Aggregations: min, max, and Everything in Between

A first step in exploring any dataset is often to compute various summary statistics. 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 aggregations 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 try out 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
        rng = np.random.default_rng()
```

```
In [2]: L = rng.random(100)
        sum(L)

Out[2]: 52.76825337322368
```

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]: 52.76825337322366
```

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

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

        89.9 ms ± 233 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
        521 µs ± 8.37 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

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 (`sum(x, 1)` initializes the sum at `1` , while `np.sum(x, 1)` sums along axis `1`), 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]:
```
(2.0114398036064074e-07, 0.9999997912802653)
```

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]:
```
(2.0114398036064074e-07, 0.9999997912802653)
```

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

```
72 ms ± 177 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
564 µs ± 3.11 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

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())
```

```
2.0114398036064074e-07 0.9999997912802653 499854.0273321711
```

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:

In [9]:
```
M = rng.integers(0, 10, (3, 4))
print(M)
```

```
[[0 3 1 2]
 [1 9 7 0]
 [4 8 3 7]]
```

NumPy aggregations will apply across all elements of a multidimensional array:

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

```
Out[10]: 45
```

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, 3, 1, 0])
```

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([3, 9, 8])
```

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 axis 0 will be collapsed: for two-dimensional arrays, values within each column will be aggregated.

Other Aggregation Functions

NumPy provides several other aggregation functions with a similar API, and additionally most have a `NaN`-safe counterpart that computes the result while ignoring missing values, which are marked by the special IEEE floating-point `NaN` value (see Handling Missing Data (03.04-Missing-Values.ipynb)).

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
`np.argmax`	`np.nanargmax`	Find index of maximum value

Function name	NaN-safe version	Description
`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

Example: What Is the Average Height of US Presidents?

Aggregates available in NumPy can act as summary statistics for a set of values. As a small example, let's consider the heights of all US presidents. This data is available in the file *president_heights.csv*, which is a 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 Part 3 (03.00-Introduction-to-Pandas.ipynb), 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
 177 185 188 188 182 185 191 182]
```

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:        180.04545454545453
Standard deviation: 6.983599441335736
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

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

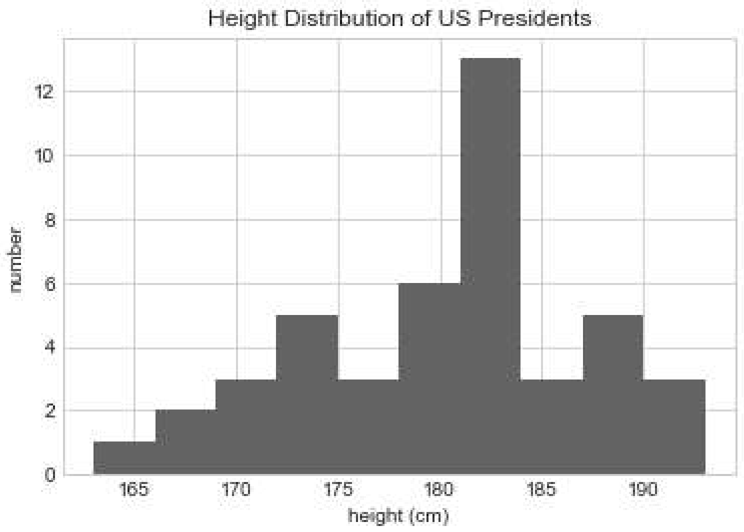
```
25th percentile:    174.75
Median:             182.0
75th percentile:    183.5
```

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 Part 4 (04.00-Introduction-To-Matplotlib.ipynb)). For example, this code generates the following chart:

In [17]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
```

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


Height Distribution of US Presidents
number
height (cm)