Introduction to Programming in Python

Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Working with PyCharm

There are no slides for this bit. Some nice tutorials are available on the web, like this one (https://www.mygreatlearning.com/blog/pycharm-tutorial/#runningacodeinpycharm).

PyCharm also has extensive documentation:

- https://www.jetbrains.com/help/pycharm/creating-and-running-your-first-python-project.html#summary (https://www.jetbrains.com/help/pycharm/creating-and-running-your-first-python-project.html#summary)
- https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html#summary (https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html#summary)

Some important packages

Numpy

- Numpy is the most fundamental package for numerical computations in Python (user guide (https://docs.scipy.org/doc/numpy/user/index.html)).
- Basically, it provides a datatype `ndarray` and defines mathematical functions for it
- An array is similar to a `list`, except that
    - it can have more than one dimension;
    - its elements are homogeneous (they all have the same type).
- NumPy provides a large number of functions (*ufuncs*) that operate elementwise on arrays. This allows *vectorized* code, avoiding loops (which are slow in Python).

Constructing Arrays

- Arrays can be constructed using the `array` function which takes sequences (e.g, lists) and converts them into arrays. The data type is inferred automatically or can be specified.

In [ ]:

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a)
```

In [ ]:

```
a = np.array([1, 2, 3, 4], dtype='float64')  # or np.array([1., 2., 3., 4.])
print(a)
```

- NumPy uses C++ data types which differ from Python's (though `float64` is equivalent to Python's `float`).

- Nested lists result in multidimensional arrays. We won't need anything beyond two-dimensional (i.e., a matrix or table).

In [ ]:

```
a = np.array([[1., 2.], [3., 4.]]); a
```

In [ ]:

```
a.shape # number of rows and columns
```

- Other functions for creating arrays include:

In [ ]:

```
np.ones([2, 3])  # there's also np.zeros, and np.empty (which results in an uninitialized array).
```

In [ ]:

```
np.arange(0, 10, 2)  # like range, but creates an array instead of a list.
```

Indexing

- Indexing and slicing operations are similar to lists:

In [ ]:

```
a = np.array([[1., 2.], [3., 4.]])
print(a)
a[0, 0] # [row, column]
```

In [ ]:

```
b = a[:, 0]; b # entire first column. note that this yields a 1-dimensional array (vector), not a matrix with one column.
```

- Apart from indexing by row and column, arrays also support *Boolean* indexing:

```
In [ ]: a = np.arange(10); a
```

```
In [ ]: ind = a < 5; ind
```

```
In [ ]: a[ind]
```

A shorter way to write this is

```
In [ ]: a[a<5]
```

This is useful for selecting elements according to some condition

Arithmetic and `ufuncs`

- NumPy `ufuncs` are functions that operate elementwise:

In [ ]:

```
a = np.arange(1, 5); np.sqrt(a)
```

- Other useful ufuncs are `exp`, `log`, `abs`, and `sqrt`.
- Basic arithmetic on arrays works elementwise:

In [ ]:

```
a = np.arange(1, 5); b = np.arange(5, 9); a, b, a+b, a-b, a/b.astype(float)
```

Broadcasting

- Operations between scalars and arrays are also supported:

In [ ]:
```
np.array([1, 2, 3, 4]) + 2
```

- This is a special case of a more general concept known as *broadcasting*, which allows operations between arrays of different shapes.
- NumPy compares the shapes of two arrays dimension-wise. It starts with the trailing dimensions, and then works its way forward. Two dimensions are compatible if
    - they are equal, or
    - one of them is 1 (or not present).
- In the latter case, the singleton dimension is "stretched" to match the larger array.

- Example:

```
In [ ]: x = np.arange(6).reshape((2, 3)); x  # x has shape (2,3).
```

```
In [ ]: m = np.mean(x, axis=0); m  # m has shape (3,).
```

```
In [ ]: x-m  # the trailing dimension matches, and m is stretched to match the 2 rows of x.
```

Array Reductions

- *Array reductions* are operations on arrays that return scalars or lower-dimensional arrays, such as the `mean` function used above.
- They can be used to summarize information about an array, e.g., compute the standard deviation:

In [ ]:

```
a = np.random.randn(300, 3)  # create a 300x3 matrix of standard normal variates.
a.std(axis=0)  # or np.std(a, axis=0)
```

- By default, reductions operate on the *flattened* array (i.e., on all the elements). For row- or columnwise operation, the `axis` argument has to be given.
- Other useful reductions are `sum`, `median`, `min`, `max`, `argmin`, `argmax`, any, and `all` (see help).

Saving Arrays to Disk

- There are several ways to save an array to disk:

In [ ]:

```
np.save('myfile.npy', a) # save `a` as a binary .npy file
```

In [ ]:

```
import os
print(os.listdir('.'))
```

In [ ]:

```
b = np.load('myfile.npy')  # load the data into variable b
os.remove('myfile.npy')  # clean up
```

In [ ]:

```
np.savetxt('myfile.csv', a, delimiter=',')  # save `a` as a CSV file (comma seperated values, can be read by MS Excel)
```

In [ ]:

```
b = np.loadtxt('myfile.csv', delimiter=',')  # load data into `b`.
os.remove('myfile.csv')
```

Reading (recommended)

- https://python-course.eu/numerical-programming/ (https://python-course.eu/numerical-programming/) 1-9, 11

Homework

Ex. 1-16 of https://github.com/rougier/numpy-100/blob/master/100_Numpy_exercises.md (https://github.com/rougier/numpy-100/blob/master/100_Numpy_exercises.md). Skip 4 and 11. Try to do these in PyCharm.