Fancy Indexing

The previous chapters discussed 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 chapter, we'll look at **another style of array indexing,** known as **fancy** or **vectorized** indexing, in which 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 [14]:
```
import numpy as np
rng = np.random.default_rng(seed=1701)

x = rng.integers(100, size=10)
print(x)
```

```
[90 40  9 30 80 67 39 15 33 79]
```

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

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

Out[ ]:
```
[30, 15, 9]
```

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

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

Out[ ]:
```
array([30, 15, 80])
```

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

In [15]:
```python
x
```

Out[15]:
```
array([90, 40,  9, 30, 80, 67, 39, 15, 33, 79])
```

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

```
Out[ ]: array([[30, 15],
               [80, 67]])
```

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]:
```python
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.**

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

In [ ]:
```python
X[row[:, np.newaxis], col]
```

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

Here, **each row value** is **matched** with **each column vector,** exactly as we saw in broadcasting of arithmetic operations.

For example:

```
In [7]: col
```

```
Out[7]: array([2, 1, 3])
```

```
In [8]: row[:, np.newaxis] * col
```

```
Out[8]: array([[0, 0, 0],
               [2, 1, 3],
               [4, 2, 6]])
```

```
In [ ]: row[:, np.newaxis] * col
```

Out[ ]:

```
array([[0, 0, 0],
       [2, 1, 3],
       [4, 2, 6]])
```

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. **For example,** given the array `X`:

In [ ]:

```
print(X)
```

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

We can **combine fancy** and **simple indices:**

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

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

We can also **combine fancy indexing** with **slicing:**

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

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

And we can **combine fancy indexing** with **masking:**

In [ ]:
```python
mask = np.array([True, False, True, False])
X[row[:, np.newaxis], mask]
```

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

All of these **indexing options** combined lead to a very **flexible set of operations** for **efficiently 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 [9]:
```
mean = [0, 0]
cov = [[1, 2],
       [2, 5]]
X = rng.multivariate_normal(mean, cov, 100)
X.shape
```

Out[9]:
```
(100, 2)
```

In [10]:
```
X
```

```
Out[10]:  array([[ 9.23761718e-01,  3.86247383e+00],
                 [ 6.61751529e-01,  2.92566223e+00],
                 [ 7.30981058e-01,  2.44134086e+00],
                 [-3.29786797e-01, -1.16688738e-01],
                 [-3.08887585e-01,  8.74389601e-01],
                 [ 8.60135824e-01,  1.64977873e+00],
                 [ 1.12535005e+00,  3.07970690e+00],
                 [-1.15071385e+00, -1.02441057e+00],
                 [ 8.52757448e-02, -9.07980378e-01],
                 [ 7.24091502e-01,  2.16337705e+00],
                 [-7.55750651e-01, -3.51313567e-01],
                 [-1.13766512e+00, -2.80758238e+00],
                 [ 1.51379786e-01, -1.23549790e+00],
                 [-2.19302975e+00, -5.36976879e+00],
                 [ 6.70249139e-01,  3.71399758e+00],
                 [ 1.45648064e+00,  2.74167311e+00],
                 [-8.50652055e-01, -1.54100459e+00],
                 [ 2.05554772e-01,  7.80629399e-01],
                 [-1.62415980e+00, -3.90121516e+00],
                 [ 8.04554777e-01,  3.42702891e+00],
                 [-8.31136835e-01, -1.42428390e+00],
```

```
[-1.52519971e+00, -3.37676226e+00],
[-3.75641948e-01, -1.31595855e+00],
[ 7.00500528e-01,  2.73668582e+00],
[ 8.20900913e-01,  5.90557023e-01],
[-3.41910865e-01, -3.74276386e-01],
[ 5.91064558e-01,  2.24773885e+00],
[ 2.37187534e+00,  4.46843628e+00],
[ 1.11908814e-01, -9.40897295e-01],
[ 7.91465930e-01,  2.64057061e+00],
[-1.42326788e+00, -2.72306289e+00],
[-5.91795754e-01,  1.91761015e+00],
[-7.10588039e-01, -1.05392878e+00],
[-1.00316680e-01,  8.65958304e-02],
[-9.50621459e-01, -8.69559492e-01],
[ 5.50992347e-01, -5.30277511e-01],
[ 7.46830441e-01,  2.26487629e+00],
[ 3.89223439e-02, -1.43137961e+00],
[ 4.92901753e-01,  7.96397066e-01],
[-8.55749544e-01, -9.99618236e-01],
[ 7.19488278e-02,  7.02146349e-01],
[-1.99932580e+00, -2.27811191e+00],
```

```
[ 1.86517951e-01,  1.23596368e+00],
[-8.09056584e-01, -1.81703893e+00],
[ 8.29265444e-01,  2.18345845e+00],
[ 5.35190376e-01,  4.47966342e-01],
[-3.27562105e-01, -1.77482480e+00],
[-2.69738048e+00, -5.62348677e+00],
[ 1.43891059e+00,  2.42258911e+00],
[ 4.70562172e-01,  2.52786853e-01],
[-5.34031981e-01, -5.85417619e-01],
[-1.63955400e+00, -4.54577650e+00],
[ 8.69157425e-01,  1.05136337e+00],
[-2.48588812e+00, -5.42296100e+00],
[ 7.94899718e-01,  2.13159391e+00],
[-4.36407775e-01,  9.33100970e-01],
[ 2.79594472e+00,  6.33408633e+00],
[-1.62721133e+00, -3.29227540e+00],
[-1.59944090e+00, -3.74038295e+00],
[-2.59751447e+00, -3.47995836e+00],
[-1.41393471e+00, -3.66357845e+00],
[ 2.32369408e-01,  8.80859656e-01],
[ 3.40511819e-02,  7.32541387e-01],
```

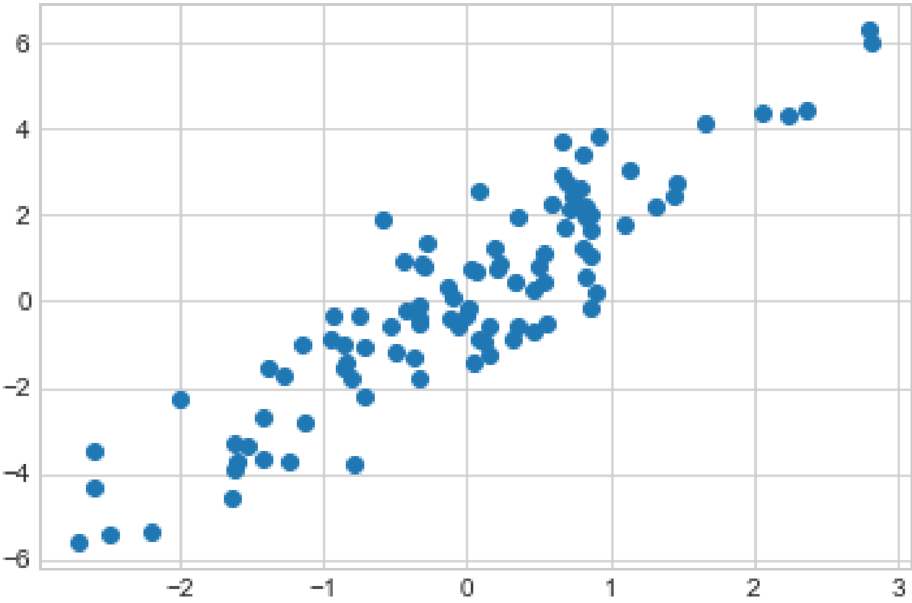
```
[ 2.23519164e+00,  4.33266160e+00],
[-3.31214432e-01, -5.39959415e-01],
[ 5.33618503e-01,  1.11631280e+00],
[-1.32861435e-01,  3.02343489e-01],
[-2.90496991e-01,  8.08089066e-01],
[ 8.55570565e-01, -1.29022391e-01],
[ 6.88371652e-01,  1.71544376e+00],
[-7.19572387e-01, -2.20706627e+00],
[ 3.41534033e-01,  4.35054321e-01],
[ 8.59854946e-01,  2.03652405e+00],
[ 8.17369954e-01,  1.94401318e+00],
[ 1.63466107e-01, -5.67376304e-01],
[ 3.27979510e-01, -9.04223047e-01],
[ 1.09101199e+00,  1.77260811e+00],
[-7.93948859e-01, -3.81180928e+00],
[-1.26787248e+00, -1.74778269e+00],
[-1.37619004e+00, -1.55017467e+00],
[ 2.81388613e+00,  6.03151912e+00],
[ 9.05995326e-02,  2.56736581e+00],
[ 8.90561967e-01,  2.02812028e-01],
[ 2.05045627e+00,  4.39542104e+00],
```

```
[ 1.65070370e+00,  4.15011809e+00],
[-2.57288282e-03, -3.39005109e-01],
[-9.32267600e-01, -3.39969892e-01],
[-4.97766658e-01, -1.17936925e+00],
[-2.83490787e-01,  1.33429965e+00],
[ 3.58560517e-01, -5.61047342e-01],
[ 8.03032647e-01,  1.20986161e+00],
[-4.26502973e-01, -2.22030027e-01],
[ 3.48464179e-01,  1.95705366e+00],
[ 4.66059518e-01, -6.79589356e-01],
[-1.23294063e-01, -4.23105904e-01],
[-1.23663760e+00, -3.70722817e+00],
[-2.59588094e+00, -4.33224979e+00],
[-5.23767992e-02, -5.80675479e-01],
[ 1.31411717e+00,  2.22815311e+00],
[ 1.08304664e-02, -1.24518261e-01]])
```

Using the plotting tools we will discuss in **Introduction to Matplotlib,** we can visualize these points as a **scatter plot:**

In [ ]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')

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 **using these indices** to **select a portion** of the original array:

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

Out[ ]:
```
array([82, 84, 10, 55, 14, 33,  4, 16, 34, 92, 99, 64,  8, 7
6, 68, 18, 59,
       80, 87, 90])
```

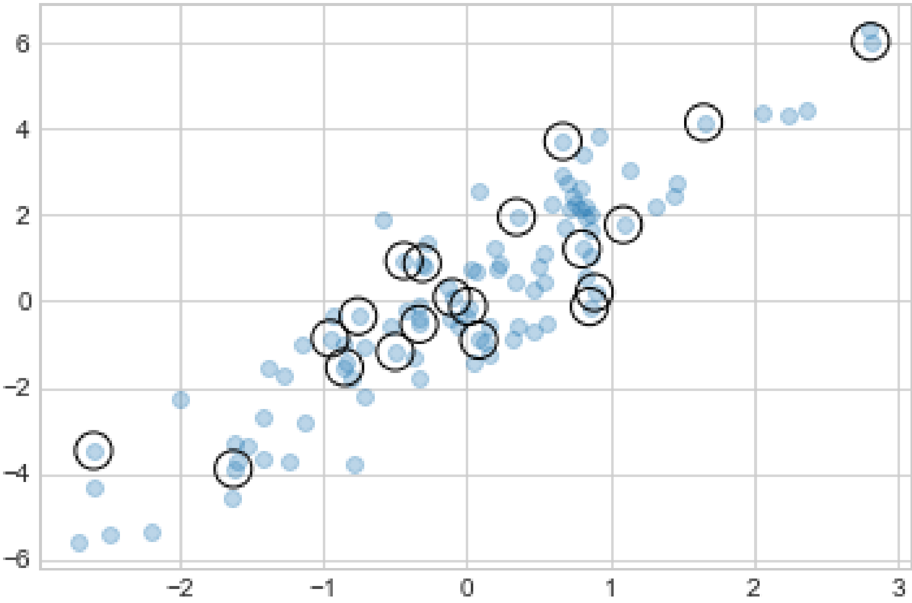
In [ ]:
```
selection = X[indices]  # fancy indexing here
selection.shape
```

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

Now to see **which points were selected.**

Let's **overplot large circles** at the **locations of the selected points:**

In [ ]:
```
plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
plt.scatter(selection[:, 0], selection[:, 1],
            facecolor='none', edgecolor='black', s=200);
```



This sort of **strategy** is often used to **quickly partition datasets.**

As is often needed in train/test splitting for validation of statistical models (see **Hyperparameters and Model Validation**).

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 [ ]:
```
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 [ ]: 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:**

```
In [16]: x = np.zeros(10)
         x
```

```
Out[16]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [18]:
```
ind = [0, 0]
```

In [19]:
```
x[ind]
```

Out[19]:
```
array([0., 0.])
```

In [21]:
```
x[ind] = [4, 6]
print(x)
```

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

Where did the 4 go? This operation first assigns `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 [22]:
```
x
```

```
Out[22]: array([6., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
In [24]: i = [2, 3, 3, 4, 4, 4]
         x[i] += 1
         x
```

```
Out[24]: 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 and do the following:

In [25]:
```
i
```

Out[25]:
```
[2, 3, 3, 4, 4, 4]
```

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

```
[0. 0. 1. 2. 3. 0. 0. 0. 0. 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 could use these **ideas** to efficiently do **custom binned computations** on data.

For example, imagine we have 100 values and would like to quickly find where they fall within an array of bins.

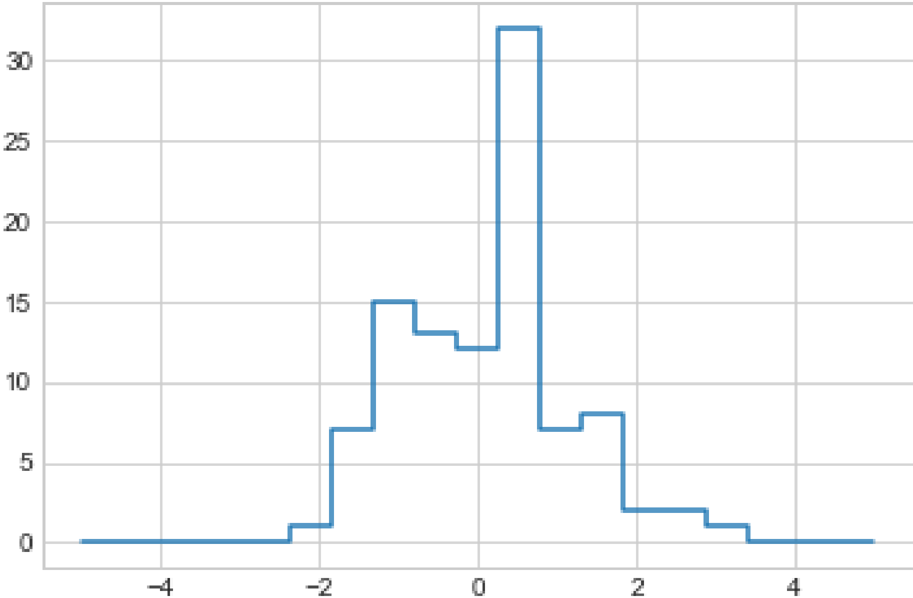
We could compute this using `ufunc.at` like this:

In [ ]:
```python
rng = np.random.default_rng(seed=1701)
x = rng.normal(size=100)

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

# find the appropriate bin for each x
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 [ ]:
```python
# plot the results
```

```
plt.plot(bins, counts, drawstyle='steps');
```



Of course, it would be **inconvenient** 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 just shown.

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 [ ]:

```
print(f"NumPy histogram ({len(x)} points):")
%timeit counts, edges = np.histogram(x, bins)

print(f"Custom histogram ({len(x)} points):")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy histogram (100 points):
33.8 µs ± 311 ns per loop (mean ± std. dev. of 7 runs, 10000 lo
ops each)
Custom histogram (100 points):
17.6 µs ± 113 ns per loop (mean ± std. dev. of 7 runs, 100000 l
oops each)
```

Our **own one-line algorithm** is **twice as fast** as 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 [ ]:
```
x = rng.normal(size=1000000)
print(f"NumPy histogram ({len(x)} points):")
%timeit counts, edges = np.histogram(x, bins)

print(f"Custom histogram ({len(x)} points):")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy histogram (1000000 points):
84.4 ms ± 2.82 ms per loop (mean ± std. dev. of 7 runs, 10 loop
s each)
Custom histogram (1000000 points):
128 ms ± 2.04 ms per loop (mean ± 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** (see Big-O Notation).

But the **advantage of coding** this algorithm yourself is that with an **understanding of these basic methods:**

You're **no longer constrained to built-in routines,** but can create your **own approaches** to exploring the data.

Key to efficiently using Python in data-intensive applications is not only 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.