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.

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]:
```
47.0819315465058
```

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]:
```
47.081931546505814
```

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)
```

```
156 ms ± 14.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.38 ms ± 266 µs per loop (mean ± std. dev. of 7 runs, 1,000 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, 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]:
```
(3.135385551189529e-07, 0.9999965021968933)
```

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]:
```
(3.135385551189529e-07, 0.9999965021968933)
```

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

```
94.1 ms ± 5.49 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
The slowest run took 4.43 times longer than the fastest. This could mean that an intermediate result is being cached.
1.38 ms ± 693 µs per loop (mean ± std. dev. of 7 runs, 1,000 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())
```

```
3.135385551189529e-07 0.9999965021968933 499746.95278424013
```

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

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)
```

```
[[0.7917884  0.84918645 0.9336178  0.20883576]
 [0.75553428 0.04762039 0.17993029 0.1912069 ]
 [0.18394151 0.56042173 0.18359826 0.46760206]]
```

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

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

Out[10]:
```
5.353283847552185
```

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) #column wise min
#M.min(axis=1) #row wise min
```

Out[11]:
```
array([0.18394151, 0.04762039, 0.17993029, 0.1912069 ])
```

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) #row wise max
#M.max(axis=0) #column wise max
```

Out[12]:
```
array([0.9336178 , 0.75553428, 0.56042173])
```

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.

In [13]:
```
%timeit sum(M)
%timeit np.sum(M)
```

```
7.51 µs ± 830 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
7.69 µs ± 961 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [14]:
```
%timeit M.min()
%timeit np.min(M)
```

```
3.54 µs ± 480 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
6.6 µs ± 329 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [15]:
```
%timeit M.max()
%timeit np.max(M)
```

```
3.79 µs ± 477 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
6.29 µs ± 138 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [16]:
```
%timeit np.min(M, axis=1)
%timeit np.min(M, axis=0)
```

```
7.8 µs ± 303 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
8.2 µs ± 996 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

In [17]:
```
%timeit np.max(M, axis=1)
%timeit np.max(M, axis=0)
```

```
8.14 µs ± 548 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
8.23 µs ± 460 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

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. 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

Function Name	NaN-safe Version	Description
`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
`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 be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all US presidents.

In [19]:
```
us_presidents_height = {
    "George Washington": {"height_cm": 188, "height_in": 74},
    "Abraham Lincoln": {"height_cm": 193, "height_in": 76},
    "Theodore Roosevelt": {"height_cm": 178, "height_in": 70},
    "Franklin D. Roosevelt": {"height_cm": 188, "height_in": 74},
    "John F. Kennedy": {"height_cm": 183, "height_in": 72},
    "Lyndon B. Johnson": {"height_cm": 193, "height_in": 76},
    "Richard Nixon": {"height_cm": 182, "height_in": 71.5},
    "Gerald Ford": {"height_cm": 183, "height_in": 72},
    "Jimmy Carter": {"height_cm": 177, "height_in": 69.5},
```

```
    "Ronald Reagan": {"height_cm": 185, "height_in": 73},
    "George H. W. Bush": {"height_cm": 188, "height_in": 74},
    "Bill Clinton": {"height_cm": 188, "height_in": 74},
    "George W. Bush": {"height_cm": 182, "height_in": 71.5},
    "Barack Obama": {"height_cm": 185, "height_in": 73},
    "Donald Trump": {"height_cm": 190, "height_in": 75},
    "Joe Biden": {"height_cm": 183, "height_in": 72}
}
```

In [20]:
```
# Extract heigths in inches
height_in=[]
for info in us_presidents_height.values():
    height_in.append(info['height_in'])
```

In [22]:
```
#Average height
avg_height_in=sum(height_in)/len(height_in)
avg_height_in
```

Out[22]:
```
72.96875
```

In [23]:
```
print('Average Height:', np.mean(height_in))
print('Standard Deviation:',np.std(height_in))
print('Standard Deviation:',np.std(height_in))
print('Minimun Height:',np.min(height_in))
print('Maximun Heigth:',np.max(height_in))
```

```
Average Height: 72.96875
Standard Deviation: 1.8495670946197114
Standard Deviation: 1.8495670946197114
Minimun Height: 69.5
Maximun Heigth: 76.0
```

In [24]:
```
# Or you can use list comprehension
# Extract height in inches
heights_in = [info["height_in"] for info in us_presidents_height.values()]
# It extracts all the heights in inches from the nested dictionaries of each U.S. president and stores them in a list.
# loops through each of these inner dictionaries and picks the "height_in" value.

# Compute average
average_height_in = sum(heights_in) / len(heights_in)
```

```
print(f"Average height of selected U.S. Presidents: {average_height_in:.2f} inches")
```

```
Average height of selected U.S. Presidents: 72.97 inches
```

In [25]:
```
heights_in
```

Out[25]:
```
[74, 76, 70, 74, 72, 76, 71.5, 72, 69.5, 73, 74, 74, 71.5, 73, 75, 72]
```

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

In [26]:
```
print("Mean height:       ", heights_in.mean())
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[26], line 1
----> 1 print("Mean height:       ", heights_in.mean())

AttributeError: 'list' object has no attribute 'mean'
```

AttributeError: 'list' object has no attribute 'mean' so we have to convert this height into numpy array

In [27]:
```
heights_in=np.array(heights_in)
```

In [28]:
```
print("Mean height:       ", heights_in.mean())
print("Standard deviation:", heights_in.std())
print("Minimum height:    ", heights_in.min())
print("Maximum height:    ", heights_in.max())
```

```
Mean height:        72.96875
Standard deviation: 1.8495670946197114
Minimum height:     69.5
Maximum height:     76.0
```

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 [29]:
```
print("25th percentile:   ", np.percentile(heights_in, 25))
print("Median:            ", np.median(heights_in))
```

```python
print("75th percentile:   ", np.percentile(heights_in, 75))
```

```
25th percentile:    71.875
Median:             73.0
75th percentile:    74.0
```