

Lecture 6 : Numpy (Advanced topics)

Data Science, DST, UIC

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.

```
In [1]: import numpy as np
```

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

```
Out[2]: 50.65184536698634
```

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]: 50.65184536698634
```

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)
```

```
72.2 ms ± 431 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
455 µs ± 4.72 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, 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]: (1.185000205738973e-07, 0.9999990419034759)
```

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]: (1.185000205738973e-07, 0.9999990419034759)
```

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

49.9 ms ± 274 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
314 µs ± 8.78 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())
1.185000205738973e-07 0.9999990419034759 500104.11629823095
```

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)

[[0.57290799 0.85510471 0.90853872 0.35495418]
 [0.40228179 0.20102073 0.17159723 0.69454527]
 [0.70900385 0.57881406 0.59033559 0.75935694]]
```

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

```
In [10]: print(M.sum())
          print(M.sum(axis=0)) # specify the axis along which to perform sum

6.798461063405746
[1.68419363 1.6349395 1.67047154 1.80885638]
```

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([0.40228179, 0.20102073, 0.17159723, 0.35495418])
```

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([0.90853872, 0.69454527, 0.75935694])
```

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.

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
<code>np.sum</code>	<code>np.nansum</code>	Compute sum of elements
<code>np.prod</code>	<code>np.nanprod</code>	Compute product of elements
<code>np.mean</code>	<code>np.nanmean</code>	Compute mean of elements
<code>np.std</code>	<code>np.nanstd</code>	Compute standard deviation
<code>np.var</code>	<code>np.nanvar</code>	Compute variance
<code>np.min</code>	<code>np.nanmin</code>	Find minimum value
<code>np.max</code>	<code>np.nanmax</code>	Find maximum value
<code>np.argmin</code>	<code>np.nanargmin</code>	Find index of minimum value
<code>np.argmax</code>	<code>np.nanargmax</code>	Find index of maximum value
<code>np.median</code>	<code>np.nanmedian</code>	Compute median of elements
<code>np.percentile</code>	<code>np.nanpercentile</code>	Compute rank-based statistics of elements
<code>np.any</code>	N/A	Evaluate whether any elements are true
<code>np.all</code>	N/A	Evaluate whether all elements are true

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

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:

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

```
In [13]: import pandas as pd
data = pd.read_csv('president_heights.csv')
data.head(5)
```

```
Out[13]:
```

	order	name	height(cm)
0	1	George Washington	189
1	2	John Adams	170
2	3	Thomas Jefferson	189
3	4	James Madison	163
4	5	James Monroe	183

```
In [14]: 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:      179.73809523809524
Standard deviation: 6.931843442745892
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))

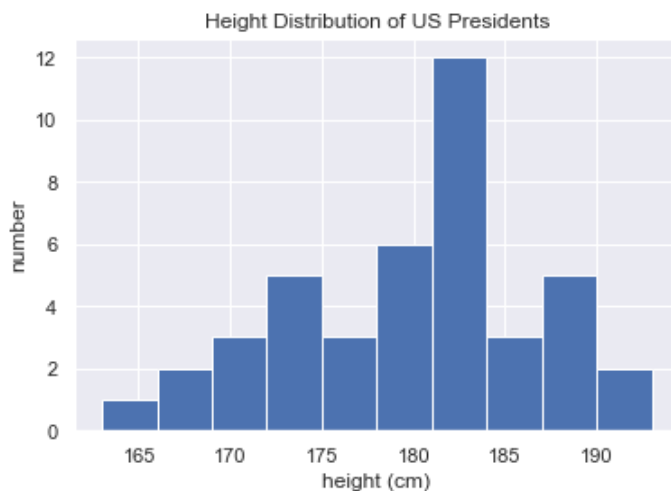
25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

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 . For example, this code generates the following chart:

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

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



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 vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying binary ufuncs (e.g., 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:

```
In [17]: a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

```
Out[17]: array([5, 6, 7])
```

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:

```
In [18]: a + 5
```

```
Out[18]: array([5, 6, 7])
```

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:

```
In [19]: M = np.ones((3, 3))
M
```

```
Out[19]: array([[1., 1., 1.],
               [1., 1., 1.],
               [1., 1., 1.]])
```

```
In [20]: M + a
```

```
Out[20]: array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.]])
```

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 the following example:

```
In [21]: a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

print(a)
print(b)

[0 1 2]
[[0]
 [1]
 [2]]
```

```
In [22]: a + b
```

```
Out[22]: array([[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
```

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!

Rules of Broadcasting

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

- 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.
- 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.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail.

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

```
In [23]: M = np.ones((2, 3))
a = np.arange(3)
print(M)
print(a)
```

```
[[1.  1.  1.]
 [1.  1.  1.]]
[0 1 2]
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `M.shape = (2, 3)`
- `a.shape = (3,)`

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)` :

```
In [24]: M + a
```

```
Out[24]: array([[1., 2., 3.],
                [1., 2., 3.]])
```

Broadcasting example 2

Let's take a look at an example where both arrays need to be broadcast:

```
In [25]: a = np.arange(3).reshape((3, 1))
b = np.arange(3)
print(a)
print(b)
```

```
[[0]
 [1]
 [2]]
[0 1 2]
```

Again, we'll start by writing out the shape of the arrays:

- `a.shape = (3, 1)`
- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`
- `b.shape -> (1, 3)`

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

Because the result matches, these shapes are compatible. We can see this here:

```
In [26]: a + b
Out[26]: array([[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
```

Broadcasting example 3

Now let's take a look at an example in which the two arrays are not compatible:

```
In [27]: M = np.ones((3, 2))
         a = np.arange(3)
         print(M)
         print(a)

[[1.  1.]
 [1.  1.]
 [1.  1.]]
[0 1 2]
```

This is just a slightly different situation than in the first example: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape = (3, 2)`
- `a.shape = (3,)`

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape -> (3, 2)`
- `a.shape -> (1, 3)`

By rule 2, the first dimension of `a` is stretched to match that of `M`:

- `M.shape -> (3, 2)`
- `a.shape -> (3, 3)`

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


```
In [28]: M + a
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-28-8cac1d547906> in <module>  
----> 1 M + a  
  
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding `a`'s shape with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword):

```
In [29]: a[:, np.newaxis].shape
```

```
Out[29]: (3, 1)
```

```
In [30]: M + a[:, np.newaxis]
```

```
Out[30]: array([[1., 1.],  
               [2., 2.],  
               [3., 3.]])
```

Also note that while we've been focusing on the `+` operator here, these broadcasting rules apply to *any* binary ufunc. For example, here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with more precision than the naive approach:

```
In [31]: np.logaddexp(M, a[:, np.newaxis])
```

```
Out[31]: array([[1.31326169, 1.31326169],  
               [1.69314718, 1.69314718],  
               [2.31326169, 2.31326169]])
```

Broadcasting in Practice

We'll now take a look at a couple simple examples of where they can be useful.

Centering an array

In the previous section, we saw that ufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. One commonly seen example is when centering an array of data. Imagine you have an array of 10 observations, each of which consists of 3 values. We'll store this in a 10×3 array:

```
In [32]: X = np.random.random((10, 3))
```

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

```
In [33]: Xmean = X.mean(0)  
Xmean
```

```
Out[33]: array([0.4541563 , 0.38313029, 0.56174229])
```

And now we can center the `X` array by subtracting the mean (this is a broadcasting operation):

```
In [34]: X_centered = X - Xmean
```

To double-check that we've done this correctly, we can check that the centered array has near zero mean:

```
In [35]: X_centered.mean(0)
```

```
Out[35]: array([ 1.11022302e-17,  4.44089210e-17, -1.11022302e-17])
```

To within machine precision, the mean is now zero.

Plotting a two-dimensional function

One place that broadcasting is very useful is in displaying images based on two-dimensional functions. If we want to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid:

```
In [36]: # x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]
print(x.shape, y.shape)
```

```
(50,) (50, 1)
```

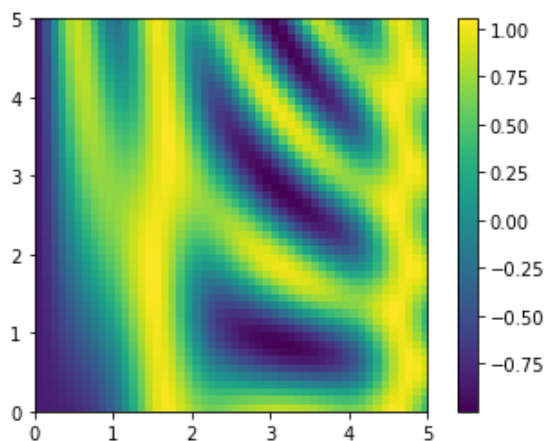
```
In [37]: z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
z.shape
```

```
Out[37]: (50, 50)
```

We'll use Matplotlib to plot this two-dimensional array :

```
In [38]: %matplotlib inline
import matplotlib.pyplot as plt
```

```
In [39]: plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
                  cmap='viridis')
plt.colorbar();
```



The result is a compelling visualization of the two-dimensional function.

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.

Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city. For example, here we'll load the daily rainfall statistics for the city of Seattle in 2014, using Pandas:

```
In [1]: import pandas as pd

# use pandas to extract rainfall inches as a NumPy array
rainfall = pd.read_csv('Seattle2014.csv')['PRCP'].values
inches = rainfall / 25.4 # 25.4mm -> inches
inches.shape
```

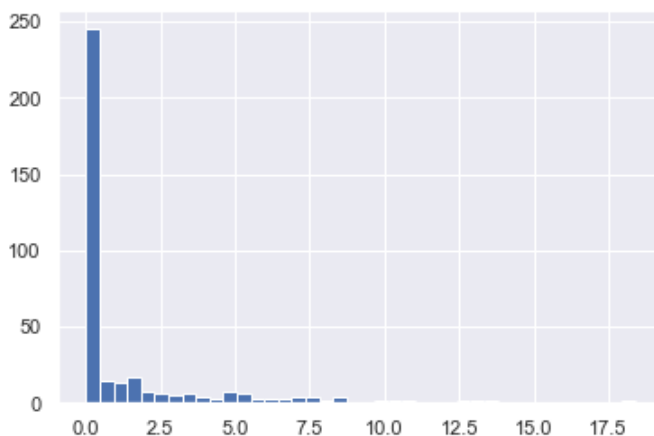
```
Out[1]: (365,)
```

The array contains 365 values, giving daily rainfall in inches from January 1 to December 31, 2014.

As a first quick visualization, let's look at the histogram of rainy days, which was generated using Matplotlib:

```
In [41]: %matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot styles

plt.hist(inches, 40);
```



This histogram gives us a general idea of what the data looks like: despite its reputation, the vast majority of days in Seattle saw near zero measured rainfall in 2014. But this doesn't do a good job of conveying some information we'd like to see: for example, how many rainy days were there in the year? What is the average precipitation on those rainy days? How many days were there with more than half an inch of rain?

Digging into the data

One approach to this would be to answer these questions by hand: loop through the data, incrementing a counter each time we see values in some desired range. For reasons discussed throughout this chapter, such an approach is very inefficient, both from the standpoint of time writing code and time computing the result. We saw previously that NumPy's ufuncs can be used in place of loops to do fast element-wise arithmetic operations on arrays; in the same way, we can use other ufuncs to do element-wise *comparisons* over arrays, and we can then manipulate the results to answer the questions we have. We'll leave the data aside for right now, and discuss some general tools in NumPy to use *masking* to quickly answer these types of questions.

Comparison Operators as ufuncs

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:

```
In [42]: x = np.array([1, 2, 3, 4, 5])

print(x < 3)  # less than
print(x > 3)  # greater than
print(x <= 3) # less than or equal
print(x >= 3) # greater than or equal
print(x != 3) # not equal
print(x == 3) # equal

# It is also possible to do an element-wise comparison of two arrays
# and to include compound expressions
print((2 * x) == (x ** 2))

[ True  True False False False]
[False False False  True  True]
[ True  True  True False False]
[False False  True  True  True]
[ True  True False  True  True]
[False False  True False False]
[False  True False False False]
```

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)`.

```
In [43]: np.less(x, 3)

Out[43]: array([ True,  True, False, False, False])
```

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```
In [44]: rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
x < 6

[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]

Out[44]: array([[ True,  True,  True,  True],
 [False, False,  True,  True],
 [ True,  True, False, False]])
```

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.

```
In [45]: print(x)

[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
```

Counting entries

To count the number of `True` entries in a Boolean array, `np.count_nonzero` is useful:

```
In [46]: # how many values less than 6?
np.count_nonzero(x < 6)

Out[46]: 8
```

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`:

```
In [47]: np.sum(x < 6)

Out[47]: 8
```

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

```
In [48]: # how many values less than 6 in each row?
np.sum(x < 6, axis=1)

Out[48]: array([4, 2, 2])
```

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`:

```
In [49]: # are there any values greater than 8?
print(np.any(x > 8))
# are there any values less than zero?
print(np.any(x < 0))
# are all values less than 10?
print(np.all(x < 10))
# are all values equal to 6?
print(np.all(x == 6))
```

```
True
False
True
False
```

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

```
In [50]: # can specify the axis along which a logical AND reduction is performed.
# are all values in each row less than 8?
np.all(x < 8, axis=1)
# np.all?
```

```
Out[50]: array([ True, False,  True])
```

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

Finally, a quick warning, Python has built-in `sum()`, `any()`, and `all()` functions. These have a different syntax than the NumPy versions, 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

We've already seen how we might count, say, all days with rain less than four inches, or all days with rain greater than two inches. But what if we want to know about all days with rain less than four inches and greater than one inch? This is accomplished through Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`. Like with the standard arithmetic operators, NumPy overloads these as ufuncs which work element-wise on (usually Boolean) arrays.

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

```
In [51]: # print((inches > 0.5) & (inches < 1))
np.sum((inches > 0.5) & (inches < 1)) # true = 1; false = 0
```

```
Out[51]: 16
```

So we see that there are 29 days with rainfall between 0.5 and 1.0 inches.

Note that the parentheses here are important—because of operator precedence rules, with parentheses removed this expression would be evaluated as follows, which results in an error:

```
inches > (0.5 & inches) < 1
```

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

```
In [52]: np.sum(~( (inches <= 0.5) | (inches >= 1) ))
```

```
Out[52]: 16
```

Combining comparison operators and Boolean operators on arrays can lead to a wide range of efficient logical operations.

The following table summarizes the bitwise Boolean operators and their equivalent ufuncs:

Using these tools, we might start to answer the types of questions we have about our weather data. Here are some examples of results we can compute when combining masking with aggregations:

```
In [53]: print("Number days without rain:      ", np.sum(inches == 0))
        print("Number days with rain:        ", np.sum(inches != 0))
        print("Days with more than 0.5 inches:", np.sum(inches > 0.5))
        print("Rainy days with < 0.2 inches  :", np.sum((inches > 0) & (inches < 0.2)))
```

Number days without rain: 215
Number days with rain: 150
Days with more than 0.5 inches: 120
Rainy days with < 0.2 inches : 23

Boolean Arrays as Masks

In the preceding section we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our `x` array from before, suppose we want an array of all values in the array that are less than, say, 5:

We can obtain a Boolean array for this condition easily, as we've already seen:

```
In [54]: print(x)
        x < 5
```

[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]

```
Out[54]: array([[False,  True,  True,  True],
               [False, False,  True, False],
               [ True,  True, False, False]])
```

Now to *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:

```
In [55]: x[x < 5]
```

```
Out[55]: array([0, 3, 3, 3, 2, 4])
```

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is `True`.

We are then free to operate on these values as we wish. For example, we can compute some relevant statistics on our Seattle rain data:

```
In [56]: # construct a mask of all rainy days
rainy = (inches > 0)

# construct a mask of all summer days (June 21st is the 172nd day)
days = np.arange(365)
summer = (days > 172) & (days < 262)

print("Median precip on rainy days in 2014 (inches): ",
      np.median(inches[rainy]))
print("Median precip on summer days in 2014 (inches): ",
      np.median(inches[summer]))
print("Maximum precip on summer days in 2014 (inches): ",
      np.max(inches[summer]))
print("Median precip on non-summer rainy days (inches):",
      np.median(inches[rainy & ~summer]))
```

```
Median precip on rainy days in 2014 (inches): 1.9488188976377954
Median precip on summer days in 2014 (inches): 0.0
Maximum precip on summer days in 2014 (inches): 8.503937007874017
Median precip on non-summer rainy days (inches): 2.0078740157480315
```

By combining Boolean operations, masking operations, and aggregates, we can very quickly answer these sorts of questions for our dataset.

Aside: Using the Keywords and/or Versus the Operators &|

One common point of confusion is the difference between the keywords `and` and `or` on one hand, and the operators `&` and `|` on the other hand. When would you use one versus the other?

The difference is this: `and` and `or` gauge the truth or falsehood of *entire object*, while `&` and `|` refer to *bits within each object*.

When you use `and` or `or`, it's equivalent to asking Python to treat the object as a single Boolean entity. In Python, all nonzero integers will evaluate as `True`. Thus:

```
In [57]: bool(42), bool(0)
```

```
Out[57]: (True, False)
```

```
In [58]: bool(42 and 0), bool(42 or 0)
```

```
Out[58]: (False, True)
```

When you use `&` and `|` on integers, the expression operates on the bits of the element, applying the *and* or the *or* to the individual bits making up the number:

```
In [59]: bin(42), bin(59)
```

```
Out[59]: ('0b101010', '0b111011')
```

```
In [60]: bin(42 & 59), bin(42 | 59)
```

```
Out[60]: ('0b101010', '0b111011')
```

Notice that the corresponding bits of the binary representation are compared in order to yield the result.

When you have an array of Boolean values in NumPy, this can be thought of as a string of bits where `1 = True` and `0 = False`, and the result of `&` and `|` operates similarly to above:


```
In [61]: A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
        B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
        A | B
```

```
Out[61]: array([ True,  True,  True, False,  True,  True])
```

Using `or` on these arrays will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
In [62]: A or B
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-62-ea2c97d9d9ee> in <module>
----> 1 A or B

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any()
or a.all()
```

Similarly, when doing a Boolean expression on a given array, you should use `|` or `&` rather than `or` or `and`:

```
In [63]: x = np.arange(10)
        (x > 4) & (x < 8)
```

```
Out[63]: array([False, False, False, False, False,  True,  True,  True, False,
                False])
```

So remember this: `and` and `or` perform a single Boolean evaluation on an entire object, while `&` and `|` perform multiple Boolean evaluations on the content (the individual bits or bytes) of an object. For Boolean NumPy arrays, the latter is nearly always the desired operation.

Fancy Indexing

In the previous sections, we saw how to access and modify portions of arrays using simple indices (e.g., `arr[0]`), slices (e.g., `arr[:5]`), and Boolean masks (e.g., `arr[arr > 0]`). In this section, we'll look at another style of array indexing, known as *fancy indexing*. Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

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:

```
In [64]: np.random.seed(0)
        x = np.random.randint(100, size=10)
        print(x)
```

```
[44 47 64 67 67  9 83 21 36 87]
```

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

```
In [65]: [x[3], x[7], x[2]]
```

```
Out[65]: [67, 21, 64]
```

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

```
In [66]: ind = [3, 7, 4]
         x[ind]
```

```
Out[66]: array([67, 21, 67])
```

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

```
In [67]: ind = np.array([[3, 7],
                        [4, 5]])
         x[ind]
```

```
Out[67]: array([[67, 21],
                [67,  9]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

```
In [68]: X = np.arange(12).reshape((3, 4))
         X
```

```
Out[68]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

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

```
In [69]: row = np.array([0, 1, 2])
         col = np.array([2, 1, 3])
         X[row, col]
```

```
Out[69]: array([ 2,  5, 11])
```

Notice that the first value in the result is `X[0, 2]`, the second is `X[1, 1]`, and the third is `X[2, 3]`.

So, for example, if we combine a column vector and a row vector within the indices, we get a two-dimensional result:

```
In [70]: X[row[:, np.newaxis], col] # after broad casting, the shape is 3*3
```

```
Out[70]: array([[ 2,  1,  3],
                [ 6,  5,  7],
                [10,  9, 11]])
```

It is always important to remember with fancy indexing that the return value reflects the *broadcasted shape of the indices*, rather than the shape of the array being indexed.

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen.

We can combine fancy and simple indices:

```
In [71]: print(X)
          X[2, [2, 0, 1]]

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
Out[71]: array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
In [72]: X[1:, [2, 0, 1]]
```

```
Out[72]: array([[ 6,  4,  5],
                [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
In [73]: mask = np.array([1, 0, 1, 0], dtype=bool)
          X[row[:, np.newaxis], mask] # mask indicate the fancy indexing of [0,2], broadcast shape 3*2
```

```
Out[73]: array([[ 0,  2],
                [ 4,  6],
                [ 8, 10]])
```

All of these indexing options combined lead to a very flexible set of operations for accessing and modifying array values.

Example: Selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. For example, we might have an N by D matrix representing N points in D dimensions, such as the following points drawn from a two-dimensional normal distribution:

```
In [74]: mean = [0, 0]
          cov = [[1, 2],
                 [2, 5]]
          X = np.random.multivariate_normal(mean, cov, 100)
          X.shape
```

```
Out[74]: (100, 2)
```

Using the plotting tools , we can visualize these points as a scatter-plot:

```
In [75]: %matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # for plot styling

plt.scatter(X[:, 0], X[:, 1]);
```



Let's use fancy indexing to select 20 random points. We'll do this by first choosing 20 random indices with no repeats, and use these indices to select a portion of the original array:

```
In [76]: indices = np.random.choice(X.shape[0], 20, replace=False)
indices
```

```
Out[76]: array([15, 89, 28, 69, 87, 14, 32, 83, 43, 21, 82, 34, 63, 78, 97,  5, 74,
          96, 76, 52])
```

```
In [77]: selection = X[indices] # fancy indexing here
selection.shape
```

```
Out[77]: (20, 2)
```

Now to see which points were selected, let's over-plot large circles at the locations of the selected points:

```
In [78]: plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
plt.scatter(selection[:, 0], selection[:, 1],
            facecolor='r', s=50);
```



This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models, and in sampling approaches to answering statistical questions.

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:

```
In [79]: x = np.arange(10)
         i = np.array([2, 1, 8, 4])
         x[i] = 99
         print(x)

[ 0 99 99  3 99  5  6  7 99  9]
```

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

```
In [80]: x[i] -= 10
         print(x)

[ 0 89 89  3 89  5  6  7 89  9]
```

Notice, though, that repeated indices with these operations can cause some potentially unexpected results. Consider the following:

```
In [83]: x = np.zeros(10)
         x[[0, 0]] = [4, 6]
         print(x)

[6.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Where did the 4 go? The result of this operation is to first assign $x[0] = 4$, followed by $x[0] = 6$. The result, of course, is that $x[0]$ contains the value 6.

Fair enough, but consider this operation:

```
In [84]: i = [2, 3, 3, 4, 4, 4]
         x[i] += 1
         x

Out[84]: array([6.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
```

You might expect that $x[3]$ would contain the value 2, and $x[4]$ would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because $x[i] += 1$ is meant as a shorthand of $x[i] = x[i] + 1$. $x[i] + 1$ is evaluated, and then the result is assigned to the indices in x . With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather nonintuitive results.

So what if you want the other behavior where the operation is repeated? For this, you can use the `at()` method of ufuncs (available since NumPy 1.8), and do the following:

```
In [81]: x = np.zeros(10)
         np.add.at(x, i, 1)
         print(x)

[0.  1.  1.  0.  1.  0.  0.  0.  1.  0.]
```

The `at()` method does an in-place application of the given operator at the specified indices (here, `i`) with the specified value (here, 1). Another method that is similar in spirit is the `reduceat()` method of `ufuncs`, which you can read about in the NumPy documentation.

Example: Binning Data

You can use these ideas to efficiently bin data to create a histogram by hand. For example, imagine we have 1,000 values and would like to quickly find where they fall within an array of bins. We could compute it using `ufunc.at` like this:

```
In [82]: np.random.seed(42)
x = np.random.randn(100) # generate 100 numbers from gaussian distribution of mean 0 and variance 1

# compute a histogram by hand
bins = np.linspace(-5, 5, 20)

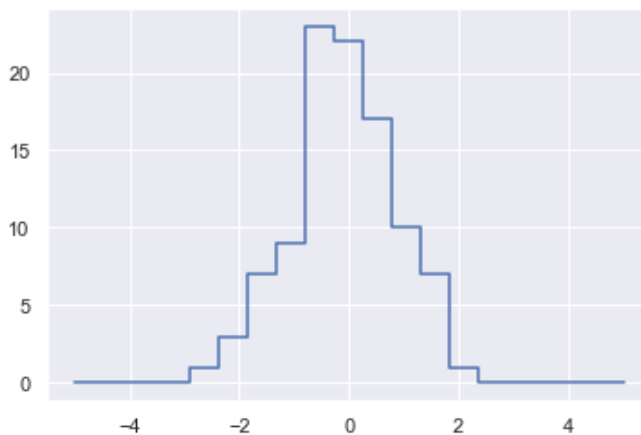
# np.zeros_like return an array of zeros with the same shape and type as a given array.
counts = np.zeros_like(bins)

# find the appropriate bin for each x
# Find indices where elements should be inserted to maintain order.
i = np.searchsorted(bins, x)

# add 1 to each of these bins
np.add.at(counts, i, 1)
```

The counts now reflect the number of points within each bin—in other words, a histogram:

```
In [83]: # plot the results
plt.plot(bins, counts, linestyle='steps');
```



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

```
plt.hist(x, bins, histtype='step');
```

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:

```
In [84]: print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)

print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

NumPy routine:

16.8 μ s \pm 92.5 ns per loop (mean \pm std. dev. of 7 runs, 100000 loops each)

Custom routine:

10.8 μ s \pm 20.4 ns per loop (mean \pm std. dev. of 7 runs, 100000 loops each)

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:

```
In [85]: x = np.random.randn(1000000)
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)

print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

NumPy routine:

51.5 ms \pm 190 μ s per loop (mean \pm std. dev. of 7 runs, 10 loops each)

Custom routine:

86.9 ms \pm 279 μ s per loop (mean \pm std. dev. of 7 runs, 10 loops each)

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. But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, you could use these building blocks to extend this to do some very interesting custom behaviors. The key to efficiently using Python in data-intensive applications is knowing about general convenience routines like `np.histogram` and when they're appropriate, but also knowing how to make use of lower-level functionality when you need more pointed behavior.