CrashCoursesNumpy

November 10, 2023

#

Python NumPy Crash Course

Abdulrahman ALLOUCHE / 10 November 2023

python is an interpreted language and it is slow. However the lack of speed of python is not really an issue. Indeed:

- The time spent computing is balanced by a much smaller development time;
- Some python tools like NumPy are as fast as plain C (or C++).

NumPy is a Python library for performing large scale numerical computations. It is extremely useful, especially in machine learning. The main purpose of numpy is to provide a very efficient data structure called the numpy array, and the tools to manipulate such arrays. Numpy array is fast, because, arrays are processed with compiled code, optimized for the CPU.

0.0.1 1. Working with NumPy

Importing NumPy

To start using NumPy in your script, you have to import it.

[82]:
```
import numpy as np
```

Converting Arrays to NumPy Arrays

You can convert your existing Python lists into NumPy arrays using the np.array() method, like this:

[83]:
```
arr = [1,2,3]
np.array(arr)
```

[83]:
```
array([1, 2, 3])
```

[84]:
```
# multidimentional arrays
nested_arr = [[1,2],[3,4],[5,6]]
np.array(nested_arr)
```

[84]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

NumPy arange Function

NumPy as an "arange()" method with which you can generate a range of values between two numbers. The arrange function takes the start, end, and an optional distance parameter.

[85]:
```
print(np.arange(0,10)) # without distance parameter
print(np.arange(0,10,2)) # with distance parameter
```

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

Zeroes and Ones

You can also generate an array or matrix of zeroes or ones using NumPy.

[86]:
```
print(np.zeros(3))
print(np.ones(3))
```

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

You can add the shape as a tuple with rows and columns.

[87]:
```
print(np.zeros((4,5)))
print(np.ones((4,5)))
```

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

You use ones_like and zeros_like

[88]:
```
x=np.zeros((4,5))
x0=np.zeros_like(x)
x1=np.ones_like(x)
print("x=\n",x)
print("x0=\n",x0)
print("x1=\n",x1)
```

```
x=
 [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
x0=
 [[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
x1=
 [[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

Identity Matrix

You can also generate an identity matrix using a built-in NumPy function called "eye".

[89]:
```
np.eye(5)
```

[89]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

NumPy Linspace Function

NumPy has a linspace method that generates evenly spaced points between two numbers.

[90]:
```
print(np.linspace(0,20.5,3))
```

```
[ 0.   10.25 20.5 ]
```

NumPy array data types

The elements in a numpy array must be of : * a basic type, e.g. integers or floats. * of the same type, so that they have the same size, e.g. 64 bits floats, or 16 bits integers.

On the contrary, python lists can contain heterogeneous objects of any type.

[91]:
```
# numpy guesses that it should use integers
x = np.array([0, 1, 2])
print(x.dtype)
# or floats:
x = np.array([0., 1., 2.])
print(x.dtype)
# here we specify a python compatible type,
# interpreted by numpy as int64
x = np.array([0., 1., 2.], dtype=int)
print(x.dtype)
# here we specify that we want 8 bits integers
x = np.array([0, 1, 2], dtype=np.int8)
print(x.dtype)
x = np.array([0, 1, 2], dtype=np.float32)
print(x.dtype)
```

```
int64
float64
int64
int8
float32
```

Random Number Generation * Normal Distribution : In a standard normal distribution, the values peak in the middle.(np.random.rand) * Uniform Distribution : The values in the distribution have the probability as a constant (np.random.randn)

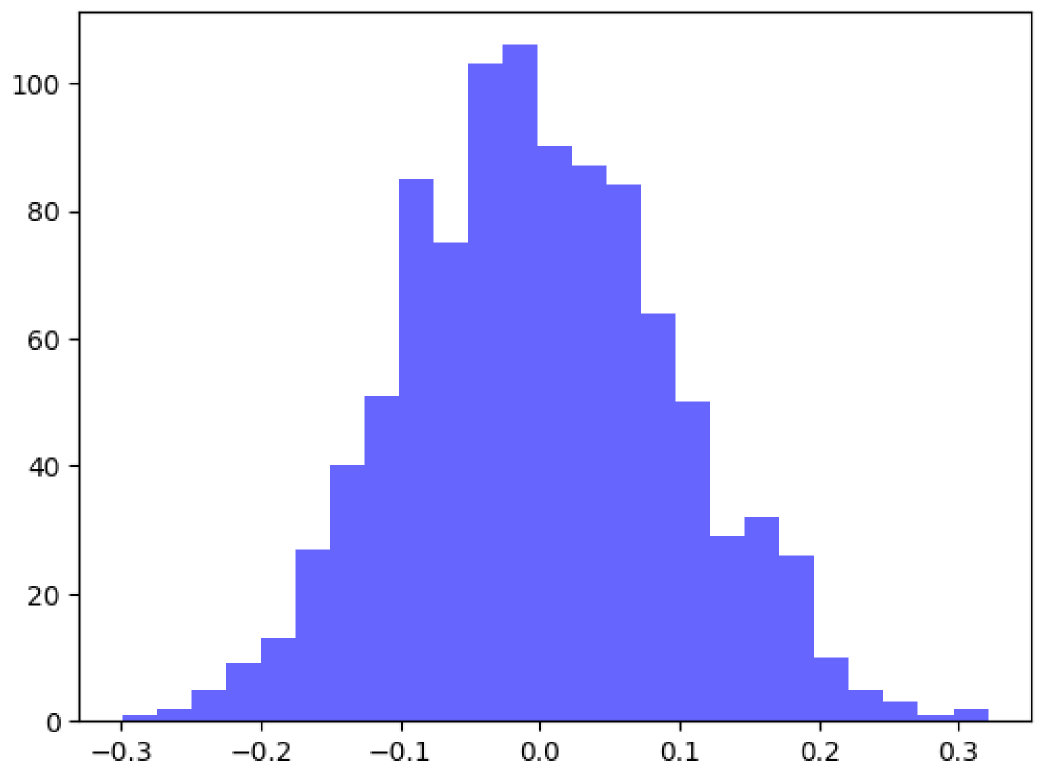
```
[92]: mu, sigma = 0, 0.1
      print(np.random.normal(loc=mu,scale=sigma,size=10)) # array
      print(np.random.normal(loc=mu, scale=sigma, size=(3,4))) # 3x4 matrix
```

```
[ 0.09354692  0.0079192  -0.0474285   0.04409397  0.06590567 -0.05572179
  0.17467228 -0.00253517 -0.01678586 -0.12822681]
[[-0.05064712  0.10340276  0.15131118 -0.14244719]
 [ 0.1279724   0.03194157  0.07181745  0.07642579]
 [-0.04120794  0.15442468 -0.13990768  0.07721351]]
```

```
[93]: print(np.random.rand(10)) # array
      print(np.random.rand(3,4)) # 3x4 matrix
```

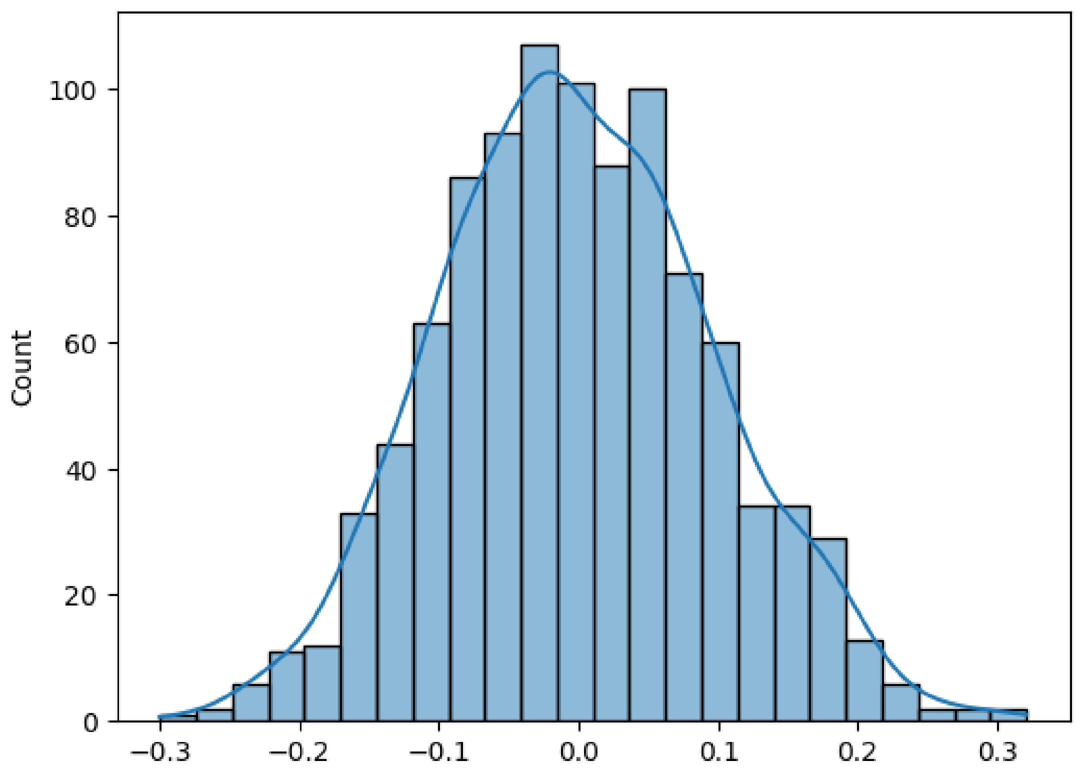
```
[0.93757401 0.62019758 0.25733381 0.72766097 0.8290075  0.54785725
 0.70585228 0.04267045 0.55297982 0.31938475]
[[0.64120693 0.72537179 0.11774118 0.32089429]
 [0.9432381  0.65462663 0.38837899 0.50451085]
 [0.42720208 0.84029076 0.14222176 0.84913406]]
```

```
[94]: import matplotlib.pyplot as plt
      data=np.random.normal(loc=mu,scale=sigma,size=1000)
      # Plotting the histogram.
      plt.hist(data, bins=25, density=False, alpha=0.6, color='b')
      plt.show()
```

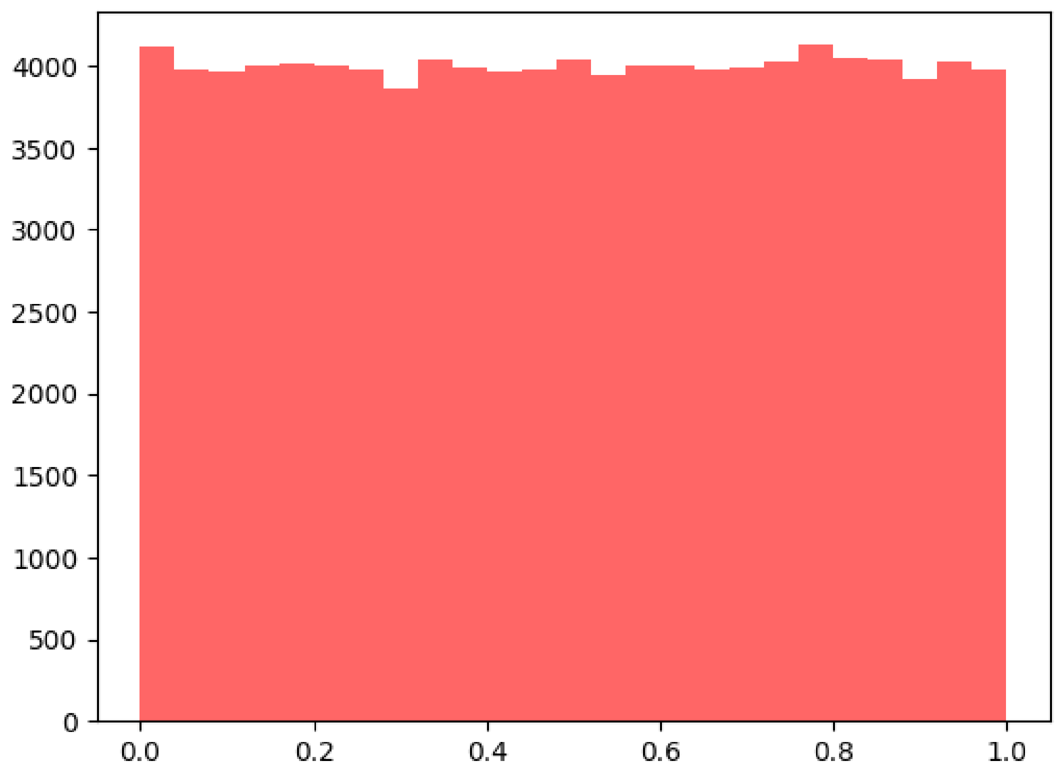


[95]:
```python
import seaborn as sns
sns.histplot(data=data, kde=True)
```

[95]:
```
<AxesSubplot: ylabel='Count'>
```



```
[96]: data=np.random.rand(100000)
      # Plotting the histogram.
      plt.hist(data, bins=25, density=False, alpha=0.6, color='r')
      plt.show()
```



Random integer Number Generation

[97]:
```
x = np.random.randint(0, 10, 20) # between 0 (included ) and 10 (not included)␣
  ↪, 20 values
print(x, "\n Type = ", x.dtype)
print(x)
```

```
[9 9 6 9 1 2 4 9 9 1 4 3 4 1 2 5 3 7 1 7]
 Type =  int64
[9 9 6 9 1 2 4 9 9 1 4 3 4 1 2 5 3 7 1 7]
```

0.0.2 2. Reshaping Arrays

To get the shape of an array, use the shape property.

[98]:
```
arr = np.random.rand(2,2)
print(arr)
print("\nShape=",arr.shape)
```

```
[[0.22642928 0.660517  ]
 [0.78565795 0.79988915]]

Shape= (2, 2)
```

To reshape an array, use the reshape() function.

```
[99]: arr = np.random.rand(2,2)
      print("arr\n",arr)
      print("A\n",arr.reshape(1,4))
      print("arr\n",arr)
      print("B\n",arr.reshape(4,1))
```

```
arr
 [[0.89338715 0.6819792 ]
 [0.11582469 0.17289587]]
A
 [[0.89338715 0.6819792  0.11582469 0.17289587]]
arr
 [[0.89338715 0.6819792 ]
 [0.11582469 0.17289587]]
B
 [[0.89338715]
 [0.6819792 ]
 [0.11582469]
 [0.17289587]]
```

```
[100]: arr = np.random.rand(2,2)
       x=arr.reshape(1,4)
       x[0,0]=1
       print("arr\n",arr)
```

```
arr
 [[1.         0.94515116]
 [0.01410088 0.39791913]]
```

Making an array one-dimensional

```
[101]: x = np.random.rand(2,2)
       y = x.flatten() # flatten always returns a copy of the array.
       print(y)
       y[0]=1
       print("x=",x)
```

```
[0.55488201 0.01276147 0.80562648 0.19386902]
x= [[0.55488201 0.01276147]
 [0.80562648 0.19386902]]
```

```
[102]: x = np.random.rand(2,2)
       y=x.ravel() # return a view on the original array, and only copies the data␣
        ↪when it's needed.
       print(y)
       y[0]=1
       print(x)
```

```
[0.84693481 0.2859682  0.19311172 0.7111882 ]
[[1.         0.2859682 ]
 [0.19311172 0.7111882 ]]
```

[103]:
```
x = np.random.rand(2,2)
y=x.reshape(-1) # like ravel : return a view on the original array, and only␣
  ↪copies the data when it's needed.
print(y)
y[0]=1
print(x)
```

```
[0.68955598 0.83682312 0.66087448 0.39836081]
[[1.         0.83682312]
 [0.66087448 0.39836081]]
```

[104]:
```
x = np.random.rand(2,2)
y=x.reshape(-1).copy() # like ravel : return a view on the original array, and␣
  ↪only copies the data when it's needed.
print(y)
y[0]=1
print(x)
```

```
[0.42116935 0.85535549 0.24161719 0.2256109 ]
[[0.42116935 0.85535549]
 [0.24161719 0.2256109 ]]
```

0.0.3 3. Numpy array indexing

Basic indexing

[105]:
```
x = np.arange(10) + 1
print("x=",x)
print("x[1]=",x[1])
print("x[-2]=",x[-2])
x[1]=0
print("After x[1] = 0; x=",x)
```

```
x= [ 1  2  3  4  5  6  7  8  9 10]
x[1]= 2
x[-2]= 9
After x[1] = 0; x= [ 1  0  3  4  5  6  7  8  9 10]
```

[106]:
```
x = np.zeros((2,3))
print("x=",x)
x[0,1] = 1
print("After x[0,1] = 1 \nx=",x)
```

```
x= [[0. 0. 0.]
 [0. 0. 0.]]
After x[0,1] = 1
```

```
x= [[0. 1. 0.]
 [0. 0. 0.]]
```

Selection with boolean indexing

[107]:
```
x = np.arange(10) + 1
print("x=",x)
# create a mask :a boolean expression for each element
x%2 == 0
```

```
x= [ 1  2  3  4  5  6  7  8  9 10]
```

[107]:
```
array([False,  True, False,  True, False,  True, False,  True, False,
        True])
```

[108]:
```
mask=x%2==0
print(x)
print("x[mask]=\n",x[mask])
# or simply
print("x[x%2==0]=\n",x[x%2==0])
```

```
[ 1  2  3  4  5  6  7  8  9 10]
x[mask]=
 [ 2  4  6  8 10]
x[x%2==0]=
 [ 2  4  6  8 10]
```

[109]:
```
x[x%2==0]=-1
print("x\n",x)
```

```
x
 [ 1 -1  3 -1  5 -1  7 -1  9 -1]
```

0.0.4 4. Slicing Data

To slice an array

[110]:
```
myarr = np.arange(0,11)
print(myarr)
sliced = myarr[0:5]
print(sliced)

sliced[:] = 99
print("sliced=\n",sliced)

print("myarr=\n",myarr)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10]
[0 1 2 3 4]
sliced=
 [99 99 99 99 99]
```

```
myarr=
 [99 99 99 99 99  5  6  7  8  9 10]
```

If you look at the above example, even though we assigned the slice of "myarr" to the variable "sliced", changing the value of "sliced" affects the original array. This is because the "slice" was just pointing to the original array.

To make an independent section of an array, use the copy() function.

[111]:
```
myarr = np.arange(0,11)
print(myarr)
sliced = myarr.copy()[0:5]
sliced[:]=99
print("sliced=myarr.copy\n",sliced)

print("myarr=\n",myarr)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10]
sliced=myarr.copy
 [99 99 99 99 99]
myarr=
 [ 0  1  2  3  4  5  6  7  8  9 10]
```

Slicing multi-dimensional arrays work similarly to one-dimensional arrays.

[112]:
```
my_matrix = np.random.randint(1,30,(3,3))
print(my_matrix)
print("single row=",my_matrix[0]) # print a single row
print("single value = ", my_matrix[0][0]) # print a single value or row 0,␣
 ↪column 0
print("single value = ",my_matrix[0,0]) #alternate way to print value from␣
 ↪row0,col0
print("single column=",my_matrix[:,0]) # print a single column
```

```
[[10 19 10]
 [28 23  8]
 [ 5 20  8]]
single row= [10 19 10]
single value =  10
single value =  10
single column= [10 28  5]
```

[113]:
```
print("2 columns=",my_matrix[:,0:2])
```

```
2 columns= [[10 19]
 [28 23]
 [ 5 20]]
```

[114]:
```
print("2 rows, 2 columns=",my_matrix[1:,0:2:1])
```

```
2 rows, 2 columns= [[28 23]
 [ 5 20]]
```

[115]:
```
print("my matrix \n",my_matrix)
print("reverse column=\n",my_matrix[:,::-1]) # reverse column order, by␣
↪specifying a -1 step on the last dimension (:: not :)
```

```
my matrix
 [[10 19 10]
 [28 23  8]
 [ 5 20  8]]
reverse column=
 [[10 19 10]
 [ 8 23 28]
 [ 8 20  5]]
```

0.0.5 5. Array Computations

few basic operations.

[116]:
```
new_arr = np.arange(1,11)
print(new_arr)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

[117]:
```
# addition
print(new_arr + 5)
```

```
[ 6  7  8  9 10 11 12 13 14 15]
```

[118]:
```
# Subtraction
print(new_arr - 5)
```

```
[-4 -3 -2 -1  0  1  2  3  4  5]
```

[119]:
```
# Array Addition
print(new_arr + new_arr)
```

```
[ 2  4  6  8 10 12 14 16 18 20]
```

[120]:
```
# Division
print(new_arr /5.0)
# Array Addition
print(new_arr / new_arr) # For zero division errors, Numpy will convert the␣
↪value to NaN (not a number).
```

```
[0.2 0.4 0.6 0.8 1.  1.2 1.4 1.6 1.8 2. ]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

few basic functions.

[121]:
```
new_arr = np.arange(1,11)
print(new_arr)
print("sum=",np.sum(new_arr))
print("mean=",np.mean(new_arr))
print("variance=",np.var(new_arr))
print("standard deviation=",np.std(new_arr))
print("sqrt=",np.sqrt(new_arr))
```

```
[ 1  2  3  4  5  6  7  8  9 10]
sum= 55
mean= 5.5
variance= 8.25
standard deviation= 2.8722813232690143
sqrt= [1.         1.41421356 1.73205081 2.         2.23606798 2.44948974
 2.64575131 2.82842712 3.         3.16227766]
```

working with 2d arrays.

We will often need to calculate row wise or column-wise sum, mean, variance, and so on. You can use the optional axis parameter to specify if you want to choose a row or a column.

[122]:
```
arr2d = np.arange(25).reshape(5,5)
print(arr2d)
print("\nSum of all elements =")
print(arr2d.sum())
print("\nSums of columns\t", end=" ")
print(arr2d.sum(axis=0))    # sum of columns
                            # axis=0 is the vertical axis
                            # axis=1 is the horisontal axis
print("\nSums of rows\t", end=" ")
print(arr2d.sum(axis=1)) #sum of rows
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]

Sum of all elements =
300

Sums of columns  [50 55 60 65 70]

Sums of rows     [ 10  35  60  85 110]
```

[ ]:
```

```