DataScientest • com

NumPy for Data Science

Introduction to Numpy arrays

Introduction

> `numpy` (for *Numerical Python*) is a very advanced digital library for handling large multidimensional arrays and high level mathematical routines (linear algebra, statistics, complex mathematical functions, etc.).
>
> The object class that we will mainly manipulate is the **`array`** class of `numpy` .
>
> These arrays correspond to N-dimensional matrices which can contain very diverse data such as tabular data, time series or images.
>
> The interest of the `numpy` module lies in the possibility of applying operations on these arrays in a very efficient way, i.e. the number of **lines of code** required and the **computational time** to perform these operations will be very reduced compared to a traditional `Python` syntax.

1. Creating a **`numpy`** array

Unlike usual classes you will see in your training, a `numpy` array can be instantiated with many different constructors.

The argument of these constructors is usually a **`tuple`** containing the desired matrix dimensions. This tuple is called the **shape** of an array:

```
# Import of the numpy module under the alias 'np'
import numpy as np

# Creating a matrix of dimensions 5x10 filled with zeros
X = np.zeros(shape = (5, 10))

# Creating a 3-dimensional 3x10x10 matrix filled with ones
X = np.ones(shape = (3, 10, 10))
```

It is also possible to create an array from a **list** using the `np.array` constructor:

```
# Creating an array from a list defined by comprehension
X = np.array([2*i for i in range(10)]) # 0, 2, 4, 6, ..., 18

# Create an array from a list of lists
X = np.array([[1, 3, 3],
              [3, 3, 1],
              [1, 2, 0]])
```

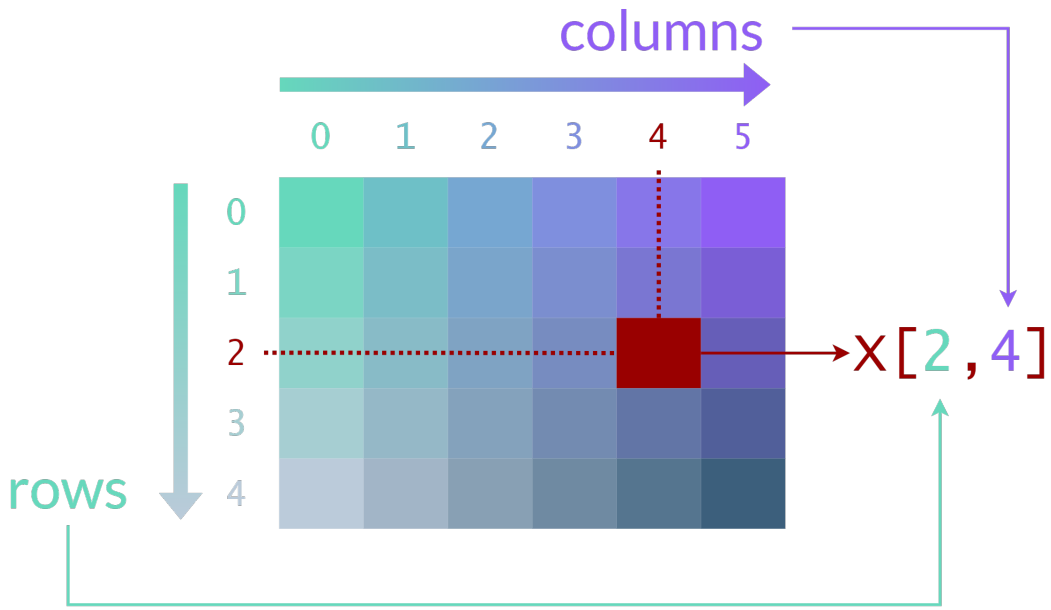
These three constructors are only examples. We will see other constructors later on.

2. Indexation of a **`numpy`** array

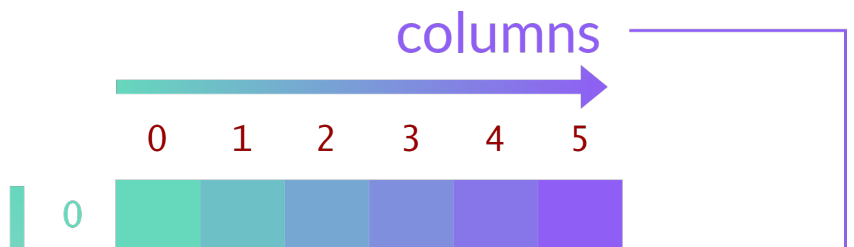
Unlike lists, a `numpy` array is multidimensional. Indexing must be done by entering the index that we want to access **on each dimension**:

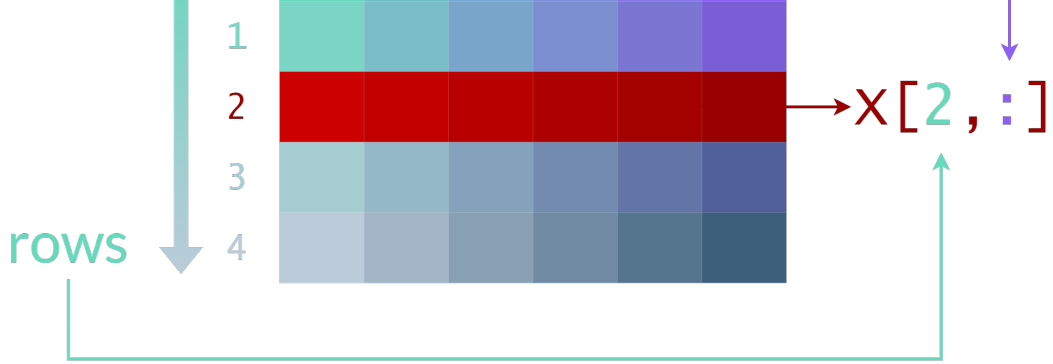
```
# Creating a matrix of dimensions 10x10 filled with ones
X = np.ones(shape = (10, 10))

# Displaying element at index (4, 3)
print(X[4, 3])

# Assigning the value -1 to the element of index (1, 5)
X[1, 5] = -1
```

As with all other Python indexable objects, the starting index of an axis is 0.



As for lists, it is possible to index an array using **slicing**:





It is possible to slice on **each dimension** of an array. In the following example, we will extract a **subarray** from X using slicing:

columns

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

# Insert your code here
X = np.zeros(shape=(6,6))
X[0:3, 0:3] = 1
X[3:, 3:] = -1
X
```

Out[1]:
```
array([[ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0., -1., -1., -1.],
       [ 0.,  0.,  0., -1., -1., -1.],
       [ 0.,  0.,  0., -1., -1., -1.]])
```

Hide solution

In [2]:

```
import numpy as np

# Creation of a matrix of shape 6x6 filled with zeros
X = np.zeros(shape = (6, 6))

# Assignment of the value 1 to the first diagonal block
X[:3, :3] = 1

# Assignment of the value -1 to the second diagonal block
X[3:, 3:] = -1

# Displaying the matrix
print(X)
```

```
[[ 1.  1.  1.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.]
 [ 0.  0.  0. -1. -1. -1.]
 [ 0.  0.  0. -1. -1. -1.]
 [ 0.  0.  0. -1. -1. -1.]]
```

- **(b)** Using array constructors and slicing, create and display the following matrix:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

You can either replace each row of a matrix of zeros with `np.array([0, 1, 2, 3, 4, 5])` or assign to each column its index.

In [19]:

```
# Insert your code here
import numpy as np
x = np.zeros((6,6))
v = [i for i in range(6)]
x[:] = v
x
```

Out[19]:

```
array([[0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.],
       [0., 1., 2., 3., 4., 5.]])
```

Hide solution

In [17]:

```
# First solution
X = np.zeros(shape = (6, 6))

# We replace each row with 'np.array([0, 1, 2, 3, 4, 5])'
for i in range (6):
    X[i ,:] = np.array ([0, 1, 2, 3, 4, 5])

# Display the matrix
print("First solution")
print(X)
print("\n")

# Second solution
X = np.zeros(shape = (6, 6))

# Each column of X is assigned its index
for i in range(6):
    X[:, i] = i

# Matrix display
print("Second solution")
print(X)
```

```
First solution
[[0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]]


Second solution
[[0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]]
```

3. Operations on Numpy arrays: Example

The `numpy` module is not only used to create matrices. Most of the time, the matrices will be created from real data.

The interest of the `numpy` module consists in its optimized **code** which allows to perform computations on large matrices in a very short time.

The `numpy` module contains basic mathematical functions such as:

Function	Numpy Function
e^x	`np.exp(x)`
$\log(x)$	`np.log(x)`
$\sin(x)$	`np.sin(x)`
$\cos(x)$	`np.cos(x)`
Rounding to **n** decimals	`np.round(x, decimals = n)`

The complete list of math operations for `numpy` is given here (https://numpy.org/doc/stable/reference/routines.math.html).

These functions can be applied to all `numpy` arrays, regardless of their dimensions:

```
X = np.array([i/100 for i in range(100)]) # 0, 0.01, 0.02, 0.03, ..., 0.98, 0.99

# Calculate the exponential of x for x = 0, 0.01, 0.02, 0.03, ..., 0.98, 0.99
exp_X = np.exp(X)
```

In the next cell we will create the array:

In [40]:

```
X = np.array([i/100 for i in range(100)])

# Insert your code here
def f(X):
    result = np.exp(np.sin(X) + np.cos(X))
    return result

print(np.round(f(X)[:10],2))
```

```
[2.72 2.75 2.77 2.8  2.83 2.85 2.88 2.91 2.94 2.96]
```

Hide solution

In [22]:

```
X = np.array([i/100 for i in range(100)])

# Definition of the function f
def f(X):
    return np.exp(np.sin(X) + np.cos(X))

# Calculation of f(X)
result = f(X)

# We round the result to 2 decimal places
rounded = np.round(result, decimals = 2)

# Displaying the first 10 elements of the rounded result
print(rounded[: 10])
```

```
[2.72 2.75 2.77 2.8  2.83 2.85 2.88 2.91 2.94 2.96]
```

- **(c)** Define a function called `f_python` that performs the same operation on every element of X using a `for` loop.

> The dimensions of an array `X` can be retrieved using the **`shape`** attribute of `X` which is a **tuple**: `shape = X.shape` .
>
> For an array with **one** dimension, the number of elements contained in this array corresponds to the **first** element of its shape: `n = X.shape[0]` .

In [36]:

```python
# Insert your code here
def f_python(X):
    if len(X.shape) == 1:
        return np.exp(np.sin(X) + np.cos(X))
    else:
        rows, cols = X.shape[0], X.shape[1]
        for r in range(rows):
            for c in range(cols):
                X[r,c] = np.exp(np.sin(X[r,c]) + np.cos(X[r,c]))
    return X

X = np.array([i/100 for i in range(100)])
Y = np.ones((5,4))
Z = np.zeros((6,6))
v = [i for i in range(6)]
Z[:] = v

print(f_python(X[:10]))
print(50*'-')
print(Y)
print(50*'-')
print(f_python(Y))
print(50*'-')
print(f_python(Z))
```

```
[2.71828183 2.74546328 2.7726365  2.79979313 2.8269247  2.85402258
 2.88107805 2.90808226 2.93502626 2.96190097]
--------------------------------------------------
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
--------------------------------------------------
[[3.98195654 3.98195654 3.98195654 3.98195654]
 [3.98195654 3.98195654 3.98195654 3.98195654]
 [3.98195654 3.98195654 3.98195654 3.98195654]
 [3.98195654 3.98195654 3.98195654 3.98195654]
 [3.98195654 3.98195654 3.98195654 3.98195654]]
--------------------------------------------------
[[2.71828183 3.98195654 1.63746709 0.42789712 0.24403439 0.50902299]
 [2.71828183 3.98195654 1.63746709 0.42789712 0.24403439 0.50902299]
 [2.71828183 3.98195654 1.63746709 0.42789712 0.24403439 0.50902299]
 [2.71828183 3.98195654 1.63746709 0.42789712 0.24403439 0.50902299]
 [2.71828183 3.98195654 1.63746709 0.42789712 0.24403439 0.50902299]
 [2.71828183 3.98195654 1.63746709 0.42789712 0.24403439 0.50902299]]
```

Hide solution

In [37]:
```
def f_python(X):
    n = X.shape[0]
    for i in range(n):
        X[i] = np.exp(np.sin(X[i]) + np.cos(X[i]))
```

> We will now compare the execution times of these two functions applied to a very large array with 10 million values.
>
> We will measure these execution times using the `time` module. To measure the execution time of a function, just take the **difference** between **the start time of execution** and **the end time**. We consider that the assignment of a variable is done instantly.

- **(d)** Run the next cell to compare execution times. This cell could take some time to run.

In [38]:
```
from time import time

# Creation of an array with 10 million values
X = np.array([i/1e7 for i in range(int(1e7))])

time_start = time()
f(X)
time_end = time()

runtime = time_end - time_start

print ("Calculating f with numpy took", runtime, "seconds")

time_start = time()
f_python(X)
time_end = time()

runtime = time_end - time_start

print ("Computing f with a for loop took", runtime, "seconds")
```

```
Calculating f with numpy took 0.26934242248535156 seconds
Computing f with a for loop took 29.84815502166748 seconds
```

Conclusion

As you can see, the computation time with a `for` loop is **extremely long**.

This is why it is always better to [illegible]

Unvalidate