Lab 1: NumPy in Python

NumPy is a popular Python library for numerical computing, which provides high-performance multidimensional arrays and tools for working with these arrays. NumPy is used extensively in scientific computing, data analysis, machine learning, and other applications where numerical data is manipulated.

NumPy provides an array object that is similar to Python's built-in list, but is more efficient for numerical operations. The NumPy array is a homogenous data structure, which means that all elements in the array must have the same data type. The array can be created from a list or other sequence, and it can be indexed and sliced like a list.

NumPy also provides a wide range of mathematical functions and operators that can be used to manipulate arrays. These include basic arithmetic operations, such as addition, subtraction, multiplication, and division, as well as more advanced functions, such as trigonometric functions, exponential functions, and linear algebra operations.

NumPy is often used in conjunction with other Python libraries, such as SciPy, matplotlib, and pandas, to create powerful data analysis and visualization tools. It is also widely used in machine learning frameworks, such as TensorFlow and scikit-learn.

To use NumPy in Python, you first need to install the library using a package manager such as pip or conda. If you already have Python, you can install NumPy with:

```
conda install numpy
```

or

```
pip install numpy
```

Once installed, you can import NumPy using the following statement:

```
import numpy as np
```

This statement imports the NumPy library and assigns it the alias "np". You can then use NumPy functions and objects by prefixing them with "np.", for example:

```
import numpy as np

# create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# perform a mathematical operation on the array
arr_squared = np.square(arr)

# print the result
print(arr_squared)
```

This code creates a NumPy array containing the integers 1 through 5, then uses the NumPy `square` function to square each element of the array. The resulting array is then printed to the console.

Creating arrays

Manual construction of arrays

- **1-D**:

```
>>>
```

```
>>> a = np.array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
>>> a.ndim
1
>>> a.shape
(4,)
>>> len(a)
4
```

- **2-D, 3-D, ...**:

```
>>>
```

```
>>> b = np.array([[0, 1, 2], [3, 4, 5]])    # 2 x 3 array
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b)     # returns the size of the first dimension
2

>>> c = np.array([[[1], [2]], [[3], [4]]])
>>> c
array([[[1],
        [2]],
       [[3],
        [4]]])
>>> c.shape
(2, 2, 1)
```

Exercise: Simple arrays

- Create a simple two dimensional array. First, redo the examples from above. And then create your own: how about odd numbers counting backwards on the first row, and even numbers on the second?
- Use the functions **`len()`**, **`numpy.shape()`** on these arrays. How do they relate to each other? And to the `ndim` attribute of the arrays?

Functions for creating arrays

In practice, we rarely enter items one by one…

- Evenly spaced:

```
>>>
```

```
>>> a = np.arange(10) # 0 .. n-1  (!)
```

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.arange(1, 9, 2) # start, end (exclusive), step
>>> b
array([1, 3, 5, 7])
```

- or by number of points:

```
>>>
>>> c = np.linspace(0, 1, 6)    # start, end, num-points
>>> c
array([0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
>>> d = np.linspace(0, 1, 5, endpoint=False)
>>> d
array([0. ,  0.2,  0.4,  0.6,  0.8])
```

- Common arrays:

```
>>>
>>> a = np.ones((3, 3))  # reminder: (3, 3) is a tuple
>>> a
array([[1.,  1.,  1.],
       [1.,  1.,  1.],
       [1.,  1.,  1.]])
>>> b = np.zeros((2, 2))
>>> b
array([[0.,  0.],
       [0.,  0.]])
>>> c = np.eye(3)
>>> c
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> d = np.diag(np.array([1, 2, 3, 4]))
>>> d
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

- **`np.random`**: random numbers (Mersenne Twister PRNG):

```
>>>
>>> a = np.random.rand(4)       # uniform in [0, 1]
>>> a
array([ 0.95799151,  0.14222247,  0.08777354,  0.51887998])

>>> b = np.random.randn(4)      # Gaussian
>>> b
array([ 0.37544699, -0.11425369, -0.47616538,  1.79664113])

>>> np.random.seed(1234)        # Setting the random seed
```

Exercise: Creating arrays using functions

- Experiment with `arange`, `linspace`, `ones`, `zeros`, `eye` and `diag`.
- Create different kinds of arrays with random numbers.
- Try setting the seed before creating an array with random values.
- Look at the function `np.empty`. What does it do? When might this be useful?

Basic data types

You may have noticed that, in some instances, array elements are displayed with a trailing dot (e.g. `2.` vs `2`). This is due to a difference in the data-type used:

```
>>>
```

```
>>> a = np.array([1, 2, 3])
>>> a.dtype
dtype('int64')

>>> b = np.array([1., 2., 3.])
>>> b.dtype
dtype('float64')
```

Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

You can explicitly specify which data-type you want:

```
>>>
```

```
>>> c = np.array([1, 2, 3], dtype=float)
>>> c.dtype
dtype('float64')
```

The **default** data type is floating point:

```
>>>
```

```
>>> a = np.ones((3, 3))
>>> a.dtype
dtype('float64')
```

There are also other types:

Complex:

```
>>>
```

```
>>> d = np.array([1+2j, 3+4j, 5+6*1j])
>>> d.dtype
dtype('complex128')
```

Bool:

```
>>>
```

```
>>> e = np.array([True, False, False, True])
>>> e.dtype
dtype('bool')
```

Strings:

```
>>>
```

```
>>> f = np.array(['Bonjour', 'Hello', 'Hallo'])
>>> f.dtype     # <--- strings containing max. 7 letters
dtype('S7')
```

Basic visualization

Now that we have our first data arrays, we are going to visualize them.

Matplotlib is a 2D plotting package. We can import its functions as below:

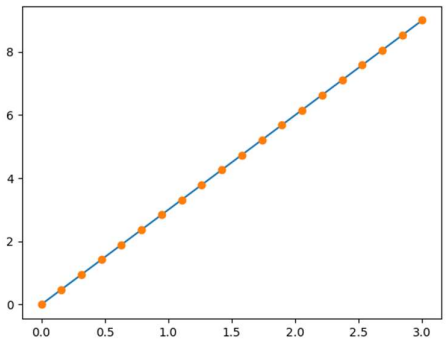
```
>>>
```

```
>>> import matplotlib.pyplot as plt  # the tidy way
```

- **1D plotting**:

```
>>>
```

```
>>> x = np.linspace(0, 3, 20)
>>> y = np.linspace(0, 9, 20)
>>> plt.plot(x, y)       # line plot
[<matplotlib.lines.Line2D object at ...>]
>>> plt.plot(x, y, 'o')  # dot plot
[<matplotlib.lines.Line2D object at ...>]
```



- **2D arrays** (such as images):

```
>>>
```

```
>>> image = np.random.rand(30, 30)
>>> plt.imshow(image, cmap=plt.cm.hot)
<matplotlib.image.AxesImage object at ...>
>>> plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar object at ...>
```