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[38]: A or B

---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-38-5d8e4f2e21c0> in <module>()
----> 1 A or B


ValueError: The truth value of an array with more than one element is...
```

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

```
In[39]: x = np.arange(10)
        (x > 4) & (x < 8)

Out[39]: array([False, False, ...,  True,  True, False, False], dtype=bool)
```

Trying to evaluate the truth or falsehood of the entire array will give the same `ValueError` we saw previously:

```
In[40]: (x > 4) and (x < 8)

---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-40-3d24f1ffd63d> in <module>()
----> 1 (x > 4) and (x < 8)


ValueError: The truth value of an array with more than one element is...
```

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[1]: import numpy as np
       rand = np.random.RandomState(42)

       x = rand.randint(100, size=10)
       print(x)

[51 92 14 71 60 20 82 86 74 74]
```

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

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

Out[2]: [71, 86, 14]
```

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

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

Out[3]: array([71, 86, 60])
```

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

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

Out[4]: array([[71, 86],
               [60, 20]])
```

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

```
In[5]: X = np.arange(12).reshape((3, 4))
       X

Out[5]: 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[6]: row = np.array([0, 1, 2])
       col = np.array([2, 1, 3])
       X[row, col]

Out[6]: 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]`. The pairing of indices in fancy indexing follows all the broadcasting rules that were mentioned in "Computation on Arrays: Broadcasting" on page 63. So,