Numpy

March 30, 2022

Ejecuta esta linea de código para descargar los datos necesarios para correr el notebook

[1]:
```
!wget  https://raw.githubusercontent.com/cosmolejo/dataRepo/master/Pokemon.csv
```

```
--2022-03-17 21:20:16--
https://raw.githubusercontent.com/cosmolejo/dataRepo/master/Pokemon.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
185.199.108.133, 185.199.109.133, 185.199.110.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 44028 (43K) [text/plain]
Saving to: ‘Pokemon.csv’

Pokemon.csv         100%[===================>]  43.00K  --.-KB/s    in 0.001s

2022-03-17 21:20:16 (38.8 MB/s) - ‘Pokemon.csv’ saved [44028/44028]
```

#Introducción a Numpy

Este notebook fue tomado de la plataforma kagle una comunidad en línea de científicos de datos y profesionales del aprendizaje automático. En ella encontrarán distintos tutoriales, bancos de datos y competencias remuneradas en ciencias de datos.

Creditos a ***ABDULLAH SAHIN***

1 Introduction

NumPy is a basic package for scientific computing. It is a **Python** language implementation which includes:

- The powerful N-dimensional array structure
- Sophisticated functions
- Tools that can be integrated into C/C++ and Fortran code
- Linear algebra, Fourier transform and random number features

In addition to being used for scientific computing, NumPy also can be used as an efficient multi-dimensional container for general data. Because it can work with any type of data, NumPy can be integrated into multiple types of databases seamlessly and efficiently.

- If you like it, thank you for you **upvotes**.
- If you have any **question**, I will happy to hear it

1. ndarray
2. Create a specific array
3. Shape and operation
4. Index
5. Mathematics
6. Matrix
7. Random Number
8. Conclusion
9. Reference

[2]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
```

Basic Porperties and array creation

Here is an array with rank 1, and the length of the axis is 3:

[3]:
```
[1,2,3]
```

[3]:
```
[1, 2, 3]
```

Below is an array with rank 2, and the length of the axis is 3 too:

[4]:
```
[[ 1, 2, 3],[ 4, 5, 6]]
```

[4]:
```
[[1, 2, 3], [4, 5, 6]]
```

We can create an array of NumPy through the array function, for example:

[5]:
```
a = np.array([1, 2, 3])
b = np.array([(1,2,3), (4,5,6)])

print("a: ",a)
print("b: ",b)
```

```
a:  [1 2 3]
b:  [[1 2 3]
 [4 5 6]]
```

Note that the square brackets are required here. And the following way of writing is wrong:

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

NumPy's array class is **ndarray**, which has an alias **numpy.array**, but it's different from **array.array** in the Python standard library. The latter is just a one-dimensional array. The features of **ndarray** are as follows: Return Contents

- **ndarray.ndim:** the dimension number of the array. It's called rank in Python.

- **ndarray.shape:** the dimension of the array. It's a series of numbers whose length is determined by the dimension ndim of the array. For example, the shape of a one-dimensional array with length n is n. And the shape of an array with n rows and m columns is n,m.
- **ndarray.size:** the number of all elements in the array.
- **ndarray.dtype:** the type of the element in the array, such as numpy.int32, numpy.int16, or numpy.float64.
- **ndarray.itemsize:** the size of each element in the array, in bytes.
- **ndarray.data:** the buffering for storing the array elements. Usually we only need to access the elements by subscripts, and don't need to access the buffer.

Let's take a look at the code example:

[6]:
```
a = np.array([1, 2, 3])
b = np.array([(1,2,3), (4,5,6)])

print('a=')
print(a)
print("a's ndim {}".format(a.ndim))
print("a's shape {}".format(a.shape))
print("a's size {}".format(a.size))
print("a's dtype {}".format(a.dtype))
print("a's itemsize {}".format(a.itemsize))

print('')

print('b=')
print(b)
print("b's ndim {}".format(b.ndim))
print("b's shape {}".format(b.shape))
print("b's size {}".format(b.size))
print("b's dtype {}".format(b.dtype))
print("b's itemsize {}".format(b.itemsize))
```

```
a=
[1 2 3]
a's ndim 1
a's shape (3,)
a's size 3
a's dtype int64
a's itemsize 8

b=
[[1 2 3]
 [4 5 6]]
b's ndim 2
b's shape (2, 3)
b's size 6
b's dtype int64
```

```
b's itemsize 8
```

We can also specify the type of the element when creating the array, for example:

[7]:
```
c = np.array( [ [1,2], [3,4] ], dtype=complex )
c
```

[7]:
```
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

1.1 Create a specific array

Return Contents

In actual project engineering, we often need some specific data, and some helper functions are provided in NumPy:

- **zeros:** used to create an array whose elements are all 0
- **ones:** used to create an array whose elements are all 1
- **empty:** used to create uninitialized data. so the content is undefined.
- **arange:** used to create an array by specifying the scope and step-length
- **linespace:** used to create an array by specifying the range and the number of elements
- **random:** used to generate random numbers

[ ]:
```
a = np.zeros((2,3))
print('np.zeros((2,3)= \n{}\n'.format(a))

b = np.ones((2,3))
print('np.ones((2,3))= \n{}\n'.format(b))

c = np.empty((2,3))
print('np.empty((2,3))= \n{}\n'.format(c))

d = np.arange(1, 2, 0.3)
print('np.arange(1, 2, 0.3)= \n{}\n'.format(d))

e = np.linspace(1, 2, 7)
print('np.linspace(1, 2, 7)= \n{}\n'.format(e))

f = np.random.random((2,3))
print('np.random.random((2,3))= \n{}\n'.format(f))
```

```
np.zeros((2,3)=
[[0. 0. 0.]
 [0. 0. 0.]]

np.ones((2,3))=
[[1. 1. 1.]
 [1. 1. 1.]]
```

```
np.empty((2,3))=
[[1. 1. 1.]
 [1. 1. 1.]]

np.arange(1, 2, 0.3)=
[1.  1.3 1.6 1.9]

np.linspace(1, 2, 7)=
[1.         1.16666667 1.33333333 1.5        1.66666667 1.83333333
 2.        ]

np.random.random((2,3))=
[[0.57804621 0.80309513 0.98162694]
 [0.31719156 0.52288142 0.95117676]]
```

1.2 Shape and operation

Return Contents

In addition to generating an array, after we have held some data, we may need to generate some new data structures based on the existing array. In this case, we can use the following functions:

- **reshape:** used to generate a new array based on the existing array and the specified shape
- **vstack:** used to stack multiple arrays in vertical direction (the dimensions of the array must be matched)
- **hstack:** used to stack multiple arrays in horizontal direction (the dimensions of the array must be matched)
- **hsplit:** used to split the array horizontally
- **vsplit:** used to split the array vertically

We'll use some examples to illustrate.

To make it easier to test, let's create a few data:

- **zero_line:** an array with a row containing three 0
- **one_column:** an array with a column containing three 1
- **a:** a matrix with 2 rows and 3 columns
- **b:** an integer array in the interval of [11,20]

[ ]:
```
zero_line = np.zeros((1,3))
one_column = np.ones((3,1))
print("zero_line = \n{}\n".format(zero_line))
print("one_column = \n{}\n".format(one_column))

a = np.array([(1,2,3), (4,5,6)])
b = np.arange(11, 20)
print("a = \n{}\n".format(a))
print("b = \n{}\n".format(b))
```

```
zero_line =
[[0. 0. 0.]]

one_column =
[[1.]
 [1.]
 [1.]]

a =
[[1 2 3]
 [4 5 6]]

b =
[11 12 13 14 15 16 17 18 19]
```

The array b is a one-dimensional array originally, and we resize it into a matrix of 3 rows and 3 columns by the reshape method:

[ ]:
```
b = b.reshape(3, -1)
print("b.reshape(3, -1) = \n{}\n".format(b))
```

```
b.reshape(3, -1) =
[[11 12 13]
 [14 15 16]
 [17 18 19]]
```

The second parameter here is set to -1, which means that it'll be determined based on actual conditions automatically. Since the array has 9 elements originally, the matrix after being resized is 3X3. The code output is as follows:

[ ]:
```
b.reshape(3, -1)
```

[ ]:
```
array([[11, 12, 13],
       [14, 15, 16],
       [17, 18, 19]])
```

Next, we'll stack the three arrays vertically through the vstack function:

[ ]:
```
c = np.vstack((a, b, zero_line))
print("c = np.vstack((a,b, zero_line)) = \n{}\n".format(c))
```

```
c = np.vstack((a,b, zero_line)) =
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [11. 12. 13.]
 [14. 15. 16.]
 [17. 18. 19.]
```

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

Similarly, we can also use the hstack for horizontal stacking. This time we need to adjust the structure of the array a first:

[ ]:
```
a = a.reshape(3, 2)
print("a.reshape(3, 2) = \n{}\n".format(a))

d = np.hstack((a, b, one_column))
print("d = np.hstack((a,b, one_column)) = \n{}\n".format(d))
```

```
a.reshape(3, 2) =
[[1 2]
 [3 4]
 [5 6]]

d = np.hstack((a,b, one_column)) =
[[ 1.  2. 11. 12. 13.  1.]
 [ 3.  4. 14. 15. 16.  1.]
 [ 5.  6. 17. 18. 19.  1.]]
```

Next, let's take a look at the split. First, we split the array d into three arrays in horizontal direction. Then we print out the middle one (the subscript is 1):

[ ]:
```
e = np.hsplit(d, 3) # Split a into 3
print("e = np.hsplit(d, 3) = \n{}\n".format(e))
print("e[1] = \n{}\n".format(e[1]))
```

```
e = np.hsplit(d, 3) =
[array([[1., 2.],
       [3., 4.],
       [5., 6.]]), array([[11., 12.],
       [14., 15.],
       [17., 18.]]), array([[13.,  1.],
       [16.,  1.],
       [19.,  1.]])]

e[1] =
[[11. 12.]
 [14. 15.]
 [17. 18.]]
```

In addition to specifying number to split the array evenly, we can also specify the number of columns to split. The following is to split the array d from the first column and the third column:

[ ]:
```
f = np.hsplit(d, (1, 3)) # # Split a after the 1st and the 3rd column
print("f = np.hsplit(d, (1, 3)) = \n{}\n".format(f))
```

```
f = np.hsplit(d, (1, 3)) =
[array([[1.],
       [3.],
       [5.]]), array([[ 2., 11.],
       [ 4., 14.],
       [ 6., 17.]]), array([[12., 13.,  1.],
       [15., 16.,  1.],
       [18., 19.,  1.]])]
```

Finally, we split the array d in the vertical direction. Similarly, if the specified number cannot make the array be split evenly, it will fail:

[ ]:
```
g = np.vsplit(d, 3)
print("np.hsplit(d, 2) = \n{}\n".format(g))

# np.vsplit(d, 2) # ValueError: array split does not result in an equal division
```

```
np.hsplit(d, 2) =
[array([[ 1.,  2., 11., 12., 13.,  1.]]), array([[ 3.,  4., 14., 15., 16.,
1.]]), array([[ 5.,  6., 17., 18., 19.,  1.]])]
```

1.3 Index

Return Contents

Next we look at how to access the data in the NumPy array.

Again, for testing convenience, let's create a one-dimensional array first. Its content is integers in the interval of [100,200).

Basically, we can specify the subscripts by array[index] to access the elements of the array.

[ ]:
```
base_data = np.arange(100, 200)
print("base_data\n={}\n".format(base_data))

print("base_data[10] = {}\n".format(base_data[10]))
```

```
base_data
=[100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117
 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135
 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153
 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171
 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
 190 191 192 193 194 195 196 197 198 199]

base_data[10] = 110
```

In NumPy, we can create an array containing several subscripts to get the elements in the target array. For example:

[ ]:
```
every_five = np.arange(0, 100, 5)
print("base_data[every_five] = \n{}\n".format(
    base_data[every_five]))
```

```
base_data[every_five] =
[100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175 180 185
 190 195]
```

The subscript array can be one-dimensional, or multi-dimensional. Let's suppose that we want to get a 2X2 matrix whose content comes from the four subscripts of 1, 2, 10, and 20 in the target array, so the code can be written:

[ ]:
```
a = np.array([(1,2), (10,20)])
print("a = \n{}\n".format(a))
print("base_data[a] = \n{}\n".format(base_data[a]))
```

```
a =
[[ 1  2]
 [10 20]]

base_data[a] =
[[101 102]
 [110 120]]
```

The above we see is the case where the target array is one-dimensional. Let's convert the following array into a 10X10 two-dimensional array.

[ ]:
```
base_data2 = base_data.reshape(10, -1)
print("base_data2 = np.reshape(base_data, (10, -1)) = \n{}\n".
 ↪format(base_data2))
```

```
base_data2 = np.reshape(base_data, (10, -1)) =
[[100 101 102 103 104 105 106 107 108 109]
 [110 111 112 113 114 115 116 117 118 119]
 [120 121 122 123 124 125 126 127 128 129]
 [130 131 132 133 134 135 136 137 138 139]
 [140 141 142 143 144 145 146 147 148 149]
 [150 151 152 153 154 155 156 157 158 159]
 [160 161 162 163 164 165 166 167 168 169]
 [170 171 172 173 174 175 176 177 178 179]
 [180 181 182 183 184 185 186 187 188 189]
 [190 191 192 193 194 195 196 197 198 199]]
```

For a two-dimensional array,

- if we only specify one subscript, the result of the access is still an array.
- if we specify two subscripts, the result of the access is the elements inside.
- we can also specify the last element by "-1".

[ ]:
```
print("base_data2[2] = \n{}\n".format(base_data2[2]))
print("base_data2[2, 3] = \n{}\n".format(base_data2[2, 3]))
print("base_data2[-1, -1] = \n{}\n".format(base_data2[-1, -1]))
```

```
base_data2[2] =
[120 121 122 123 124 125 126 127 128 129]

base_data2[2, 3] =
123

base_data2[-1, -1] =
199
```

In addition, we can also specify the scope by ":", such as: 2:5 . Only to write ":" indicates the full scope.

Please see the code below:

It will:

- get all the elements of the row whose subscript is 2
- get all the elements of the column whose subscript is 3
- get all the elements of the rows whose subscripts are in [2,5) and the columns * whose subscripts are in [2,4). Please observe the following output carefully:

[ ]:
```
print("base_data2[2, :]] = \n{}\n".format(base_data2[2, :]))
print("base_data2[:, 3]] = \n{}\n".format(base_data2[:, 3]))
print("base_data2[2:5, 2:4]] = \n{}\n".format(base_data2[2:5, 2:4]))
```

```
base_data2[2, :]] =
[120 121 122 123 124 125 126 127 128 129]

base_data2[:, 3]] =
[103 113 123 133 143 153 163 173 183 193]

base_data2[2:5, 2:4]] =
[[122 123]
 [132 133]
 [142 143]]
```

Mathematics Return Contents

There are also a lot of mathematical functions in NumPy. Here are some examples.

[ ]:
```
base_data = (np.random.random((5, 5)) - 0.5) * 100
print("base_data = \n{}\n".format(base_data))

print("np.amin(base_data) = {}".format(np.amin(base_data)))
print("np.amax(base_data) = {}".format(np.amax(base_data)))
print("np.average(base_data) = {}".format(np.average(base_data)))
print("np.sum(base_data) = {}".format(np.sum(base_data)))
print("np.sin(base_data) = \n{}".format(np.sin(base_data)))
```

```
base_data =
[[ 46.19324186  48.86135528 -25.26785545 -45.93230239  26.49102785]
 [ 23.59589215  47.9289554  -12.43078937  -5.66174324  32.11854558]
 [ 12.08463332 -27.67307239  31.1112311  -24.82263157  18.83736958]
 [ 25.99503038 -28.6865881    9.85929279  43.22050001  -0.2764504 ]
 [ -1.21159163  42.26013737 -38.90095056  17.17136133  -1.50876002]]

np.amin(base_data) = -45.93230239229113
np.amax(base_data) = 48.861355284657606
np.average(base_data) = 8.53423355587606
np.sum(base_data) = 213.3558388969015
np.sin(base_data) =
[[ 0.80200713 -0.98614282 -0.1347035  -0.92895777  0.97750466]
 [-0.99942385 -0.72087611  0.13516624  0.58220822  0.64621863]
 [-0.46331945 -0.56568318 -0.30000265  0.30516306 -0.01218604]
 [ 0.75933411  0.40067571 -0.42097037 -0.69022296 -0.27294255]
 [-0.93617669 -0.98856641 -0.93270379 -0.99423835 -0.99807637]]
```

[ ]:
```
arr = np.arange(1,20)
arr = arr * arr              #Multiplies each element by itself
print("Multpiles: ",arr)
arr = arr - arr              #Substracts each element from itself
print("Substracts: ",arr)
arr = np.arange(1,20)
arr = arr + arr              #Adds each element to itself
print("Add: ",arr)
arr = arr / arr              #Divides each element by itself
print("Divide: ",arr)
arr = np.arange(1,20)
arr = arr + 50
print("Add +50: ",arr)
```

```
Multpiles:  [  1   4   9  16  25  36  49  64  81 100 121 144 169 196 225 256 289
324
 361]
Substracts:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
Add:  [ 2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38]
Divide:  [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
Add +50:  [51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69]
```

[ ]:
```
print("Sqrt: ",np.sqrt(arr))#Returns the square root of each element
print("Exp: ",np.exp(arr))     #Returns the exponentials of each element
print("Sin: ",np.sin(arr))     #Returns the sin of each element
print("Cos: ",np.cos(arr))     #Returns the cosine of each element
print("Log: ",np.log(arr))     #Returns the logarithm of each element
print("Sum: ",np.sum(arr))     #Returns the sum total of elements in the array
print("Std: ",np.std(arr))     #Returns the standard deviation of in the array
```

```
Sqrt:  [7.14142843 7.21110255 7.28010989 7.34846923 7.41619849 7.48331477
 7.54983444 7.61577311 7.68114575 7.74596669 7.81024968 7.87400787
 7.93725393 8.         8.06225775 8.1240384  8.18535277 8.24621125
 8.30662386]
Exp:  [1.40934908e+22 3.83100800e+22 1.04137594e+23 2.83075330e+23
 7.69478527e+23 2.09165950e+24 5.68572000e+24 1.54553894e+25
 4.20121040e+25 1.14200739e+26 3.10429794e+26 8.43835667e+26
 2.29378316e+27 6.23514908e+27 1.69488924e+28 4.60718663e+28
 1.25236317e+29 3.40427605e+29 9.25378173e+29]
Sin:  [ 0.67022918  0.98662759  0.39592515 -0.55878905 -0.99975517 -0.521551
  0.43616476  0.99287265  0.63673801 -0.30481062 -0.96611777 -0.7391807
  0.1673557   0.92002604  0.82682868 -0.02655115 -0.85551998 -0.89792768
 -0.11478481]
Cos:  [ 0.7421542  -0.16299078 -0.91828279 -0.82930983  0.02212676  0.85322011
  0.89986683  0.11918014 -0.77108022 -0.95241298 -0.25810164  0.67350716
  0.98589658  0.39185723 -0.56245385 -0.99964746 -0.5177698   0.44014302
  0.99339038]
Log:  [3.93182563 3.95124372 3.97029191 3.98898405 4.00733319 4.02535169
 4.04305127 4.06044301 4.07753744 4.09434456 4.11087386 4.12713439
 4.14313473 4.15888308 4.17438727 4.18965474 4.20469262 4.21950771
 4.2341065 ]
Sum:  1140
Std:  5.477225575051661
```

1.4 Matrix

Return Contents

Now, let's take a look at how to use NumPy in a matrix way.

First, let's create a 5X5 random integer matrix. There are two ways to get the transpose of a matrix: **.T** or **transpose** function. In addition, the matrix can be multiplied through the **dot** function. The sample code is as follows:

[ ]:
```
base_data = np.floor((np.random.random((5, 5)) - 0.5) * 100)
print("base_data = \n{}\n".format(base_data))

print("base_data.T = \n{}\n".format(base_data.T))
print("base_data.transpose() = \n{}\n".format(base_data.transpose()))

matrix_one = np.ones((5, 5))
```

```
print("matrix_one = \n{}\n".format(matrix_one))

minus_one = np.dot(matrix_one, -1)
print("minus_one = \n{}\n".format(minus_one))

print("np.dot(base_data, minus_one) = \n{}\n".format(
    np.dot(base_data, minus_one)))
```

```
base_data =
[[ 27. -17.  39.  15.  40.]
 [ 24.  13. -41.  44.  -8.]
 [ 12. -16.  19. -38.   5.]
 [-31. -44.  14.  17. -38.]
 [-25.   9. -31.   0. -15.]]

base_data.T =
[[ 27.  24.  12. -31. -25.]
 [-17.  13. -16. -44.   9.]
 [ 39. -41.  19.  14. -31.]
 [ 15.  44. -38.  17.   0.]
 [ 40.  -8.   5. -38. -15.]]

base_data.transpose() =
[[ 27.  24.  12. -31. -25.]
 [-17.  13. -16. -44.   9.]
 [ 39. -41.  19.  14. -31.]
 [ 15.  44. -38.  17.   0.]
 [ 40.  -8.   5. -38. -15.]]

matrix_one =
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]

minus_one =
[[-1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1.]]

np.dot(base_data, minus_one) =
[[-104. -104. -104. -104. -104.]
 [ -32.  -32.  -32.  -32.  -32.]
 [  18.   18.   18.   18.   18.]
```

```
 [  82.   82.   82.   82.   82.]
 [  62.   62.   62.   62.   62.]]
```

1.5 Random Number

Return Contents

At the end of the article, let's take a look at the use of random numbers.

Random numbers are a feature we use very often during the programming process, such as generating demo data, or disordering existing data sequence randomly to segment the modeling data and the verification data.

The numpy.random package contains a number of algorithms for random numbers. Here we list the four most common usage:

The four usages are:

- to generate 20 random numbers, each of which is between **[0.0, 1.0)**
- to generate a random number based on the specified **shape**
- to generate a specified number (such as 20) of random integers within the specified range (such as **[0, 100)**)
- to disorder the sequence of the existing data (**[0, 1, 2, ..., 19]**) randomly The output is as follows:

[8]:
```
print("random: {}\n".format(np.random.random(20)));

print("rand: {}\n".format(np.random.rand(3, 4)));

print("randint: {}\n".format(np.random.randint(0, 100, 20)));

print("permutation: {}\n".format(np.random.permutation(np.arange(20))));
```

```
random: [0.215081   0.57735915 0.20811259 0.37881527 0.82054122 0.30472902
 0.76276674 0.94424167 0.13843456 0.85759865 0.4965837  0.77363327
 0.54794253 0.74841182 0.15544318 0.05927336 0.82318256 0.1817038
 0.01359952 0.83053727]

rand: [[0.85233414 0.07414227 0.90252677 0.33360404]
 [0.16110139 0.71985488 0.04608619 0.84776958]
 [0.8069466  0.32312964 0.21395472 0.17265418]]

randint: [22 47 70 41 63 42 11 58 18 53 17 60 77 61 66 14  8 64 39 14]

permutation: [11 10  5  2 17  9  6 12 14 18  8  3 13  0 19 15  4  1  7 16]
```

1.6 Conclusion

Return Contents

- If you like it, thank you for you upvotes.
- If you have any question, I will happy to hear it

1.7 Reference

Return Contents

- https://www.tutorialdocs.com/article/python-numpy-tutorial.html
- https://towardsdatascience.com/lets-talk-about-numpy-for-datascience-beginners-b8088722309f
- http://www.numpy.org