Advanced indexing & Broadcasting

Lecture 06

Dr. Colin Rundel

Advanced Indexing

Advanced Indexing

> Advanced indexing is triggered when the selection object, obj, is a non-tuple sequence object, an ndarray (of data type integer or bool), or a tuple with at least one sequence object or ndarray (of data type integer or bool).
>
> - There are two types of advanced indexing: integer and Boolean.
> - Advanced indexing always returns a *copy* of the data (contrast with basic slicing that returns a view).

From NumPy - User Guide - Advanced Indexing

From last time: subsetting with tuples

Unlike lists, an ndarray can be subset by a tuple containing integers,

```
x = np.arange(16).reshape((4,4)); x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
x[(0,1,3), :]
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

```
x[:, (0,1,3)]
```

```
array([[ 0,  1,  3],
       [ 4,  5,  7],
       [ 8,  9, 11],
       [12, 13, 15]])
```

```
np.shares_memory(x, x[(0,1,3), :])
```

```
False
```

```
x[(0,1,3),]
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

```
x[(0,1,3)]
```

```
IndexError: too many indices for array: array is 2-dimensional, but 3 were indexed
```

```
x[(0,1)]
```

```
1
```

```
x[0,1]
```

```
1
```

Integer array subsetting (lists)

Lists of integers can be used to subset in the same way:

```
x = np.arange(16).reshape((4,4)); x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
x[[1,3]]
```

```
array([[ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

```
x[[1,3], ]
```

```
array([[ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

```
x[:, [1,3]]
```

```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11],
       [13, 15]])
```

```
x[[0,1,3],]
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

```
x[[0,1,3]]
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

```
x[[1.,3]]
```

```
IndexError: only integers, slices (`:`), ellipsis
(`...`), numpy.newaxis (`None`) and integer or
boolean arrays are valid indices
```

Integer array subsetting (ndarrays)

Similarly we can also us integer ndarrays:

```python
x = np.arange(6)
y = np.array([0,1,3])
z = np.array([1., 3.])
```

```python
x[y,]
```

```
array([0, 1, 3])
```

```python
x[y]
```

```
array([0, 1, 3])
```

```python
x[z]
```

```
IndexError: arrays used as indices must be of
integer (or boolean) type
```

```python
x = np.arange(16).reshape((4,4))
y = np.array([1,3])
```

```python
x[y]
```

```
array([[ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

```python
x[y, ]
```

```
array([[ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

```python
x[:, y]
```

```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11],
       [13, 15]])
```

```python
x[y, y]
```

```
array([ 5, 15])
```

Exercise 1

Given the following matrix,

```
x = np.arange(16).reshape((4,4))
x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

write an expression to obtain the center 2x2 values (i.e. 5, 6, 9, 10 as a matrix).

Boolean indexing

Lists or ndarrays of boolean values can also be used to subset (positions with `True` are kept and `False` are discarded)

```
x = np.arange(6); x
```

```
array([0, 1, 2, 3, 4, 5])
```

```
x[[True, False, True, False, True, False]]
```

```
array([0, 2, 4])
```

```
x[[True]]
```

```
IndexError: boolean index did not match indexed array along dimension 0; dimension is 6 but corresponding boolean dimension is 1
```

```
x[np.array([True, True, False, False, True, False])]
```

```
array([0, 1, 4])
```

```
x[np.array([True])]
```

```
IndexError: boolean index did not match indexed array along dimension 0; dimension is 6 but corresponding boolean dimension is 1
```

Boolean expressions

The primary utility of boolean subsetting comes from vectorized comparison operations,

```
x = np.arange(6); x
```

```
array([0, 1, 2, 3, 4, 5])
```

```
x > 3
```

```
array([False, False, False, False,  True,  True])
```

```
x[x>3]
```

```
array([4, 5])
```

```
x % 2 == 1
```

```
array([False,  True, False,  True, False,  True])
```

```
x[x % 2 == 1]
```

```
array([1, 3, 5])
```

```
y = np.arange(9).reshape((3,3))
y % 2 == 0
```

```
array([[ True, False,  True],
       [False,  True, False],
       [ True, False,  True]])
```

```
y[y % 2 == 0]
```

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

NumPy and Boolean operators

If we want to use a logical operators on an array we need to use `&`, `|`, and `~` instead of `and`, `or`, and `not` respectively.

```
x = np.arange(6); x
```

```
array([0, 1, 2, 3, 4, 5])
```

```
y = x % 2 == 0; y
```

```
array([ True, False,  True, False,  True, False])
```

```
~y
```

```
array([False,  True, False,  True, False,  True])
```

```
y & (x > 3)
```

```
array([False, False, False, False,  True, False])
```

```
y | (x > 3)
```

```
array([ True, False,  True, False,  True,  True])
```

meshgrid()

One other useful function in NumPy is `meshgrid()` which generates all possible combinations between the input vectors (as a tuple of ndarrays),

```
pts = np.arange(3)
x, y = np.meshgrid(pts, pts)
x
```

```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

```
y
```

```
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])
```

```
np.sqrt(x**2 + y**2)
```

```
array([[0.        , 1.        , 2.        ],
       [1.        , 1.41421356, 2.23606798],
       [2.        , 2.23606798, 2.82842712]])
```

Exercise 2

We will now use this to attempt a simple brute force approach to numerical optimization, define a grid of points using `meshgrid()` to approximate the minima the following function:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

Considering values of $x, y \in (-1, 3)$, which value(s) of x, y minimize this function?

Broadcasting

Broadcasting

This is an approach for deciding how to generalize arithmetic operations between arrays with differing shapes.

```
x = np.array([1, 2, 3])
x * 2
```

```
array([2, 4, 6])
```

```
x * np.array([2])
```

```
array([2, 4, 6])
```

```
x * np.array([2,2,2])
```

```
array([2, 4, 6])
```

In the first example `2` is equivalent to the array `np.array([2])` which is being broadcast across the longer array `x`.

Efficiancy

Using broadcasts can be much more efficient as it does not copy the underlying data,

```
x = np.arange(1e5)
y = np.array([2]).repeat(1e5)
```

```
%timeit x * 2
17.3 μs ± 101 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit x * np.array([2])
17.2 μs ± 239 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit x * y
36 μs ± 121 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

General Broadcasting

> When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when
>
> 1. they are equal, or
> 2. one of them is 1
>
> If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Example

Why does the code on the left work but not the code on the right?

```
x = np.arange(12).reshape((4,3))
x
```

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

```
x + np.array([1,2,3])
```

```
array([[ 1,  3,  5],
       [ 4,  6,  8],
       [ 7,  9, 11],
       [10, 12, 14]])
```

```
x    (2d array): 4 x 3
y    (1d array):     3
----------------------
x+y  (2d array): 4 x 3
```

```
x = np.arange(12).reshape((3,4))
x
```

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

```
x + np.array([1,2,3])
```

```
ValueError: operands could not be broadcast
together with shapes (3,4) (3,)
```

```
x    (2d array): 3 x 4
y    (1d array):     3
----------------------
x+y  (2d array): Error
```

A quick fix

```python
x = np.arange(12).reshape((3,4)); x
```

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

```python
x + np.array([1,2,3]).reshape(3,1)
```

```
array([[ 1,  2,  3,  4],
       [ 6,  7,  8,  9],
       [11, 12, 13, 14]])
```

```
x    (2d array): 3 x 4
y    (2d array): 3 x 1
----------------------
x+y  (2d array): 3 x 4
```

Examples (2)

```
x = np.arange(12).reshape((4,3))
y = 1
x+y
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
x     (2d array): 4 x 3
y     (1d array):     1
----------------------
x+y   (2d array): 4 x 3
```

```
x = np.arange(12).reshape((4,3))
y = np.array([1,2,3])
x+y
```

```
array([[ 1,  3,  5],
       [ 4,  6,  8],
       [ 7,  9, 11],
       [10, 12, 14]])
```

```
x     (2d array): 4 x 3
y     (1d array):     3
----------------------
x+y   (2d array): 4 x 3
```

Examples (3)

```
x = np.array([0,10,20,30]).reshape((4,1))
y = np.array([1,2,3])
```

```
x
```

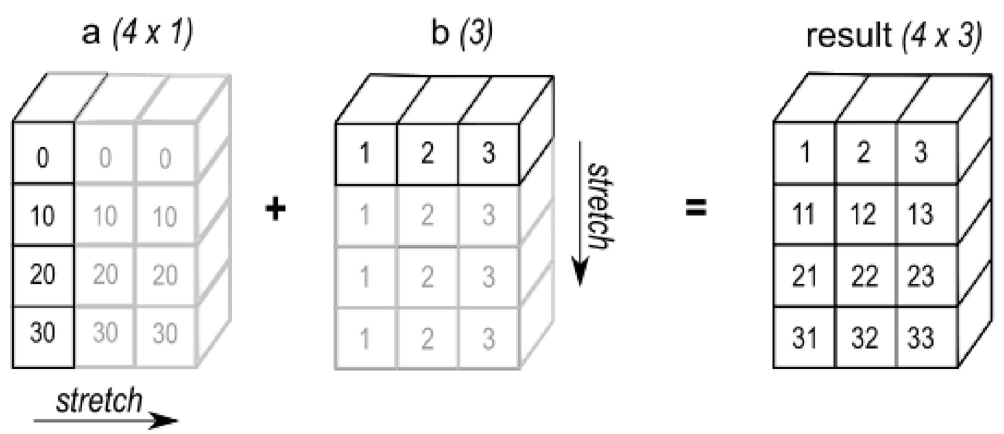
```
array([[ 0],
       [10],
       [20],
       [30]])
```

```
y
```

```
array([1, 2, 3])
```

```
x+y
```

```
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```



From NumPy user guide - Broadcasting

Exercise 3

For each of the following combinations determine what the resulting dimension will be using broadcasting

- A [128 x 128 x 3] + B [3]
- A [8 x 1 x 6 x 1] + B [7 x 1 x 5]
- A [2 x 1] + B [8 x 4 x 3]
- A [3 x 1] + B [15 x 3 x 5]
- A [3] + B [4]

Demo 1 - Standardization

Below we generate a data set with 3 columns of random normal values. Each column has a different mean and standard deviation which we can check with `mean()` and `std()`.

```
rng = np.random.default_rng(1234)
d = rng.normal(loc=[-1,0,1], scale=[1,2,3], size=(1000,3))
```

```
d.mean(axis=0)
```

```
array([-1.0294382 , -0.01396257,  1.01241784])
```

```
d.std(axis=0)
```

```
array([0.99674719, 2.03222595, 3.10625219])
```

Use broadcasting to standardize all three columns to have mean 0 and standard deviation 1.

Check the new data set using `mean()` and `std()`.

Broadcasting and assignment

In addition to arithmetic operators, broadcasting can be used with assignment via array indexing,

```
x = np.arange(12).reshape((3,4))
y = -np.arange(4)
z = -np.arange(3)
```

```
x[:] = y
x
```

```
array([[ 0, -1, -2, -3],
       [ 0, -1, -2, -3],
       [ 0, -1, -2, -3]])
```

```
x[...] = y
x
```

```
array([[ 0, -1, -2, -3],
       [ 0, -1, -2, -3],
       [ 0, -1, -2, -3]])
```

```
x[:] = z
```

```
ValueError: could not broadcast input
array from shape (3,) into shape (3,4)
```

```
x[:] = z.reshape((3,1))
x
```

```
array([[ 0,  0,  0,  0],
       [-1, -1, -1, -1],
       [-2, -2, -2, -2]])
```

Basic file IO

Reading and writing ndarrays

We will not spend much time on this as most data you will encounter is more likely to be a tabular format (e.g. data frame) and tools like Pandas are more appropriate.

For basic saving and loading of NumPy arrays there are the `save()` and `load()` functions which use a built in binary format.

```
x = np.arange(1e5)

np.save("data/x.npy", x)

new_x = np.load("data/x.npy")

np.all(x == new_x)
```

```
True
```

Additional functions for saving (`savez()`, `savez_compressed()`, `savetxt()`) exist for saving multiple arrays or saving a text representation of an array.

Reading delimited data

While not particularly recommended, if you need to read delimited (csv, tsv, etc.) data into a NumPy array you can use `genfromtxt()`,

```
with open("data/mtcars.csv") as file:
    mtcars = np.genfromtxt(file, delimiter=",", skip_header=True)

mtcars
```

```
array([[  6.   , 160.   , 110.   ,   3.9  ,   2.62 ,  16.46 ,   0.   ,   1.   ,   4.   ,   4.   ],
       [  6.   , 160.   , 110.   ,   3.9  ,   2.875,  17.02 ,   0.   ,   1.   ,   4.   ,   4.   ],
       [  4.   , 108.   ,  93.   ,   3.85 ,   2.32 ,  18.61 ,   1.   ,   1.   ,   4.   ,   1.   ],
       [  6.   , 258.   , 110.   ,   3.08 ,   3.215,  19.44 ,   1.   ,   0.   ,   3.   ,   1.   ],
       [  8.   , 360.   , 175.   ,   3.15 ,   3.44 ,  17.02 ,   0.   ,   0.   ,   3.   ,   2.   ],
       [  6.   , 225.   , 105.   ,   2.76 ,   3.46 ,  20.22 ,   1.   ,   0.   ,   3.   ,   1.   ],
       [  8.   , 360.   , 245.   ,   3.21 ,   3.57 ,  15.84 ,   0.   ,   0.   ,   3.   ,   4.   ],
       [  4.   , 146.7  ,  62.   ,   3.69 ,   3.19 ,  20.   ,   1.   ,   0.   ,   4.   ,   2.   ],
       [  4.   , 140.8  ,  95.   ,   3.92 ,   3.15 ,  22.9  ,   1.   ,   0.   ,   4.   ,   2.   ],
       [  6.   , 167.6  , 123.   ,   3.92 ,   3.44 ,  18.3  ,   1.   ,   0.   ,   4.   ,   4.   ],
       [  6.   , 167.6  , 123.   ,   3.92 ,   3.44 ,  18.9  ,   1.   ,   0.   ,   4.   ,   4.   ],
       [  8.   , 275.8  , 180.   ,   3.07 ,   4.07 ,  17.4  ,   0.   ,   0.   ,   3.   ,   3.   ],
       [  8.   , 275.8  , 180.   ,   3.07 ,   3.73 ,  17.6  ,   0.   ,   0.   ,   3.   ,   3.   ],
       [  8.   , 275.8  , 180.   ,   3.07 ,   3.78 ,  18.   ,   0.   ,   0.   ,   3.   ,   3.   ],
       [  8.   , 472.   , 205.   ,   2.93 ,   5.25 ,  17.98 ,   0.   ,   0.   ,   3.   ,   4.   ],
       [  8.   , 460.   , 215.   ,   3.   ,   5.424,  17.82 ,   0.   ,   0.   ,   3.   ,   4.   ],
       [  8.   , 440.   , 230.   ,   3.23 ,   5.345,  17.42 ,   0.   ,   0.   ,   3.   ,   4.   ],
```