UNIVERSITY OF PORTSMOUTH

Python for Data Analysis NumPy Basics (Week 4)

Atefeh Khazaei

atefeh.khazaei@port.ac.uk

What we will learn this week?

- NumPy arrays
- Operations on NumPy arrays

NumPy

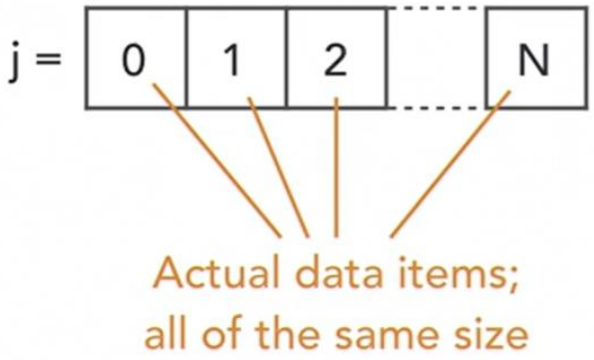
- NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.
- One of the reasons NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data.
- Excellent choice for large, homogeneous data sets.
- A foundation for many mathematical packages, and to integrate Python with C and Fortran

NumPy (cont.)

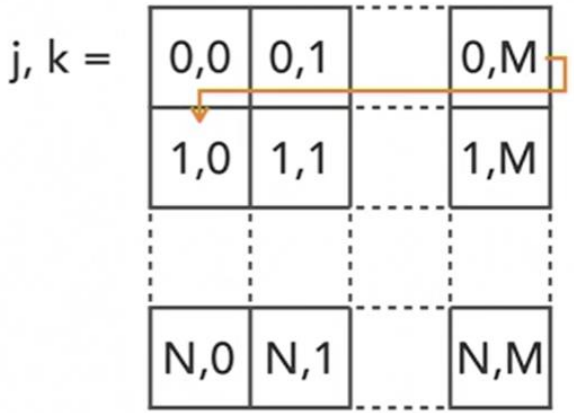
- There are a number of reasons for the efficiency of NumPy on large arrays of data:
  - NumPy internally stores data in a contiguous block of memory, independent of other built-in Python objects. NumPy arrays also use much less memory than built-in Python sequences.
  - NumPy operations perform complex computations on entire arrays without the need for Python for loops.

NumPy (cont.)

- The elements sit side by side in the memory and they all have the same size.




```
v[j]
ndim = 1
shape = (N,)
```




```
A[j,k]
ndim = 2
shape = (N,M)
```

NumPy (cont.)

- To give you an idea of the performance difference, consider a NumPy array of one million integers, and the equivalent Python list:

```
import numpy as np
my_arr = np.arange(1000000)
my_list = list(range(1000000))
```

```
%time for _ in range(10): my_arr2 = my_arr * 2
```

```
Wall time: 18 ms
```

```
%time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

```
Wall time: 856 ms
```

The NumPy ndarray
A Multidimensional Array Object

- One of the key features of NumPy is its N-dimensional array object, or ndarray which is a fast, flexible container for large datasets in Python.
- Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

The NumPy ndarray (cont.)
A Multidimensional Array Object

- An ndarray is a generic multidimensional container for homogeneous data.
  - All of the elements must be the same type.
- Every array has:
  - a shape, a tuple indicating the size of each dimension.
  - a dtype, an object describing the data type of the array

```
import numpy as np

# Generate some random data
data = np.random.randn(2, 3)
```

```
data
```

```
array([[-0.39474072,  0.42497135,  1.5441822 ],
       [ 2.2756087 ,  1.7820965 ,  0.61795775]])
```

```
data * 10
```

```
array([[-3.94740722,  4.24971352, 15.44182195],
       [22.75608696, 17.82096497,  6.17957748]])
```

```
data + data
```

```
array([[-0.78948144,  0.8499427 ,  3.08836439],
       [ 4.55121739,  3.56419299,  1.2359155 ]])
```

```
data.shape
```

```
(2, 3)
```

```
data.dtype
```

```
dtype('float64')
```

The NumPy ndarray (cont.)
Creating ndarrays

- The easiest way to create an array is to use the array function.

```
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
arr1
```

```
array([6. , 7.5, 8. , 0. , 1. ])
```

```
data2 = [6, 7.5, 8, 0, 'b']
arr2 = np.array(data2)
arr2
```

```
array(['6', '7.5', '8', '0', 'b'], dtype='<U32')
```

```
data3 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr3 = np.array(data3)
arr3
```

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

```
data4 = [[1, 2, 3, 4], [5, 6]]
arr4 = np.array(data4)
arr4
```

```
<ipython-input-22-6f6401a593ad>:2: VisibleDeprecationWa
rning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarray
s with different lengths or shapes) is deprecated. If y
ou meant to do this, you must specify 'dtype=object' wh
en creating the ndarray.
  arr4 = np.array(data4)
```

```
array([list([1, 2, 3, 4]), list([5, 6])], dtype=object)
```

The NumPy ndarray (cont.)
Creating ndarrays

- There are a number of other functions for creating new arrays.
  - zeros and ones create arrays of 0s or 1s, respectively, with a given length or shape.
  - Empty creates an array without initializing its values to any particular value.
    - Initial content is random and depends on the state of memory
    - It's not safe to assume that np.empty will return an array of all zeros. In some cases, it may return uninitialized "garbage" values.

```
np.zeros(10)
```

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

```
x = np.ones((3, 6))
x
```

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

```
np.zeros_like(x)
```

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

```
np.empty((2, 3, 2))
```

```
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

The NumPy ndarray (cont.)

Creating ndarrays

See more examples in the practical files.

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones`, `ones_like`	Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a ones array of the same shape and dtype
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and dtype with all values set to the indicated "fill value" `full_like` takes another array and produces a filled array of the same shape and dtype
`eye, identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

The NumPy ndarray (cont.)
Arithmetic with NumPy Arrays

- Arrays are important because they enable you to express batch operations on data without writing any for loops.
- NumPy users call this vectorization.
- Any arithmetic operations between equal-size arrays apply the operation element-wise.

```
import numpy as np
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
arr
```

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

```
arr * arr
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

```
1 / arr
```

```
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])
```

```
arr ** 0.5
```

```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```

```
arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
arr > arr2
```

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

The NumPy ndarray (cont.)
Basic Indexing and Slicing

- NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements.

- ✓ The value is propagated (or broadcasted henceforth) to the entire selection.
- ✓ The array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

```
import numpy as np
arr = np.arange(10)
arr
```

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

```
arr[5]
```

```
5
```

```
arr[5:8]
```

```
array([5, 6, 7])
```

```
arr[5:8] = 12
arr
```

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

The NumPy ndarray (cont.)
Indexing and Slicing / 2D arrays

```
import numpy as np
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d
```

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

```
arr2d[2]
```

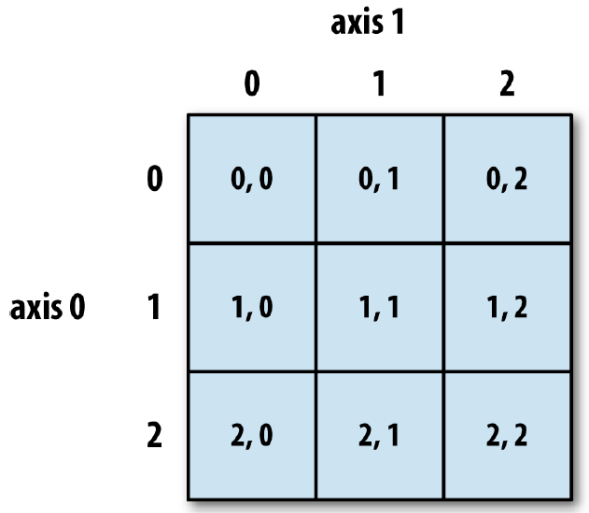
```
array([7, 8, 9])
```

```
arr2d[0][2]
```

```
3
```

```
arr2d[0,2]
```

```
3
```



The NumPy ndarray (cont.)

Indexing and Slicing / 3D arrays

```
import numpy as np
# arr3d is a 2 × 2 × 3 array
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
arr3d
```

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

```
arr3d[0]
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
arr3d[1, 0]
```

```
array([7, 8, 9])
```

```
arr3d[1, 0, 2]
```

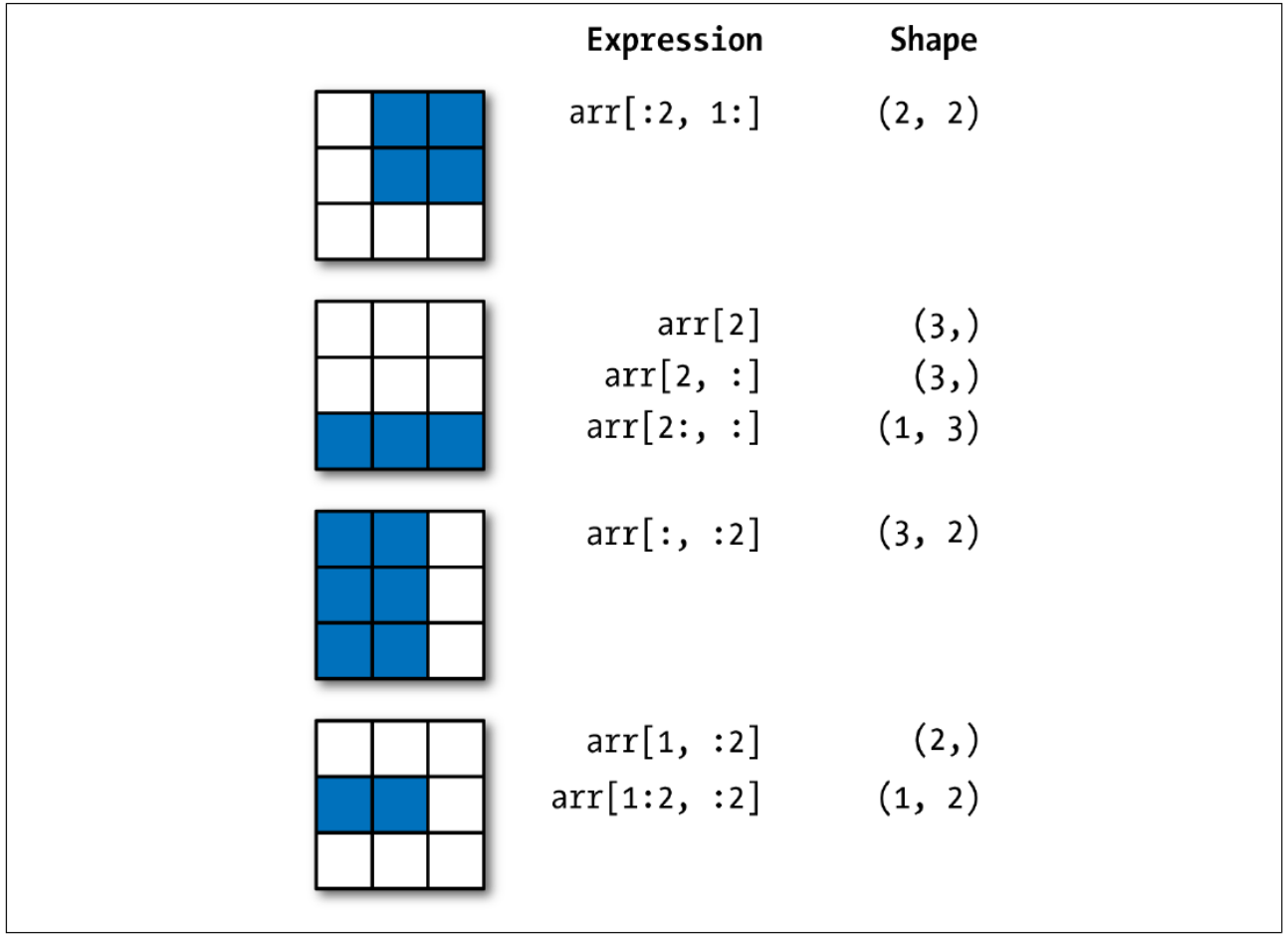
```
9
```

```
arr3d[0] = 42
arr3d
```

```
array([[[42, 42, 42],
        [42, 42, 42]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

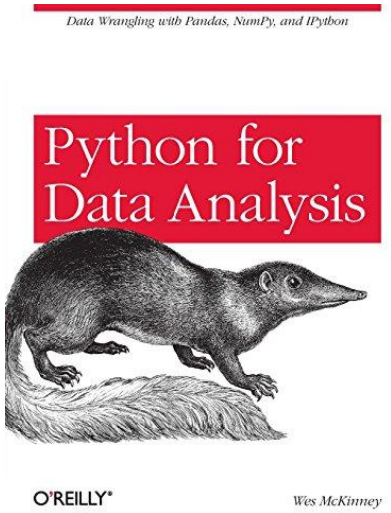
The NumPy ndarray (cont.)
Indexing with slices



Expression	Shape
arr[:2, 1:]	(2, 2)
arr[2]	(3,)
arr[2, :]	(3,)
arr[2:, :]	(1, 3)
arr[:, :2]	(3, 2)
arr[1, :2]	(2,)
arr[1:2, :2]	(1, 2)

References & More Resources

- References:
  - McKinney, Wes. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.



- More Resources:
  - Python Data Analysis on Linkedin Learning:

    https://www.linkedin.com/learning/python-data-analysis-2
  - Learning Python on Linkedin Learning

    https://www.linkedin.com/learning/learning-python
    - To use Linkedin Learning, you can log in with your university account:

      https://myport.port.ac.uk/study-skills/linkedin-learning

COURSE

Python Data Analysis

By: Michele Vallisneri

COURSE

Learning Python

By: Joe Marini

Practical Session

- Please download Week04_example.ipynb and Week04_nparray.ipynb files, run them, and then complete their comments.
- Please read the practical sheet (Week04_Practicals.pdf) and do the exercise.