numpy

1 Numeric data manipulation, taublar dataframe and data visualization

2 Introduction to numpy

Numpy is a framework for working with numerical data. It is designed to be fast and memory efficient. Numpy is widely used amongst python libraries

[52]:
```
# first thing you have to do is to import the library
import numpy as np  # np is an alias
```

2.0.1 Why numpy?

Because it's faster, memory efficient and provides lot of apis to facilitate the work with vector data

[5]:
```
# simple comparison between numpy and lists
# import the time library to measure the performance
import time

# generate a list of numbers between 1 and 1_000_000
rand_nums_py = range(1_000_000)

# generate a list of numbers between 1 and 1_000_000 with numpy
rand_nums_numpy = np.arange(1_000_000)

# compute the mean with pure python
start = time.time()
mean_py = sum(rand_nums_py) / len(rand_nums_py)
stop = time.time()
time_py = stop - start

# compute the mean with numpy
start = time.time()
mean_py = rand_nums_numpy.mean()
stop = time.time()
time_numpy = stop - start
```

```
print(f'Time execution with python: {time_py:.2e}s')
print(f'Time execution with numpy: {time_numpy:.2e}s')
print(f'Ratio of between times: {time_py / time_numpy:.2f}')
```

```
Time execution with python: 2.71e-02s
Time execution with numpy: 1.54e-03s
Ratio of between times: 17.58
```

2.1 Core features of numpy

We will start by looking at the api to create arrays, delete array, access element of arrays, delete element of arrays.

2.2 Arrays creation

Arrays can be created in different ways. Here are some examples:

[6]:
```
# create an array from an existing list, tuple or generator
l = [1, 2, 3]
t = [4, 5, 6]
g = range(7, 10)

a = np.array(l); print(a)
a = np.array(t); print(a)
a = np.array(g); print(a)
```

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

[7]:
```
%%writefile array_data.txt
# this is a comment in the file
x, y, z
1, 2, 3
1, 4, 5
2, 3, 5
```

```
Writing array_data.txt
```

[10]:
```
# read from txt file with comments and headers
w = np.genfromtxt('array_data.txt',  # filename
                  skip_header=2,  # number of rows to skip
                  delimiter=',', #type of value separator
                  unpack=True) # columns into single variables
print('Data read from .txt file')
print(x); print(y); print(z)
```

```
Data read from .txt file
```

```
[1. 1. 2.]
[2. 4. 3.]
[3. 5. 5.]
```

[21]:
```
# generate array from a particular function
a = np.arange(0, 1, 0.1) # from 0 to 1 with 0.1 steps
# print(a)
b = np.linspace(0, 5, 10) # from 0 to 5 such that there are 10 elements
# print(b)
c = np.empty(10) # 10 elements with random values inside
# print(c)
d = np.zeros(10) # 10 elements with 0s inside
# print(d)
e = np.full_like(d, 3) # array big like d but with 3 elements inside
# print(e)
f = np.random.random(10)  # array of random elements
# print(f)
```

2.3 Accessing elements of arrays

Accessing elements of array is similar to python syntax for lists but it can be extended to a more powerful behaviour

[22]:
```
a = np.arange(5, 15)  # 10 elements from 5 to 15 excluded
a
```

[22]:
```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

[24]:
```
first = a[0]  # access the first element
last = a[-1]  # access the last element
slice1 = a[:4] # access element from 0 to 4 excluded (0, 1, 2, 3)
slice2 = a[1:3] # access elements from 1 included to 3 excluded (1, 2)
slice3 = a[-3:-1]  # access element from the third last included to the last␣
 ↪exclued (-3, -2)
slice4 = a[4:]  # access elements from 4 included until the end
slice5 = a[1:8:2]  # access from 1 inclued to 8 excluded with steps of 2 (1, 3,␣
 ↪5, 7)
slice6 = a[::3] # access every third element (0, 3, 6, 9)
# bonus: reverse an array
slice7 = a[::-1]
```

[25]:
```
a
```

[25]:
```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

[26]:
```
mask = a > 8
mask
```

[26]:
```
array([False, False, False, False,  True,  True,  True,  True,  True,
        True])
```

[28]:
```
# more indexing methods
submask = [1, 3, 7, 9]
slice8 = a[submask]  # access the elements in the positions described by submask

# boolean indexing.
mask = a > 8  # mask is a variable of booleans
print(mask)
slice9 = a[mask]  # a boolean maks can be used to select only certain element
slice10 = a[a > 8]  # alternative syntax

print(slice9, slice10)
```

```
[False False False False  True  True  True  True  True  True]
[ 9 10 11 12 13 14] [ 9 10 11 12 13 14]
```

2.4 Some interesting numpy functions

2.4.1 Math functions

numpy has a fairly big set of functions to for mathematical operations

[32]:
```
# standard math functions, like mean, standard deviations, max, min, etc.
# are all available
a = np.array([10, 8, 9])
print('Mean: ', np.mean(a))
print('Standard deviation: ', np.std(a))
print('Max: ', np.max(a))
print('Min: ', np.min(a))
print('Ix Max: ', np.argmax(a))
print('Ix Min: ', np.argmin(a))
```

```
Mean:  9.0
Standard deviation:  0.816496580927726
Max:  10
Min:  8
Ix Max:  0
Ix Min:  1
```

2.4.2 Inserting and deleting elements

numpy are blocks of memory with a different layout than the list. For this reason it's not possible to append and element like a list, thoug similar functions are provided

[33]:
```
a = np.array([1, 2, 3])
```

```
b = np.append(a, 3); print(b) # insert the number 3 at the end
c = np.insert(a, 2, 12); print(c)  # insert the value 12 at index 2
d = np.delete(a, 1); print(d) # delete the value at index 1
```

```
[1 2 3 3]
[ 1  2 12  3]
[1 3]
```

2.5 Vectorization and broadcasting

These following two concepts are probably the most powerful concepts used in numpy. Sometimes they're used without we notice it. Let's consider the following examples

[35]:
```
a = np.array([1, 2, 3])

# I want the square of each element: there is the dedicated function
print(np.square(a))
# I want each element to the power of 6 for example:
print(np.power(a, 6)); print(a ** 6)
# I want to divide each element by 3
print(a / 3)
# I want to subtract 2 from each element
print(a - 2)
```

```
[1 4 9]
[  1  64 729]
[  1  64 729]
[0.33333333 0.66666667 1.        ]
[-1  0  1]
```

The operations between an array/vector and a scalar value are possible because numpy can smartly understand the dimension of both variables and apply the operator to each value of the first element using the value of the second.

[36]:
```
x = np.array([1, 1, 3])
y = np.array([2, 3, 3])

# I  want to mulitply each element of x by the correspndding element of y
print(x * y)
```

```
[2 3 9]
```

So if there are two array with the same dimension (1d in this case) the operation are applied element-wise. If there is a 1d array and a scalar then the scalr is applied to each element of the array.

2.6 Multi dimensional array

Sometimes we want to work with n-dimensional arrays. Let's see some examples

[13]:
```
# Create a 2d array 2x2 of random elements
m = np.random.random((2, 2))
m
```

[13]:
```
array([[0.02037515, 0.02675105],
       [0.71676358, 0.09473717]])
```

[38]:
```
# create a 4x6 array of elements from 1 to 24
m = np.arange(1, 25)
print(m)
m = m.reshape((4, 6))
m
```

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

[38]:
```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24]])
```

[39]:
```
# example of error: 25 elements are not divisible by 4 so
# I can't create a matrix of 4 rows
m = np.arange(0, 25)
m = m.reshape((4, 6))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-39-cce8bcffcf17> in <module>
      2 # I can't create a matrix of 4 rows
      3 m = np.arange(0, 25)
----> 4 m = m.reshape((4, 6))

ValueError: cannot reshape array of size 25 into shape (4,6)
```

[44]:
```
# sometimes we only need to reshape an array knowing the number of rows
# but we don't know the number od columns (or viceversa). In this case
# we can use -1 in the reshape function
m = np.arange(0, 60)
reshape1 = m.reshape((-1, 2)) # it will create a 30x2
reshape2 = m.reshape((2, -1)) # it will create a 2x30
reshape3 = m.reshape((2, 6, -1)) # it will create a 2x6x5
print(reshape3)
```

```
[[[ 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]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]
  [45 46 47 48 49]
  [50 51 52 53 54]
  [55 56 57 58 59]]]
```

2.7 The axis arguments

some numpy functions accepts an `axis` argument. This is used in multidimensional array to apply the operation over a particular axis. Let's see a simple example:

School grades There three students attending the physics class. Their marks over time are saved into a simple .txt file

[45]:
```
%%writefile school_grades.txt
tizio, caio, sempronio
9, 9, 10
8, 6, 9
10, 9, 9
7, 7, 8
```

```
Writing school_grades.txt
```

[46]:
```
# read the data into a single array
grades = np.genfromtxt('school_grades.txt', skip_header=1, delimiter=',')
grades
```

[46]:
```
array([[ 9.,  9., 10.],
       [ 8.,  6.,  9.],
       [10.,  9.,  9.],
       [ 7.,  7.,  8.]])
```

[47]:
```
# the shape of the array is
grades.shape
```

[47]:
```
(4, 3)
```

[48]:
```
# I want the global average mark for each student
mean_marks_per_student = np.mean(grades, axis=0)
mean_marks_per_student
```

[48]:
```
array([8.5 , 7.75, 9.  ])
```

[49]:
```
# I want the average mark between the students for each test
mean_marks_per_test = np.mean(grades, axis=1)
mean_marks_per_test
```

[49]:
```
array([9.33333333, 7.66666667, 9.33333333, 7.33333333])
```

So the array has a *4* rows and *3* columns. The shape can be accessed by the attribute `grades.shape` and returns a tuple `(nrows, ncolumns)`. The axis argument can be used to apply a function over an axis and it works this way:

- If the axis is 0 it will collpase the axis at the 0-th position in the shape of the array. This means from `(nrows, ncolumns)` -> `(ncolumns)`
- If the axis is 1 it will collapse the axis at the 1-st position in the shape of the array. This means that `(nrows, ncolumns)` -> `(nrows)`
- In general if we have an array of the shape `(s1, s2, s3, ...)` the axis is an integer number (or a tuple) from 0 to `ndim-1`. The number specified in the `axis` arguments will be the dimensions the will collapse.

```
a = np.array([...])
a.shape = (s1, s2, s3, s4)
           0,  1,  2,  3

np.mean(a, axis=0) -> (s2, s3, s4)
np.mean(a, axis=(0, 1)) -> (s3, s4)
np.mean(a, axis=3) -> (s1, s2, s3)
```

[51]:
```
a = np.random.random(420)
print(a.shape)
a = a.reshape((2, 3, 2, 5, 7))
#              0, 1, 2, 3, 4
print(a.shape)

print(a.mean(axis=1).shape) # 0, 2, 3, 4
print(a.mean(axis=0).shape) # 1, 2, 3, 4
print(a.mean(axis=(2, 3)).shape) # 0, 1, 4
print(a.mean(axis=-1).shape) # 0, 1, 2, 3
```

```
(420,)
(2, 3, 2, 5, 7)
(2, 2, 5, 7)
(3, 2, 5, 7)
(2, 3, 7)
(2, 3, 2, 5)
```