02.04-Computation-on-arrays-aggregates

May 8, 2018

This notebook contains an excerpt from the Python Data Science Handbook by Jake VanderPlas; the content is available on GitHub.

1 Aggregations: Min, Max, and Everything In Between

Often when 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.

1.1 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 ţs 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.

1.2   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)
```

```
In [7]: %timeit min(big_array)
        %timeit np.min(big_array)
```

```
10 loops, best of 3: 82.3 ms per loop
1000 loops, best of 3: 497 ţs per loop
```

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

```
In [8]: print(big_array.min(), big_array.max(), big_array.sum())
```

```
1.17171281366e-06 0.999997678497 499911.628197
```

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

1.2.1   Multi dimensional 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:

```
In [9]: M = np.random.random((3, 4))
        print(M)
```

```
[[ 0.8967576    0.03783739  0.75952519  0.06682827]
 [ 0.8354065    0.99196818  0.19544769  0.43447084]
 [ 0.66859307   0.15038721  0.37911423  0.6687194 ]]
```

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

```
In [10]: M.sum()
```

```
Out[10]: 6.0850555667307118
```

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`:

```
In [11]: M.min(axis=0)
```

```
Out[11]: array([ 0.66859307,  0.03783739,  0.19544769,  0.06682827])
```

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

```
In [12]: M.max(axis=1)
```

```
Out[12]: array([ 0.8967576 ,  0.99196818,  0.6687194 ])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array that will be collapsed*, rather than the dimension that will be returned. So specifying `axis=0` means that the first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

1.2.2 Other aggregation functions

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 (for a fuller discussion of missing data, see Handling Missing Data). Some of these `NaN`-safe functions were not added until NumPy 1.8, so they will not be available in older NumPy versions.

The following table provides a list of useful aggregation functions available in NumPy:

Function Name	NaN-safe Version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute mean of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmin	np.nanargmin	Find index of minimum value

Function Name	NaN-safe Version	Description
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

We will see these aggregates often throughout the rest of the book.

1.3 Example: What is the Average Height of US Presidents?

Aggregates available in NumPy can be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all US presidents. This data is available in the file *president_heights.csv*, which is a simple comma-separated list of labels and values:

```
In [13]: !head -4 data/president_heights.csv

order,name,height(cm)

1,George Washington,189

2,John Adams,170

3,Thomas Jefferson,189
```

We'll use the Pandas package, which we'll explore more fully in Chapter 3, to read the file and extract this information (note that the heights are measured in centimeters).

```
In [14]: import pandas as pd
         data = pd.read_csv('data/president_heights.csv')
         heights = np.array(data['height(cm)'])
         print(heights)

[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173
 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
 177 185 188 188 182 185]
```

Now that we have this data array, we can compute a variety of summary statistics:

```
In [15]: print("Mean height:       ", heights.mean())
         print("Standard deviation:", heights.std())
         print("Minimum height:    ", heights.min())
         print("Maximum height:    ", heights.max())
```

```
Mean height:       179.738095238
Standard deviation: 6.93184344275
Minimum height:    163
Maximum height:    193
```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

```
In [16]: print("25th percentile:   ", np.percentile(heights, 25))
         print("Median:            ", np.median(heights))
         print("75th percentile:   ", np.percentile(heights, 75))

25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

We see that the median height of US presidents is 182 cm, or just shy of six feet.

Of course, sometimes it's more useful to see a visual representation of this data, which we can accomplish using tools in Matplotlib (we'll discuss Matplotlib more fully in Chapter 4). For example, this code generates the following chart:

```
In [17]: %matplotlib inline
         import matplotlib.pyplot as plt
         import seaborn; seaborn.set()  # set plot style

In [18]: plt.hist(heights)
         plt.title('Height Distribution of US Presidents')
         plt.xlabel('height (cm)')
         plt.ylabel('number');
```

Height Distribution of US Presidents
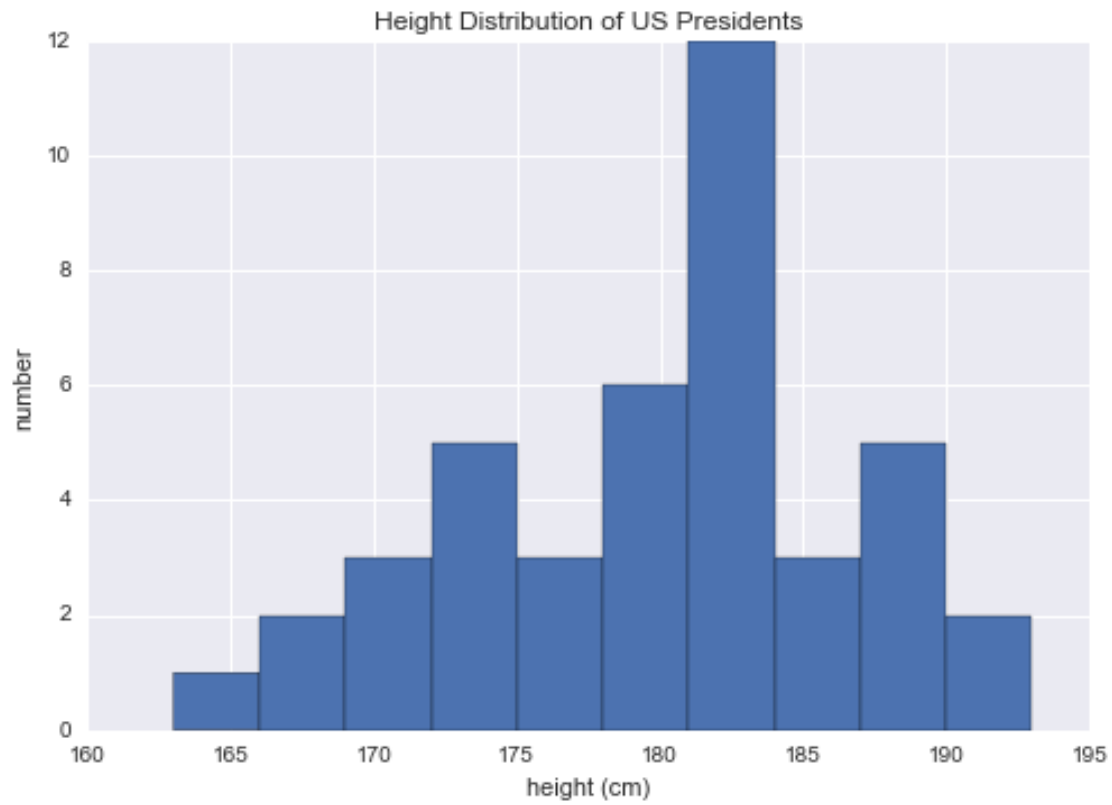
These aggregates are some of the fundamental pieces of exploratory data analysis that we'll explore in more depth in later chapters of the book.