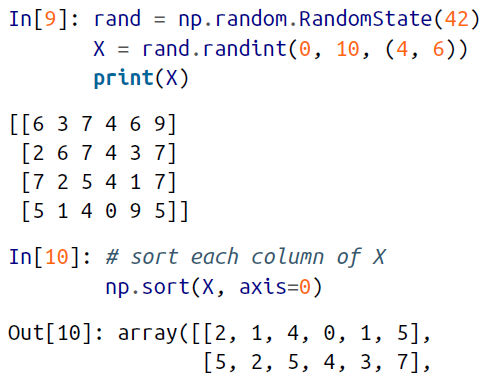
🡪 A related function is argsort, which instead returns the *indices* of the sorted elements:

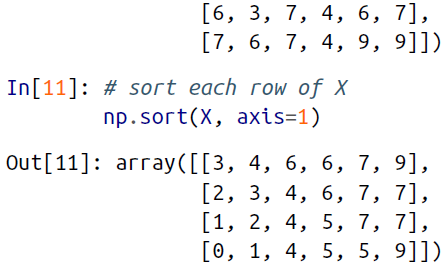
****

🡪 The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, and so on. These indices can then be used (via fancy indexing) to construct the sorted array if desired:

****’

**🡺Sorting along rows or columns**

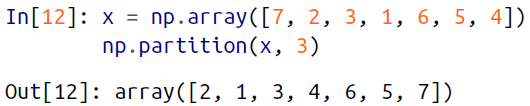
🡪 A useful feature of NumPy’s sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument. For example:  
 ****

****

🡪 Keep in mind that this treats each row or column as an independent array, and any relationships between the row or column values will be lost!

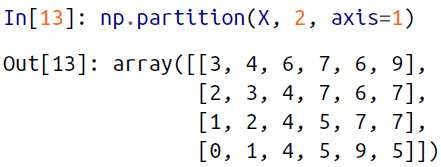
**🡺 Partial Sorts: Partitioning**

🡪Sometimes we’re not interested in sorting the entire array, but simply want to find the *K* smallest values in the array. NumPy provides this in the np.partition function. np.partition takes an array and a number *K*; the result is a new array with the smallest *K* values to the left of the partition, and the remaining values to the right, in arbitrary order:



🡪 Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have arbitrary order.

🡪 Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:



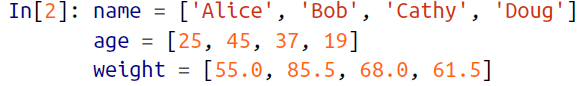
🡪 Finally, just as there is a np.argsort that computes indices of the sort, there is a np.argpartition that computes indices of the partition.

1. **Structured Data: NumPy’s Structured Arrays**

🡪 While often our data can be well represented by a homogeneous array of values, sometimes this is not the case.

🡪 This section demonstrates the use of NumPy’s *structured arrays* and *record arrays*, which provide efficient storage for compound, hetero‐geneous data. While the patterns shown here are useful for simple operations,scenarios like this often lend themselves to the use of Pandas DataFrames, which we’llexplore in pandas

🡪 Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we’d like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

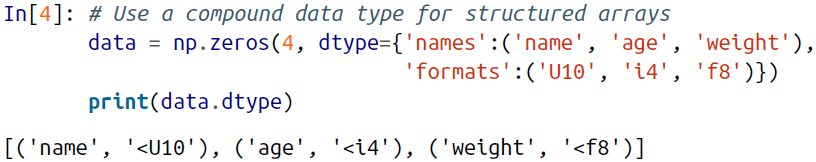


🡪 But this is a bit clumsy. There’s nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

🡪 Recall that previously we created a simple array using an expression like this:

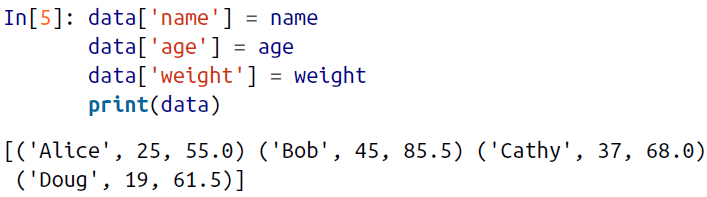


🡪 We can similarly create a structured array using a compound data type specification:



🡪 Here 'U10' translates to “Unicode string of maximum length 10,” 'i4' translates to “4-byte (i.e., 32 bit) integer,” and 'f8' translates to “8-byte (i.e., 64 bit) float.” We’ll discuss other options for these type codes in the following section.

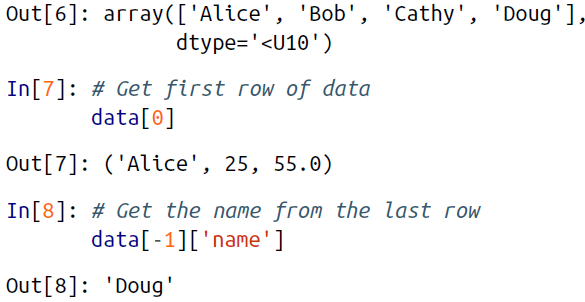
🡪 Now that we’ve created an empty container array, we can fill the array with our lists of values:



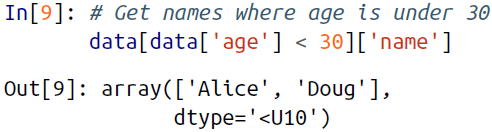
🡪 As we had hoped, the data is now arranged together in one convenient block of memory.

🡪The handy thing with structured arrays is that you can now refer to values either by index or by name:





🡪 Using Boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

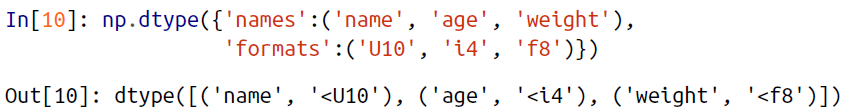


🡪 Note that if you’d like to do any operations that are any more complicated than these, you should probably consider the Pandas package, covered in the next chapter.

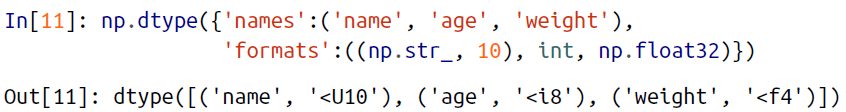
🡪As we’ll see, Pandas provides a DataFrame object, which is a structure built on NumPy arrays that offers a variety of useful data manipulation functionality similar to what we’ve shown here, as well as much, much more.

1. **Creating Structured Arrays**

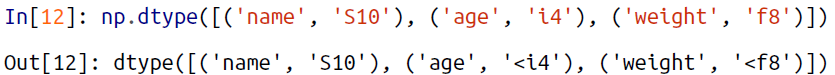
🡪 Structured array data types can be specified in a number of ways. Earlier, we saw the dictionary method:



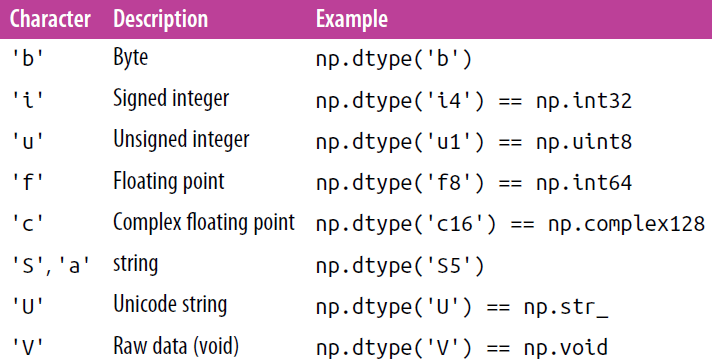
🡪 For clarity, numerical types can be specified with Python types or NumPy dtypes instead:



🡪 A compound type can also be specified as a list of tuples:



🡪 The shortened string format codes may seem confusing, but they are built on simple principles. The first (optional) character is < or >, which means “little endian” or “big endian,” respectively, and specifies the ordering convention for significant bits.



1. **On to Pandas**

🡪This section on structured and record arrays is purposely at the end of this chapter, because it leads so well into the next package we will cover: Pandas. Structured arrays like the ones discussed here are good to know about for certain situations, especially in case you’re using NumPy arrays to map onto binary data formats in C, Fortran, or another language. For day-to-day use of structured data, the Pandas package is a much better choice