NumPy

November 23, 2021

Lau Møller Andersen
November 23 2021

1 What is NumPy

NumPy is a package that allows for manipulation and handling of n-dimensional arrays
It does effectively in terms of computational time by using C++ code
We often abbreviate it *np* when we import it

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

1.1 Dimensions in NumPy

Arrays can be of n dimesions where $n \in \mathbb{N}_0$
Except for zero-dimensional arrays they can be defined from lists or lists of lists

[2]:
```
# zero-dimesional array

zero_dim = np.array(0)
print('Zero-dimensional array:')
print('n dimensions: ' + str(zero_dim.ndim))
print('shape: ' + str(zero_dim.shape))
print('values:\n' + str(zero_dim))
print('\n')

# one-dimensional array
a_list = [0, 1] # lists are basic python classes
one_dim = np.array(a_list)
print('One-dimensional array:')
print('n dimensions: ' + str(one_dim.ndim))
print('shape: ' + str(one_dim.shape))
print('values:\n' + str(one_dim))
print('\n')

# two-dimensional array
a_list_of_lists = [[0, 1], [2, 3]] # lists are basic python classes
two_dim = np.array(a_list_of_lists)
```

```python
print('Two-dimensional array:')
print('n dimensions: ' + str(two_dim.ndim))
print('shape: ' + str(two_dim.shape))
print('values:\n' + str(two_dim))
print('\n')

# three-dimensional array
a_list_of_lists_of_lists = [[[0, 1], [2, 3]], [[4, 5], [6, 7]]] # lists are␣
 ↪basic python classes
three_dim = np.array(a_list_of_lists_of_lists)
print('Three-dimensional array:')
print('n dimensions: ' + str(three_dim.ndim))
print('shape: ' + str(three_dim.shape))
print('values:\n' + str(three_dim))
print('\n')

# and so on ad nauseam
```

```
Zero-dimensional array:
n dimensions: 0
shape: ()
values:
0


One-dimensional array:
n dimensions: 1
shape: (2,)
values:
[0 1]


Two-dimensional array:
n dimensions: 2
shape: (2, 2)
values:
[[0 1]
 [2 3]]


Three-dimensional array:
n dimensions: 3
shape: (2, 2, 2)
values:
[[[0 1]
  [2 3]]
```

```
[[4 5]
  [6 7]]]
```

1.2 Operations over dimensions

Many operations can be done over specific dimensions
Here, we'll apply `np.mean` in different ways - but first have a look at the default of the *axis* parameter in the documentation, before we carry on.
What does it mean to flatten an array?

[3]:
```
?np.mean
```

1.2.1 Flattening an array

Flattening means making the array one-dimensional by stacking the entries horizontally on end of another

[4]:
```
print(np.ravel(zero_dim))
print(np.ravel(one_dim))
print(np.ravel(two_dim))
print(np.ravel(three_dim))
```

```
[0]
[0 1]
[0 1 2 3]
[0 1 2 3 4 5 6 7]
```

1.2.2 Finding the mean

Now, we'll apply `np.mean` with the default axis (`axis=None`), which results in a single number because it is done over the flattened array

[5]:
```
print(np.mean(zero_dim))
print(np.mean(one_dim))
print(np.mean(two_dim))
print(np.mean(three_dim))
```

```
0.0
0.5
1.5
3.5
```

now, we are going to set `axis=0` - why can't we do it for `zero_dim`?

[ ]:
```

```

[6]:
```
print('Original data - one dim')
print(one_dim)
```

```python
print('Mean over first axis')
print(np.mean(one_dim, axis=0))
print('\n') # adding a line

print('Original data - two dim')
print(two_dim)
print('Mean over first axis')
print(np.mean(two_dim, axis=0))
print('\n') # adding a line

print('Original data - three dim')
print(three_dim)
print('Mean over first axis')
print(np.mean(three_dim, axis=0))
print('\n') # adding a line
```

```
Original data - one dim
[0 1]
Mean over first axis
0.5


Original data - two dim
[[0 1]
 [2 3]]
Mean over first axis
[1. 2.]


Original data - three dim
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
Mean over first axis
[[2. 3.]
 [4. 5.]]
```

When we are taking the mean over an axis, we are effectively removing that axis by collapsing it to a single number, i.e. the mean. That means that we reduce the number of dimensions by 1

[7]:
```python
one_dim_0_mean   = np.mean(one_dim,   axis=0)
two_dim_0_mean   = np.mean(two_dim,   axis=0)
three_dim_0_mean = np.mean(three_dim, axis=0)
```

```
print('n dimensions')
print(one_dim_0_mean.ndim)
print(two_dim_0_mean.ndim)
print(three_dim_0_mean.ndim)

print('shapes')
print(one_dim_0_mean.shape)
print(two_dim_0_mean.shape)
print(three_dim_0_mean.shape)
```

```
n dimensions
0
1
2
shapes
()
(2,)
(2, 2)
```

1.2.3 Calculating the means manually

We'll now calculate the means manually to showcase how they are calculated by `np.mean`.

[8]:
```
one_dim_0_man_mean   = (one_dim[0]   + one_dim[1])   / 2
two_dim_0_man_mean   = (two_dim[0]   + two_dim[1])   / 2 ## == (two_dim[0, :]
↪+ two_dim[1, :])
three_dim_0_man_mean = (three_dim[0] + three_dim[1]) / 2 ## == (three_dim[0, :,
↪:]   + three_dim[1, :, :])

## check whether they agree with the means calculated by np.mean

print('logical tests')
print(one_dim_0_mean   == one_dim_0_man_mean)
print(two_dim_0_mean   == two_dim_0_man_mean)
print(three_dim_0_mean == three_dim_0_man_mean)
print('\n')

## let's look a bit further at the two- and three-dimensional arrays
print('2d')
print(two_dim[0, :])
print(two_dim[1, :])
print('2d mean')
print(two_dim_0_mean)
print('\n')

print('3d')
print(three_dim[0, :, :])
```

```
print(three_dim[1, :, :])
print('3d mean')
print(three_dim_0_mean)
print('\n')
```

```
logical tests
True
[ True  True]
[[ True  True]
 [ True  True]]


2d
[0 1]
[2 3]
2d mean
[1. 2.]


3d
[[0 1]
 [2 3]]
[[4 5]
 [6 7]]
3d mean
[[2. 3.]
 [4. 5.]]
```

1.3 Logical indexing

We'll now have a quick look at how to logically index arrays

[9]:
```
x = np.arange(0, 10)
y = np.array([1, 1, 1, 1, 1,
              2, 2, 2, 2, 2])
print(x[y == 1])
print(x[y == 2])
print(x[y == 3])

X = np.repeat(x, 10).reshape(10, 10)
print(X)

print(X[y==1, y==2])
```

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

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