

```
In[29]: np.multiply.accumulate(x)

Out[29]: array([ 1,  2,  6, 24, 120])
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`), which we'll explore in [“Aggregations: Min, Max, and Everything in Between” on page 58](#).

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:

```
In[30]: x = np.arange(1, 6)
        np.multiply.outer(x, x)

Out[30]: array([[ 1,  2,  3,  4,  5],
                [ 2,  4,  6,  8, 10],
                [ 3,  6,  9, 12, 15],
                [ 4,  8, 12, 16, 20],
                [ 5, 10, 15, 20, 25]])
```

The `ufunc.at` and `ufunc.reduceat` methods, which we'll explore in [“Fancy Indexing” on page 78](#), are very helpful as well.

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*. This subject is important enough that we will devote a whole section to it (see [“Computation on Arrays: Broadcasting” on page 63](#)).

Ufuncs: Learning More

More information on universal functions (including the full list of available functions) can be found on the [NumPy](#) and [SciPy](#) documentation websites.

Recall that you can also access information directly from within IPython by importing the packages and using IPython's tab-completion and help (?) functionality, as described in [“Help and Documentation in IPython” on page 3](#).

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:

```
In[1]: import numpy as np
In[2]: L = np.random.random(100)
        sum(L)
Out[2]: 55.61209116604941
```

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

```
In[3]: np.sum(L)
Out[3]: 55.612091166049424
```

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

```
In[4]: big_array = np.random.rand(1000000)
        %timeit sum(big_array)
        %timeit np.sum(big_array)

10 loops, best of 3: 104 ms per loop
1000 loops, best of 3: 442 µs per loop
```

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:

```
In[5]: min(big_array), max(big_array)
Out[5]: (1.1717128136634614e-06, 0.9999976784968716)
```

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

```
In[6]: np.min(big_array), np.max(big_array)
Out[6]: (1.1717128136634614e-06, 0.9999976784968716)
```