Week 5 - Advanced NumPy

Check the accompanying Jupyter notebook
for interactive examples!

Advanced Indexing

Advanced indexing (also sometimes called **fancy indexing**) happens when the selection object…

- … is a non-tuple sequence object (e.g. a list)
- … is a `np.ndarray`
- … is a tuple, where at least one element is one of the above

Advanced indexing always returns a **copy** instead of a view.

```
array = np.arange(9).reshape(3, 3)


array[1, 2]        # basic indexing
array[(1, 2)]      # basic indexing

array[[1, 2]]      # fancy indexing

index = np.arange(3)
array[index]       # fancy indexing

array[[1, 2], 0]   # fancy indexing
```

Integer Array Indexing

Simple case:

Index is a tuple of **np.ndarrays** of shape `(n, )` with integer `dtype`, **one for each dimension**:

- selects **n** elements
- `i`-th entry of each array corresponds to index of the `i`-th element along that dimension

```
array = np.arange(9).reshape(3, 3)

index_row = np.array([0, 1, 2])
index_col = np.array([2, 2, 0])

array[index_row, index_col]
# → [2, 5, 6]

index = np.zeros(10, dtype=int)

array[index, index]
# → [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Integer Array Indexing

Generalization:

Index is a tuple of np.ndarrays of **any broadcast-compatible shapes**, one for each dimension:

- selects as many elements as the broadcasted shape demands
- output shape is the broadcasted shape
- otherwise identical to the case before

```
array = np.arange(9).reshape(3, 3)

index_row = np.array([[0, 1], [0, 1]])
index_col = np.array([[0, 0], [1, 1]])

array[index_row, index_col]
# → [[0, 3],
#     [1, 4]]

array[index_row, 0]
# → [[0, 3],
#     [0, 3]]
```

Combining Advanced Indexing and Slicing

… is possible but can be complicated.

If you really need it, and you don’t get the result you’d intuitively expect, read this:

https://numpy.org/doc/stable/reference/arrays.indexing.html#combining-advanced-and-basic-indexing

Special Case: Single Nested List

What do you think the example on the right will return?

```
array = np.arange(9).reshape(3, 3)
array[[[0, 1], [0, 1]]]
```

Special Case: Single Nested List

What do you think the example on the right will return?

Answer: A FutureWarning!

Currently, this is interpreted as

`array[[0, 1], [0, 1]]`

i.e. two separate index lists. In a future version of NumPy, this will change to

`array[np.array([[0, 1], [0, 1]])]`

i.e. a single 2D array indexing only the first dimension.

⇒ **avoid using this for now.**

```
array = np.arange(9).reshape(3, 3)
array[[[0, 1], [0, 1]]]

# current behaviour

# → [0, 4]

# future behaviour

# → [[[0, 1, 2],
#      [3, 4, 5]],
#
#     [[0, 1, 2],
#      [3, 4, 5]]]
```

Example: `np.argsort`

How to sort one array according to values of another array?

```
letters = np.array(["y", "P", "n", "o", "t", "h"])
order = np.array([2, 1, 6, 5, 3, 4])

letters[np.argsort(order)]
# → ['P', 'y', 't', 'h', 'o', 'n']
```

We sort the array of numbers using `np.argsort`, which returns not the elements but their **indices**.

Because of **advanced indexing**, this array of indices can be used to index the `letters` array in the correct order.

`np.nonzero` & Friends

`array.nonzero()` returns the indices of all values in an array that are True-ish (≠ 0).

The output format is a tuple of **n** 1D arrays, where **n** is the number of dimensions of the array.

⇒ appropriate format for indexing

`np.where` and `np.argwhere` are similar, but have slightly different use-cases.

```
array = np.array([[0, 2], [0, -1]])

array.nonzero()
# → ([0, 1], [1, 1])

# often used with boolean arrays
(array < 0).nonzero()
# → ([1], [1])

# can be used for indexing
# (please immediately forget this)
array[(array < 0).nonzero()]
# → [-1]
```

Boolean Array Indexing

Boolean arrays can be used directly for indexing, without the need for `array.nonzero()`

If the array and the index have the same shape, a 1D array with all elements where the index is True is returned.

You can also use boolean indexing on individual dimensions. In conjunction with slices, this can again get a bit complicated.

```
array = np.array([[0, 2], [0, -1]])

index = np.array([[True, False],
                  [False, True]])

array[index]        # → [0, -1]

array[array < 0]    # → [-1]

array[[True, False], 0]
# → [0]
```

Sidenote: Bitwise Operators

If you want to perform element-wise logical operations (like and, or, xor) on NumPy arrays, you can use the bitwise operators:

and	&
or	|
exclusive or	^
not	~

```
array = np.array([[0, 2], [0, -1]])

array[(array < 0) | (array > 0)]
# → [2, -1]
```

Concatenating Arrays

NumPy provides confusingly many functions for concatenating arrays, but they all build on `np.concatenate`.

It takes a tuple of arrays (of appropriate shape) and concatenates them along a given axis.

`np.r_` is a useful shorthand.

- does not use function parentheses but [ ]
- first element is a **string** giving the axis
- remaining elements are arrays to be concatenated

```
array = np.arange(4).reshape(2, 2)

np.concatenate((array, array), axis=0)
# → [[0, 1],
#     [2, 3],
#     [0, 1],
#     [2, 3]]

np.concatenate((array, array), axis=1)
# →  [[0, 1, 0, 1],
#     [2, 3, 2, 3]]

np.r_["1", array, array]
# →  [[0, 1, 0, 1],
#     [2, 3, 2, 3]]
```

Coordinate Grids

Grids allow you to get all possible pairs of two arrays. This is useful for example when evaluating a multidimensional function over a coordinate grid.

NumPy provides the `meshgrid` function:

- takes any number of 1D arrays (i.e. number of dimensions of the grid)
- returns a list of n n-dimensional arrays that constitute all possible combinations

There are also the `np.mgrid` and `np.ogrid` objects that do similar things.

```
x_values = np.array([2, 3, 4])
y_values = np.array([-1, 0])

xx, yy = np.meshgrid(x_values, y_values)

xx
# → [[2, 3, 4],
#     [2, 3, 4]]

yy
# → [[-1, -1, -1],
#     [ 0,  0,  0]]

some_2d_function(xx, yy)
```