1_numpy

October 23, 2023

1 NumPy arrays

The NumPy array, formally called ndarray in NumPy documentation, is the real workhorse of data structures for scientific and engineering applications. The NumPy array is similar to a list but where all the elements of the list are of the same type. The elements of a **NumPy array** are usually numbers, but can also be booleans, strings, or other objects. When the elements are numbers, they must all be of the same type.

[77]:
```
import numpy as np
```

1.1 Creating Numpy Arrays

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples
- using functions that are dedicated to generating numpy arrays, such as `arange`, `linspace`, etc.
- reading data from files which will be covered in the files section

1.1.1 From lists

For example, to create new vector and matrix arrays from Python lists we can use the `numpy.array` function.

[79]:
```
#this is a list
a = [0, 0, 1, 4, 7, 16, 31, 64, 127]
```

[82]:
```
type(a)
```

[82]:
```
list
```

[87]:
```
#this creates an array out of a list
b=np.array(a,dtype=float)
type(b)
```

[87]:
```
numpy.ndarray
```

1.1.2 Using array-generating functions

For larger arrays it is inpractical to initialize the data manually, using explicit python lists. Instead we can use one of the many functions in `numpy` that generate arrays of different forms. Some of the more common are:

[93]:
```
# create a range

x = np.arange(0, 10, 1) # arguments: start, stop, step
x
```

[93]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[109]:
```
x = np.arange(-5, -2, 0.1)
x
```

[109]:
```
array([-5. , -4.9, -4.8, -4.7, -4.6, -4.5, -4.4, -4.3, -4.2, -4.1, -4. ,
       -3.9, -3.8, -3.7, -3.6, -3.5, -3.4, -3.3, -3.2, -3.1, -3. , -2.9,
       -2.8, -2.7, -2.6, -2.5, -2.4, -2.3, -2.2, -2.1])
```

linspace and logspace The `linspace` function creates an array of N evenly spaced points between a starting point and an ending point. The form of the function is linspace(start, stop, N).If the third argument N is omitted,then N=50.

[101]:
```
# using linspace, both end points ARE included
np.linspace(0,10,25)
```

[101]:
```
array([ 0.        ,  0.41666667,  0.83333333,  1.25      ,  1.66666667,
        2.08333333,  2.5       ,  2.91666667,  3.33333333,  3.75      ,
        4.16666667,  4.58333333,  5.        ,  5.41666667,  5.83333333,
        6.25      ,  6.66666667,  7.08333333,  7.5       ,  7.91666667,
        8.33333333,  8.75      ,  9.16666667,  9.58333333, 10.        ])
```

`logspace` is doing equivelent things with logaritmic spacing. Other types of array creation techniques are listed below. Try around with these commands to get a feeling what they do.

[102]:
```
np.logspace(0, 10, 10, base=np.e)
```

[102]:
```
array([1.00000000e+00, 3.03773178e+00, 9.22781435e+00, 2.80316249e+01,
       8.51525577e+01, 2.58670631e+02, 7.85771994e+02, 2.38696456e+03,
       7.25095809e+03, 2.20264658e+04])
```

mgrid `mgrid` generates a multi-dimensional matrix with increasing value entries, for example in columns and rows:

[111]:
```
x, y = np.mgrid[0:1:0.1, 0:5] # similar to meshgrid in MATLAB
```

[117]:
```
x
```

[117]:
```
array([[0. , 0. , 0. , 0. , 0. ],
       [0.1, 0.1, 0.1, 0.1, 0.1],
       [0.2, 0.2, 0.2, 0.2, 0.2],
       [0.3, 0.3, 0.3, 0.3, 0.3],
       [0.4, 0.4, 0.4, 0.4, 0.4],
       [0.5, 0.5, 0.5, 0.5, 0.5],
       [0.6, 0.6, 0.6, 0.6, 0.6],
       [0.7, 0.7, 0.7, 0.7, 0.7],
       [0.8, 0.8, 0.8, 0.8, 0.8],
       [0.9, 0.9, 0.9, 0.9, 0.9]])
```

[116]:
```
y
```

[116]:
```
array([[0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.],
       [0., 1., 2., 3., 4.]])
```

[121]:
```
np.array([[1,2,3],[4,5,6],[7,8,9]])
```

[121]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

diag `diag` generates a diagonal matrix with the list supplied to it. The values can be also offset from the main diagonal.

[122]:
```
# a diagonal matrix
np.diag([1,2,3])
```

[122]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

[131]:
```
## diagonal with offset from the main diagonal
np.diag([1,2,3], k=1).size
```

[131]:
```
16
```

zeros and ones `zeros` and `ones` creates a matrix with the dimensions given in the argument and filled with 0 or 1.

[138]:
```
np.zeros((3,3))
```

[138]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[20]:
```
np.ones((3,3))
```

[20]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

1.2 Manipulating NumPy arrays

1.2.1 Slicing

Slicing is the name for extracting part of an array by the syntax `M[lower:upper:step]`

[139]:
```
A = np.array([1,2,3,4,5])
A
```

[139]:
```
array([1, 2, 3, 4, 5])
```

[141]:
```
A[1:4]
```

[141]:
```
array([2, 3, 4])
```

Any of the three parameters in `M[lower:upper:step]` can be ommited.

[142]:
```
A[::] # lower, upper, step all take the default values
```

[142]:
```
array([1, 2, 3, 4, 5])
```

[143]:
```
A[::2] # step is 2, lower and upper defaults to the beginning and end of the␣
↪array
```

[143]:
```
array([1, 3, 5])
```

Negative indices counts from the end of the array (positive index from the begining):

[144]:
```
A = np.array([1,2,3,4,5])
```

[145]:
```
A[-1] # the last element in the array
```

[145]:
```
5
```

[146]:
```
A[2:] # the last three elements
```

[146]:
```
array([3, 4, 5])
```

Index slicing works exactly the same way for multidimensional arrays:

[150]:
```
A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
A
```

[150]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

[151]:
```
# a block from the original array
A[1:3, 1:4]
```

[151]:
```
array([[11, 12, 13],
       [21, 22, 23]])
```

Note: Differences

Slicing can be effectively used to calculate differences for example for the calculation of derivatives. Here the position y_i of an object has been measured at times t_i and stored in an array each. We wish to calculate the average velocity at the times t_i from the arrays by

$$v_i = \frac{y_i - y_{i-1}}{t_i - t_{i-1}} \tag{1}$$

[152]:
```
y = np.array([ 0. , 1.3, 5. , 10.9, 18.9, 28.7, 40. ])
t = np.array([ 0. , 0.49, 1. , 1.5 , 2.08, 2.55, 3.2 ])
```

[154]:
```
v = (y[1:]-y[:-1])/(t[1:]-t[:-1])
v
```

[154]:
```
array([ 2.65306122,  7.25490196, 11.8       , 13.79310345, 20.85106383,
       17.38461538])
```

1.2.2 Reshaping

Arrays can be reshaped into any form, which contains the same number of elements.

[159]:
```
a=np.zeros(4)
a
```

[159]:
```
array([0., 0., 0., 0.])
```

[165]:
```
np.reshape(a,(2,2))
```

[165]:
```
array([[0., 0.],
       [0., 0.]])
```

1.2.3 Adding a new dimension: newaxis

With `newaxis`, we can insert new dimensions in an array, for example converting a vector to a column or row matrix.

[166]:
```
v = np.array([1,2,3])
v
```

[166]:
```
array([1, 2, 3])
```

[167]:
```
v.shape
```

[167]:
```
(3,)
```

[168]:
```
# make a column matrix of the vector v
v[:, np.newaxis]
```

[168]:
```
array([[1],
       [2],
       [3]])
```

[169]:
```
# column matrix
v[:,np.newaxis].shape
```

[169]:
```
(3, 1)
```

[170]:
```
# row matrix
v[np.newaxis,:].shape
```

[170]:
```
(1, 3)
```

1.2.4 Stacking and repeating arrays

Using function `repeat`, `tile`, `vstack`, `hstack`, and `concatenate` we can create larger vectors and matrices from smaller ones. Please try the individual functions yourself in your notebook. We wont discuss them in detail.

Tile and repeat

[176]:
```
a = np.array([[1, 2], [3, 4]])
a
```

[176]:
```
array([[1, 2],
       [3, 4]])
```

[172]:
```
# repeat each element 3 times
np.repeat(a, 3)
```

[172]:
```
array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4])
```

[173]:
```
# tile the matrix 3 times
np.tile(a, 3)
```

[173]:
```
array([[1, 2, 1, 2, 1, 2],
       [3, 4, 3, 4, 3, 4]])
```

Concatenate

[174]:
```
b = np.array([[5, 6]])
```

[175]:
```
np.concatenate((a, b), axis=0)
```

[175]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

[181]:
```
np.concatenate((a, b.T), axis=1)
```

[181]:
```
array([[1, 2, 5],
       [3, 4, 6]])
```

Hstack and vstack

[63]:
```
np.vstack((a,b))
```

[63]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

[64]:
```
np.hstack((a,b.T))
```

[64]:
```
array([[1, 2, 5],
       [3, 4, 6]])
```

1.3 Applying mathematical functions

All kinds of mathematica operations can be carried out on arrays. Typically these operation act element wise as seen from the examples below.

1.3.1 Operation involving one array

[182]:
```
a=np.arange(0, 10, 1.5)
a
```

[182]:
```
array([0. , 1.5, 3. , 4.5, 6. , 7.5, 9. ])
```

[185]:
```
a/2
```

[185]:
```
array([0.  , 0.75, 1.5 , 2.25, 3.  , 3.75, 4.5 ])
```

[186]:
```
a**2
```

[186]:
```
array([ 0.  ,  2.25,  9.  , 20.25, 36.  , 56.25, 81.  ])
```

[187]:
```
 np.sin(a)
```

[187]:
```
array([ 0.        ,  0.99749499,  0.14112001, -0.97753012, -0.2794155 ,
        0.93799998,  0.41211849])
```

[188]:
```
np.exp(-a)
```

[188]:
```
array([1.00000000e+00, 2.23130160e-01, 4.97870684e-02, 1.11089965e-02,
       2.47875218e-03, 5.53084370e-04, 1.23409804e-04])
```

[189]:
```
(a+2)/3
```

[189]:
```
array([0.66666667, 1.16666667, 1.66666667, 2.16666667, 2.66666667,
       3.16666667, 3.66666667])
```

1.3.2 Operations involving multiple arrays

Operation between multiple vectors allow in particular very quick operations. The operations address then elements of the same index. These operations are called vector operations since the concern the whole array at the same time. The product between two vectors results therefore not in a dot product, which gives one number but in an array of multiplied elements.

[192]:
```
a = np.array([34., -12, 5.,1.2])
b = np.array([68., 5.0, 20.,40.])
```

[193]:
```
a + b
```

[193]:
```
array([102. ,  -7. ,  25. ,  41.2])
```

[194]:
```
2*b
```

[194]:
```
array([136.,  10.,  40.,  80.])
```

[195]:
```
a*np.exp(-b)
```

[195]:
```
array([ 9.98743918e-29, -8.08553640e-02,  1.03057681e-08,  5.09802511e-18])
```

[198]:
```
v1=np.array([1,2,3])
v2=np.array([4,2,3])
```