[Table of contents (../toc.ipynb)](../toc.ipynb)

NumPy

- Numpy is probably the most applied and used Python package. Almost any scientific package builds on numpy.
- Numpy provides multidimensional array objects, and allows to deal with matrix computations.
- Add to this, numpy provides interfaces to C and C++.
- Numpy's methods are very efficient implemented.
- Please find numpy's documentation here https://numpy.org/ (https://numpy.org/).
- Numpy is one building block of Python's scientific eco system. If you want to know more about scientific Python, consult the scipy lecture notes on https://scipy-lectures.org/ (https://scipy-lectures.org/).

Numpy import

First, let us import numpy, create a vector and compare this with a Python list, which ships per default with Python.

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

In [2]:
```
a_list = range(0, 8000)
a_np_array = np.arange(0, 8000)
```

Now, let us compare how long it takes to add 3 to each element of the list and the numpy array with `%timeit` magic command.

In [3]:
```
%timeit [a_i + 3 for a_i in a_list]
```

```
412 µs ± 16.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [4]:
```
%timeit a_np_array + 3
```

```
2.67 µs ± 40.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Numpy is much faster than the list and the code of numpy is easier to read!

Create arrays manually

- The "manual" syntax to create a numpy array is
    - `np.array([x_0, x_1, ..., x_n])` for one dimensional arrays
    - and `np.array([[x_0, x_1, ... x_n], [y_0, y_1, ..., y_n]])` for multidimensional arrays.
- Add to this, various functions like `np.ones()`, `np.eye()`, `np.arange()`, `np.linspace()`, and many more create specific arrays which are often required in matrix computing.

In [5]:
```
# a one dimensional array
a = np.array([0, 1, 2, 3])
a
```

Out[5]:
```
array([0, 1, 2, 3])
```

In [6]:
```
# here two dimesions
b = np.array([[0, 1], [2, 3]])
b
```

Out[6]:
```
array([[0, 1],
       [2, 3]])
```

In [7]:
```
# now check their shapes
print(a.shape)
print(b.shape)
```

```
(4,)
(2, 2)
```

Basic attributes

In addition to `ndarray.shape`, numpy arrays contain the attributes:

- `ndarray.ndim` which is the dimension of the array,
- `ndarray.size` which is the number of elements in the array,
- `ndarray.dtype` which is the type of the array (int16, int32, uint16, ..., the default is int64, or float64).

In [8]:
```
print('a.dtype = ', a.dtype)
print('b.dtype = ', b.dtype)
```

```
a.dtype =  int64
b.dtype =  int64
```

In [9]:
```
print('a.ndim = ', a.ndim)
print('b.ndim = ', b.ndim)
```

```
a.ndim =  1
b.ndim =  2
```

In [10]:
```
print('a.size = ', a.size)
print('b.size = ', b.size)
```

```
a.size =  4
b.size =  4
```

You can also specify the type.

In [11]:
```
np.array([1, 2, 3], dtype=np.uint16)
```

Out[11]:
```
array([1, 2, 3], dtype=uint16)
```

More array constructors

np.arange

- Create arrays with start, stop, and step size.

In [12]:
```
# linear growing array, zero based
np.arange(8)
```

Out[12]:
```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

In [13]:
```
# np.arange with start, end, step call
np.arange(0, 12, 2)
```

Out[13]:
```
array([ 0,  2,  4,  6,  8, 10])
```

np.linspace

- Create equidistant arrays with start, stop, and number of points.

In [14]:
```
np.linspace(0, 10, 3)
```

Out[14]:
```
array([ 0.,   5., 10.])
```

In [15]:
```
np.linspace(0, 10, 13)
```

Out[15]:
```
array([ 0.        ,  0.83333333,  1.66666667,  2.5       ,  3.33333333,
        4.16666667,  5.        ,  5.83333333,  6.66666667,  7.5       ,
        8.33333333,  9.16666667, 10.        ])
```

Special matrices

In [16]:
```
np.ones(3)
```

Out[16]:
```
array([1., 1., 1.])
```

In [17]:
```
np.ones(shape=(3, 3))
```

Out[17]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

In [18]:

```
np.eye(3)
```

Out[18]:

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

In [19]:
```
np.zeros(3)
```

Out[19]:
```
array([0., 0., 0.])
```

In [20]:
```
np.zeros(shape=(3, 3))
```

Out[20]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Random number arrays

- Numpy's `np.random.xxx` module offers many random number generators.

In [21]:
```
np.random.rand(3, 3) # uniform distribution over [0, 1)
```

Out[21]:
```
array([[0.85364355, 0.20699322, 0.18675912],
       [0.98231555, 0.52772497, 0.54494537],
       [0.93818789, 0.91388519, 0.44828561]])
```

In [22]:
```
np.random.randn(3, 3) # standard normal distribution
```

Out[22]:
```
array([[-2.20626319, -0.63197416,  0.80843017],
       [-0.33016822,  0.56584215, -0.16235999],
       [ 1.76202852, -0.18058352, -0.03921261]])
```

Operators

- All operators $+$, $-$, $*$, $>$, ..., work element wise per default.
- A new array will be created unless you use the $+=$, $-=$ and so forth operators.
- Matrix multiplication can be done with `@` operator (requires Python 3.5) or `.dot` method.

In [23]:
```
a = np.ones(4)
a + 4
```

Out[23]:
```
array([5., 5., 5., 5.])
```

In [24]:
```
a * 2
```

Out[24]:
```
array([2., 2., 2., 2.])
```

In [25]:
```
a[2] = 5
a > 4
```

Out[25]:
```
array([False, False,  True, False])
```

In [26]:
```
a = np.array([0, 1, 2, 3])
b = np.array([0, 1, 2, 3])
a * b
```

Out[26]:
```
array([0, 1, 4, 9])
```

In [27]:
```
a @ b # matrix product
```

Out[27]:
```
14
```

In [28]:
```
a.dot(b) # the "old" way to write a matrix prodcuct
```

Out[28]:
```
14
```

Universal functions

- Numpy offers almost all mathematical functions you might need, such as
    - exp
    - max, min
    - sqrt
    - argmax
    - median, mean, stdev
    - ...

In [29]:

```
a = np.linspace(0, 8, 16)
np.exp(a)
```

Out[29]:

```
array([1.00000000e+00, 1.70460487e+00, 2.90567775e+00, 4.95303242e+00,
       8.44296317e+00, 1.43919161e+01, 2.45325302e+01, 4.18182703e+01,
       7.12836271e+01, 1.21510418e+02, 2.07127249e+02, 3.53070116e+02,
       6.01845038e+02, 1.02590798e+03, 1.74876773e+03, 2.98095799e+03])
```

Shape modification

- There are many ways to change the shape of an array.
- Most prominent methods are `reshape` and `transpose`.

In [30]:
```
a = np.arange(12).reshape(3, 4)
a
```

Out[30]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [32]:
```
a.reshape(6, 2)
```

Out[32]:
```
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])
```

In [33]:
```
a.reshape(a.size) # Flatten an array, similar as a.shape(-1), a.flatten() or a.rave
l()
```

Out[33]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [32]:
```
a = np.zeros((1, 4))
a
```

Out[32]:
```
array([[0., 0., 0., 0.]])
```

In [33]:
```
a.transpose() # transpose
```

Out[33]:
```
array([[0.],
       [0.],
       [0.],
       [0.]])
```

In [34]:

```python
a.T # transpose, short hand code
```

Out[34]:

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

Indexing and iterating

- Is very similar to lists.
- Indexing is done by braces `[]`.

In [36]:
```
a = np.random.rand(3, 3)
```

In [37]:
```
# modify one element
a[1, 1] = 96
```

In [39]:
```
a[0, :]
```

Out[39]:
```
array([0.90156267, 0.15860076, 0.92586081])
```

In [40]:

```python
# iteration over first dimension
for row in a:
    print(row)
```

```
[0.90156267 0.15860076 0.92586081]
[ 0.82601745 96.          0.33312838]
[0.60133987 0.53074791 0.81656546]
```

If you want to iterate over all elements instead, use the flat attribute.

In [41]:
```
for a_i in a.flat:
    print(a_i)
```

```
0.9015626721609997
0.15860075987052058
0.9258608073857245
0.8260174527163244
96.0
0.3331283759775331
0.6013398695667754
0.5307479085804412
0.8165654631841754
```

Slicing arrays

- Slicing is also done with braces.
- There is a great "conversion" table for Matlab users here https://numpy.org/devdocs/user/numpy-for-matlab-users.html (https://numpy.org/devdocs/user/numpy-for-matlab-users.html).
- The basic slicing syntax is `[start:stop:step]`, see a full tutorial here https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html (https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html).

In [42]:

```
a = np.arange(20)
a
```

Out[42]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19])
```

All indices are zero based, the stop is not inclusive, and negative means to reverse counting (count from end to start).

In [43]:
```
a[-1] # last element
```
Out[43]:
```
19
```
In [44]:
```
a[0] # first element
```
Out[44]:
```
0
```
In [45]:
```
a[2:-1:2] # start from index 2 to last index and take every second value
```
Out[45]:
```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Stacking arrays

In [46]:
```
a = np.ones(3)
b = np.ones(3) + 2
np.vstack((a, b))
```

Out[46]:
```
array([[1., 1., 1.],
       [3., 3., 3.]])
```

In [47]:
```
np.hstack((a, b))
```

Out[47]:
```
array([1., 1., 1., 3., 3., 3.])
```

Comparing floats

- The `numpy.allclose` is ideal to compare arrays element wise with relative or absolute tolerance.
- The syntax is `np.allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)`.

In [48]:
```
np.allclose([1e10, 1e-7], [1.00001e10, 1e-8])
```

Out[48]:
```
False
```

In [49]:
```
np.allclose([1e10,1e-7], [1.00001e10,1e-8], atol=1e-3) # with larger absolute toler
ance, it returns True
```

Out[49]:
```
True
```

Linear algebra

- Numpy comes with a linear algebra module `numpy.linalg`.
- Next comes an example to solve an overdetermined system of equations where:
  - $A \in \mathbb{R}^{n \times m}$ denotes input data,
  - $X \in \mathbb{R}^{n}$ is the parameter vector,
  - and $b \in \mathbb{R}^{m}$ is the output vector which follows $b = A^\top X$

In [50]:
```
n = 2; m = 20
A = np.random.rand(n, m) # input data, two dimensions, 20 samples
X = np.array([[3], [5]]) # two dimensional prameter vector
b = A.T @ X # model the output

# now solve the system of equations
np.linalg.solve(a=(A @ A.T), b=(A @ b))
```

Out[50]:
```
array([[3.],
       [5.]])
```

In [51]:
```
np.linalg.inv(A @ A.T)
```

Out[51]:
```
array([[ 0.34901833, -0.31003372],
       [-0.31003372,  0.46588732]])
```

Read csv data

- Numpy's `genfromtxt` is very convenient to read csv files into numpy arrays.
- The syntax is `dat = genfromtxt('my_file.csv', delimiter=',')`, and there are many additional import properties like skip_rows, missing_values, ...

We will load this .csv file into a numpy array.

```
"Time", "Torque"
0, 200
1, 220
2, 225
3, 230
4, 231
```

In [81]:

```python
# This if else is a fix to make the file available for Jupyter and Travis CI
import os

if os.path.isfile('my_file.csv'):
    file = 'my_file.csv'
else:
    file = '02_tools-and-packages/my_file.csv'
```

In [82]:
```python
from numpy import genfromtxt

genfromtxt(file, delimiter=',', skip_header=1)
```

Out[82]:
```
array([[  0., 200.],
       [  1., 220.],
       [  2., 225.],
       [  3., 230.],
       [  4., 231.]])
```

You can also use the column names during the import with `names=True`.

In [53]:
```
dat = genfromtxt(file, delimiter=',', names=True)
dat
```

Out[53]:
```
array([(0., 200.), (1., 220.), (2., 225.), (3., 230.), (4., 231.)],
      dtype=[('Time', '<f8'), ('Torque', '<f8')])
```

In [54]:
```
print(dat["Time"])
print(dat["Torque"])
```

```
[0. 1. 2. 3. 4.]
[200. 220. 225. 230. 231.]
```

Matrix computations become so much simpler with numpy

Remember from last lesson that we used lists to plot for instance a parabola. This code was not that handy.

```
x = [i for i in range(-20, 21)]
y = [x_i**2 for x_i in x]
```

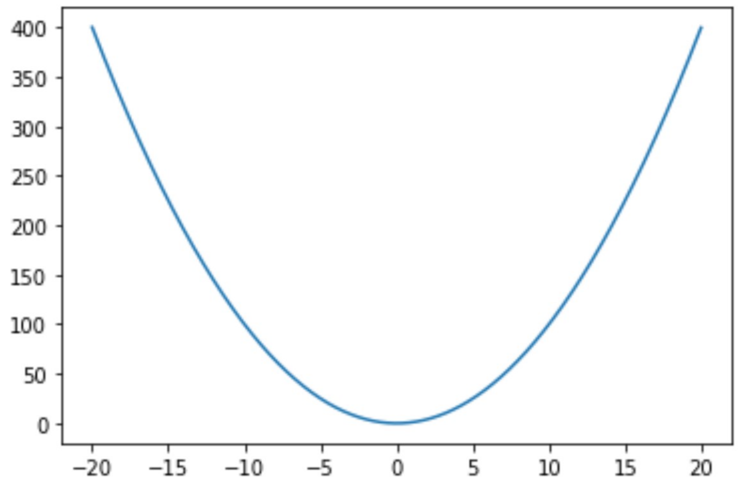
With numpy, such tasks become very simple.

In [55]:

```
%matplotlib inline
import matplotlib.pyplot as plt

x = np.arange(-20, 20, 0.01)

plt.plot(x, x**2) # x**2 is easy to read compared with a list [x_i**2 for x_i in x]
plt.show()
```



Exercise: Numpy mini project (20 minutes)

Now that you are familiar with matplotlib and numpy, you can solve the first more elaborate data task.

This is what you should strive for:

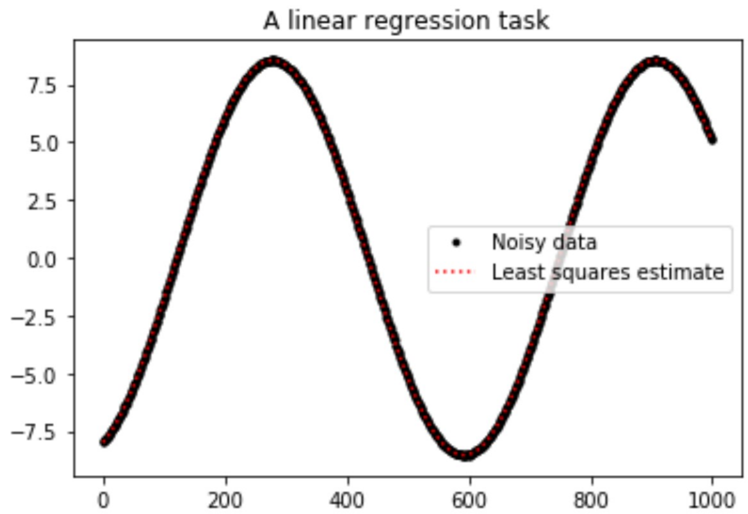
- Write a Python script which creates a linear multiplier model of type $AX \approx b$. The dimension of A should be 1000 times 2, and the respective dimension of X becomes 2.
- Select two values for X as you like. These are the true model parameters.
- Generate input data (random numbers or sine waves, as you like) and compute b.
- The $\approx$ sign in the above equation is due to noise that you should add to b.
- The noise should be Gaussian $\mathcal{N} \sim (0, 0.01)$.
- Plot the noisy data b.
- Use `np.linalg.solve` to compute the least squares solution $\hat{X}$ for the parameters.
- Use the same input data and $\hat{X}$ to compute and to plot the fit of the least squares solution.
- Print the true and estimated parameters.

Solution

Please find one possible solution in `solution_numpy.py` (solution_numpy.py) file.

In [56]:
```
%run solution_numpy.py
```

```
True params [ 3. -8.]
Estimated params [ 2.99942396 -7.99943241]
```




```
<Figure size 432x288 with 0 Axes>
```