Python-Kurs_3

April 30, 2021

1 Python course 3: NumPy and SciPy

Markus Reinert, 2021-04-30

1.1 Import the necesseray modules

[1]:
```
import numpy as np
from scipy import special
```

If you need to use a function many times, you can use the following notation:

[2]:
```
# from numpy import sin
```

For visualization purposes, we also import MatPlotLib, which we'll discuss in the next course.

[3]:
```
from matplotlib import pyplot as plt
```

1.2 Mathematics

Let's calculate $\sin(x)$ for some values of x.

[4]:
```
for x in [-np.pi / 2, 0, 1, np.pi / 2]:
    print("The sine of", x, "is", np.sin(x))
    # Uncomment the following line for a nicer way of printing
    # print(f"sin({x:+.2f}) = {np.sin(x)}")
```

```
The sine of -1.5707963267948966 is -1.0
The sine of 0 is 0.0
The sine of 1 is 0.8414709848078965
The sine of 1.5707963267948966 is 1.0
```

Let's calculate $\Gamma(4)$ which should be equal to 3! (factorial).

[5]:
```
special.gamma(4)
```

[5]:
```
6.0
```

1.3 Random numbers

[6]:

```python
print("A random number between 0 and 9:", np.random.randint(10))
```

```
A random number between 0 and 9: 8
```

To compute many random numbers at once, use the following notation:

[7]:

```python
print("100 random numbers between 0 and 9:")
np.random.randint(10, size=(100))
```

```
100 random numbers between 0 and 9:
```

[7]:

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

1.4 Arrays

Let's create a list and convert it to an array.

[8]:

```python
l = [3, 5, 2.1, 0, -3]
a = np.array(l)

print(l)
print(a)
```

```
[3, 5, 2.1, 0, -3]
[ 3.   5.   2.1  0.  -3. ]
```

They look the same, but note that all elements in `a` are floats (they have a dot at the end).

Let's create a vector and an array full with zeros.

[9]:

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

[9]:

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

[10]:

```python
np.zeros((3, 3))  # note the double-brackets
```

[10]:

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

Let's create a 3D-array with ones.

[11]:

```python
np.ones((2, 3, 4))
```

```
[11]: array([[[1., 1., 1., 1.],
              [1., 1., 1., 1.],
              [1., 1., 1., 1.]],

             [[1., 1., 1., 1.],
              [1., 1., 1., 1.],
              [1., 1., 1., 1.]]])
```

Let's create an array of ones that is compatible to our array `a` above.

```
[12]: b = np.ones_like(a)
      print(b)
```

```
[1. 1. 1. 1. 1.]
```

To create an array of tens, there are several ways:

```
[13]: np.full(3, fill_value=10)
```

```
[13]: array([10, 10, 10])
```

```
[14]: np.ones(3) * 10
```

```
[14]: array([10., 10., 10.])
```

1.5 Numerical 1D-ranges

The function `np.arange` works similar to Python's `range` …

```
[15]: y = np.arange(10)
      print(y)
```

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

… but can also do non-integer steps …

```
[16]: np.arange(0, 10, 0.5)
```

```
[16]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. ,
             6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
```

… and can go backwards as well.

```
[17]: np.arange(10, 0, -0.5)
```

```
[17]: array([10. ,  9.5,  9. ,  8.5,  8. ,  7.5,  7. ,  6.5,  6. ,  5.5,  5. ,
              4.5,  4. ,  3.5,  3. ,  2.5,  2. ,  1.5,  1. ,  0.5])
```

When we want to make sure that the start- and end-point are in the range, we use:

[18]:

```python
x = np.linspace(-1, 1, 100)
print(x)
```

```
[-1.         -0.97979798 -0.95959596 -0.93939394 -0.91919192 -0.8989899
 -0.87878788 -0.85858586 -0.83838384 -0.81818182 -0.7979798  -0.77777778
 -0.75757576 -0.73737374 -0.71717172 -0.6969697  -0.67676768 -0.65656566
 -0.63636364 -0.61616162 -0.5959596  -0.57575758 -0.55555556 -0.53535354
 -0.51515152 -0.49494949 -0.47474747 -0.45454545 -0.43434343 -0.41414141
 -0.39393939 -0.37373737 -0.35353535 -0.33333333 -0.31313131 -0.29292929
 -0.27272727 -0.25252525 -0.23232323 -0.21212121 -0.19191919 -0.17171717
 -0.15151515 -0.13131313 -0.11111111 -0.09090909 -0.07070707 -0.05050505
 -0.03030303 -0.01010101  0.01010101  0.03030303  0.05050505  0.07070707
  0.09090909  0.11111111  0.13131313  0.15151515  0.17171717  0.19191919
  0.21212121  0.23232323  0.25252525  0.27272727  0.29292929  0.31313131
  0.33333333  0.35353535  0.37373737  0.39393939  0.41414141  0.43434343
  0.45454545  0.47474747  0.49494949  0.51515152  0.53535354  0.55555556
  0.57575758  0.5959596   0.61616162  0.63636364  0.65656566  0.67676768
  0.6969697   0.71717172  0.73737374  0.75757576  0.77777778  0.7979798
  0.81818182  0.83838384  0.85858586  0.87878788  0.8989899   0.91919192
  0.93939394  0.95959596  0.97979798  1.        ]
```

1.6 From 1D to 2D

[19]:

```python
X, Y = np.meshgrid(x, y)
```

[20]:

```python
print("The x-coordinate has", x.ndim, "dimension and length:", x.size)
print("The y-coordinate has", y.ndim, "dimension and length:", y.size)
print("So the coordinate matrix X has", X.ndim, "dimensions")
print("with a shape of", X.shape, "and thus the size", X.size)
print(
    "The same for Y; dimensions:", Y.ndim,
    "and shape:", X.shape,
    "and size:", X.size,
)
```

```
The x-coordinate has 1 dimension and length: 100
The y-coordinate has 1 dimension and length: 10
So the coordinate matrix X has 2 dimensions
with a shape of (10, 100) and thus the size 1000
The same for Y; dimensions: 2 and shape: (10, 100) and size: 1000
```

[21]:

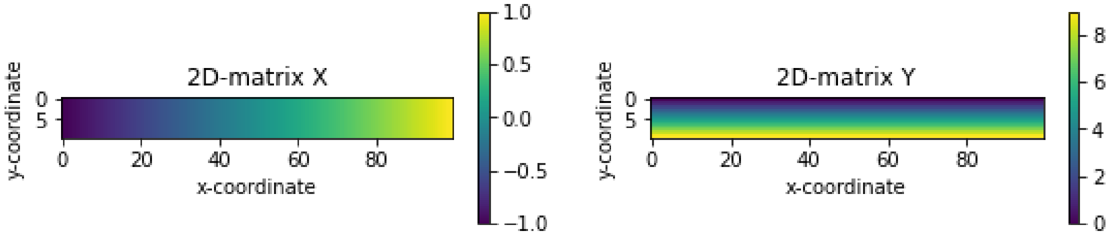
```python
fig, axs = plt.subplots(ncols=2, figsize=(10, 2))

ax = axs[0]
ax.set_title("2D-matrix X")
ax.set_xlabel("x-coordinate")
ax.set_ylabel("y-coordinate")
```

```
im = ax.imshow(X)
fig.colorbar(im, ax=ax)

ax = axs[1]
ax.set_title("2D-matrix Y")
ax.set_xlabel("x-coordinate")
ax.set_ylabel("y-coordinate")
im = ax.imshow(Y)
fig.colorbar(im, ax=ax)
```

[21]:
```
<matplotlib.colorbar.Colorbar at 0x7f1f2be590d0>
```



Note that you can also use the following notation, which is more to type but also works for lists:

[22]:
```
np.size(X)
```

[22]:
```
1000
```

[23]:
```
np.size(l)
```

[23]:
```
5
```

[24]:
```
np.size([1, 2, 3])
```

[24]:
```
3
```

For 1D-lists, this is equivalent to `len`, but this is not generally the case.

1.7 Calculations with vectors and matrices

Let's create a matrix …

[25]:
```
A = np.random.random((3, 4))
print(A)
```

```
[[0.70502187 0.4808327  0.61391773 0.70081782]
 [0.34821719 0.51828264 0.5294195  0.8239228 ]
 [0.03038753 0.55942508 0.09365669 0.93163837]]
```

… and two vectors

[26]:
```
vector = np.array([0, 1, 2, 3])
print(vector)

vector2 = np.array([1, 0, -1])
print(vector2)
```

```
[0 1 2 3]
[ 1  0 -1]
```

Elementwise multiplication works like this:

[27]:
```
A * vector
```

[27]:
```
array([[0.        , 0.4808327 , 1.22783546, 2.10245347],
       [0.        , 0.51828264, 1.058839  , 2.47176839],
       [0.        , 0.55942508, 0.18731339, 2.79491511]])
```

but if the dimensions don’t match, we have to extend the vector to the correct shape like this:

[28]:
```
A * vector2[:, np.newaxis]
```

[28]:
```
array([[ 0.70502187,  0.4808327 ,  0.61391773,  0.70081782],
       [ 0.        ,  0.        ,  0.        ,  0.        ],
       [-0.03038753, -0.55942508, -0.09365669, -0.93163837]])
```

The multiplication from linear algebra works like this:

[29]:
```
A @ vector
```

[29]:
```
array([3.81112162, 4.04889003, 3.54165358])
```

Or, more explicitely:

[30]:
```
np.dot(A, vector)
```

[30]:
```
array([3.81112162, 4.04889003, 3.54165358])
```

We can select parts of an array with indices:

[31]:
```
A[:, 1:3]
```

[31]:
```
array([[0.4808327 , 0.61391773],
       [0.51828264, 0.5294195 ],
       [0.55942508, 0.09365669]])
```

or based on a condition:

[32]:
```
(A > 0.5) & (A < 0.9)
```

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

Extract the values where the condition is True:

```
[33]: A[(A > 0.5) & (A < 0.9)]
```

```
[33]: array([0.70502187, 0.61391773, 0.70081782, 0.51828264, 0.5294195 ,
             0.8239228 , 0.55942508])
```

Obtain their indices:

```
[34]: np.where((A > 0.5) & (A < 0.9))
```

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

We can also use arrays in mathematical functions:

```
[35]: 2 * A  # elementwise multiplication with 2
```

```
[35]: array([[1.41004374, 0.9616654 , 1.22783546, 1.40163565],
              [0.69643439, 1.03656528, 1.058839  , 1.64784559],
              [0.06077506, 1.11885016, 0.18731339, 1.86327674]])
```

```
[36]: vector + 1  # elementwise addition
```

```
[36]: array([1, 2, 3, 4])
```

```
[37]: np.sqrt(A)  # calculate the square root of every value in A
```

```
[37]: array([[0.83965581, 0.69342101, 0.78352902, 0.83714863],
              [0.59009931, 0.7199185 , 0.72761219, 0.90770193],
              [0.17432019, 0.74794724, 0.30603381, 0.96521416]])
```

Last time, we have seen the following notation to do calculations for every element in a list. With NumPy, you can make this more efficiently, especially for large vectors.

Do you know how?

(Hint: I need two lines of code for it.)

```
[38]: [1 / x if x != 0 else 0.0 for x in vector]
```

```
[38]: [0.0, 1.0, 0.5, 0.3333333333333333]
```

1.8 Joining arrays into a single array

```
[39]: np.concatenate((A, vector2[:, np.newaxis]), axis=1)
```

```
[39]: array([[ 0.70502187,  0.4808327 ,  0.61391773,  0.70081782,  1.        ],
             [ 0.34821719,  0.51828264,  0.5294195 ,  0.8239228 ,  0.        ],
             [ 0.03038753,  0.55942508,  0.09365669,  0.93163837, -1.        ]])
```

```
[ ]:
```