NumPy Arrays and Linear Algebra

Python 3

General comments

Assignments. Any issue with the last assignment? How long did it take you?

Plan.

0. Feedback on last week's exercises
1. NumPy Arrays
2. Operations on NumPy arrays
3. Simple uses of NumPy arrays →
   a. Simple data statistics
   b. Linear Algebra
   c. Random Arrays
4. Introduction to Matplotlib

Books.

Data Science Handbook, by Jake VanderPlas. Available online **here**.

Part 0
Last week's exercises

Do you have any question?

Is there anything you are not sure you have understood?

Last week's exercises

Comprehensions: >>> Q13

Exo 1. factoriel(n): Integers in numpy?

Exo 2. n choose k: Complexity?

Exo 4. Sampling:
— Choice of dx
— 1-line Sampling with list comprehensions

Exo 7. Cumulative distribution: — Integral from -infinity to 0?

Scientific notations: — Recall: Print with scientific notation

Int32 and int64

```
def factFn(n):
    fact=1
    for i in range(1,n+1):
        fact=fact*i
    return fact

import numpy as np
n=12

print("\n", factFn(n))
print(" 12!=", np.prod(np.arange(1,n+1, dtype='int32')))
print("2^31=", 2**31)

n=13
print("\n", factFn(n))
print(" 13!=", np.prod(np.arange(1,n+1, dtype='int32'),dtype='int32'))
print("2^31=", 2**31)
```

```
 479001600
 12!= 479001600
2^31= 2147483648

 6227020800
 13!= 1932053504
2^31= 2147483648
```

Largest *n* for which you can compute *n!* on 32 bits?

Largest *n* for which you can compute *n!* on 64 bits?

Int32 and int64

```
def factFn(n):
    fact=1
    for i in range(1,n+1):
        fact=fact*i
    return fact

import numpy as np
n=12

print("\n", factFn(n))
print(" 12!=", np.prod(np.arange(1,n+1, dtype='int32')))
print("2^31=", 2**31)

n=13
print("\n", factFn(n))
print(" 13!=", np.prod(np.arange(1,n+1, dtype='int32'),dtype='int32'))
print("2^31=", 2**31)
```

```
 479001600
 12!= 479001600
2^31= 2147483648

 6227020800
 13!= 1932053504
2^31= 2147483648
```

Largest *n* for which you can compute *n!* on 32 bits?

12

Largest *n* for which you can compute *n!* on 64 bits?

20

Algorithmic **Complexity**

Factoriel(n):

```
fact=1
for i in range(1,n+1):
    fact=fact*i
print(fact)
```

Algorithmic **Complexity**

Factoriel(n):

```
fact=1
for i in range(1,n+1):
    fact=fact*i
print(fact)
```

O(n) simple operations

Algorithmic **Complexity**

Factoriel(n):

```
fact=1
for i in range(1,n+1):
    fact=fact*i
print(fact)
```

O(n) simple operations

N_choose_K (N, K):

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Algorithmic **Complexity**

Factoriel(n):

```
fact=1
for i in range(1,n+1):
    fact=fact*i
print(fact)
```

O(n) simple operations

N_choose_K (N, K):

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

O(n) simple operations

$$\binom{n}{k} = \binom{n}{k-1}\frac{n-k+1}{k}$$

Algorithmic **Complexity**

Factoriel(n):

```
fact=1
for i in range(1,n+1):
    fact=fact*i
print(fact)
```

O(n) simple operations

N_choose_K (N, K):

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

O(n) simple operations

$$\binom{n}{k} = \binom{n}{k-1}\frac{n-k+1}{k}$$

```
def NchooseK(N,K):
    result=1
    for k in range(K):
        result=(result*(N-k))//(k+1)
    return(result)
```

O(K) simple operations

$$\binom{n}{k} = \binom{n}{n-k}$$

Algorithmic **Complexity**

Factoriel(n):

```
fact=1
for i in range(1,n+1):
    fact=fact*i
print(fact)
```

O(n) simple operations

N_choose_K (N, K):

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

O(n) simple operations

$$\binom{n}{k} = \binom{n}{k-1}\frac{n-k+1}{k}$$

```
def NchooseK(N,K):
    result=1
    for k in range(K):
        result=(result*(N-k))//(k+1)
    return(result)
```

O(K) simple operations

$$\binom{n}{k} = \binom{n}{n-k}$$

```
def NchooseK_bis(N,K):
    if K > N//2:
        return NchooseK(N,N-K)
    else:
        return NchooseK(N,K)
```

min(K, N-K) simple operations

Sampling

```
def sampling3(f, xmin, xmax, n):
    dx=(xmax-xmin)/(n-1)
    return [[xmin+i*dx, f(xmin+i*dx)] for i in range(n)]
```

n points ==> (n-1) intervals

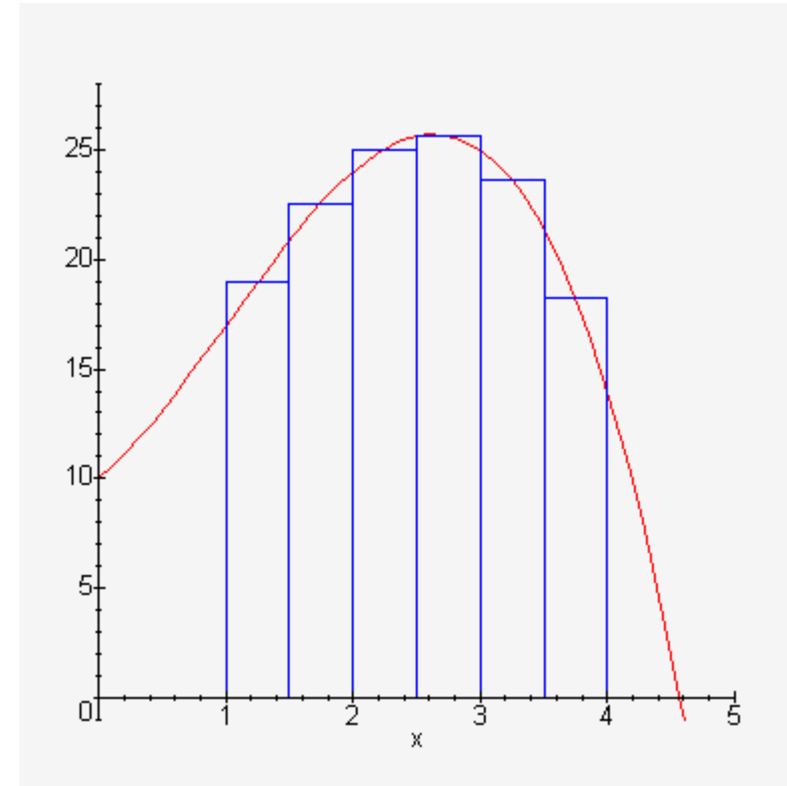
Sampling

```
def sampling3(f, xmin, xmax, n):
    dx=(xmax-xmin)/(n-1)
    return [[xmin+i*dx, f(xmin+i*dx)] for i in range(n)]
```

"1-line" sampling using comprehensions

Cumulative

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{t^2}{2}\right) \mathrm{d}t$$



$$\int_a^b f(x)\,\mathrm{d}x \simeq h \sum_{i=0}^{n-1} f\left(a + \left(i + \frac{1}{2}\right) h\right)$$

How to compute the integral from -infinity?

We use:

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{0} \exp\left(-\frac{t^2}{2}\right) \mathrm{d}t = \frac{1}{2}\left[\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} \exp\left(-\frac{t^2}{2}\right) \mathrm{d}t\right] = \frac{1}{2}$$

Which gives:

$$F(x) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \int_{0}^{x} \exp\left(-\frac{t^2}{2}\right) \mathrm{d}t$$

Scientific notations

format(): Scientific notation and rounding to a certain precision

```
x = 0.0079873129957356789
print(format(x,'.2E'))
print(format(x,'.3E'))
```

```
7.99E-03
7.987E-03
```

```
type(format(x,'.2E'))
```

```
str
```

```
y=float(format(x,'.2E'))
print(y)
```

```
0.00798
```

round(): Round to a given number of decimals

```
x = 0.0079873129957356789
round(x, 2)
```

```
0.01
```

Scipy Constants

scipy.constants: see documentation [here]

```
from scipy import constants
```

Examples:

```
# Speed of light in vacuum in meters:
constants.speed_of_light
```

```
299792458.0
```

```
# hbar in SI units:
constants.hbar
```

```
1.0545718176461565e-34
```

Useful tools
seen in the assignments

pandas.DataFrame(): to display nice tables

```
import pandas as pd

d = [ ["Mark", 12, 95],
     ["Jay", 11, 88],
     ["Jack", 14, 90]]

df = pd.DataFrame(d, columns = ['Name','Age','Percent'])
print(df)
```

```
   Name  Age  Percent
0  Mark   12       95
1   Jay   11       88
2  Jack   14       90
```

Part 1

NumPy Arrays

Definition, Methods & Attributes, Indexing & Slicing, Copy

Definition

NumPy Arrays

List: ordered and mutable collection of **objects**.

Objects can be of **any type**, bring flexibility, but **inefficiency**

NumPy array: — **fixed-type**: less flexibility, but more efficiency

— **multi-dimensional** array.

```
import numpy as np
```

Definition

NumPy Arrays

List: ordered and mutable collection of **objects**.

Objects can be of **any type**, bring flexibility, but **inefficiency**

NumPy array: — **fixed-type**: less flexibility, but more efficiency

— **multi-dimensional** array.

One-dimensional arrays:

- create **from a list**:

```
list_of_numbers = [10,11,12,13,14,15,16,17,18]
x = np.array(list_of_numbers)
```

```
array([10, 11, 12, 13, 14, 15, 16, 17, 18])
```

- create with **arange()**

```
x=np.arange(1, 10)
```

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

—> works similarly to **range()**

>>> Q1

Definition

NumPy Arrays

List: ordered and mutable collection of **objects**.

Objects can be of **any type**, bring flexibility, but **inefficiency**

NumPy array: — **fixed-type**: less flexibility, but more efficiency

— **multi-dimensional** array.

Multi-dimensional arrays:

- create **from a list of lists**:

! Same number of elements !

```
list_of_list_of_3numbers = [[1,2,3],[3,4,6]]
x2=np.array(list_of_list_of_3numbers)
```

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

>>> Q2

- create from 1d-array, **using reshape()**

```
y=np.arange(1, 10)
y.reshape((3, 3))
```

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

new dimensions (= “shape")

! Shapes must match !

NumPy Arrays: Attributes

```
x=np.array([[1,2,3],[3,4,6]])
```

```
[[1 2 3]
 [3 4 6]]
```

```
print("ndim: ", x.ndim)
print("shape:", x.shape)
print("size: ", x.size)
print("dtype: ", x.dtype)
```

`ndim:  2`	number of dimensions
`shape: (2, 3)`	size of each dimension
`size:  6`	total size of the array
`dtype:  int64`	data type of the array

NumPy Arrays: Attributes

```
x=np.array([[1,2,3],[3,4,6]])
```

```
[[1 2 3]
 [3 4 6]]
```

```
print("ndim: ", x.ndim)
print("shape:", x.shape)
print("size: ", x.size)
print("dtype: ", x.dtype)
```

`ndim: 2`	number of dimensions
`shape: (2, 3)`	size of each dimension
`size: 6`	total size of the array
`dtype: int64`	data type of the array

Remember: the type is fixed!

If numerical types don't match, Python will upcast, when possible:

```
# Here, integers are up-cast to floating point:
x2=np.array([3.13, 4, 2, 3])
x2.dtype
```

```
dtype('float64')
```

>>> Ex1

NumPy Arrays: Methods

Other useful methods to create arrays from scratch:

>>> Q4

```
np.zeros((3,3), dtype=int)
```

```
np.ones((3, 5), dtype=float)
```

```
np.full((3, 5), 5.128)
```

```
np.linspace(0, 1, 5)
```

```
np.eye(3)
```

NumPy Arrays: Methods

Other useful methods to create arrays from scratch:

>>> Q4

```
np.zeros((3,3), dtype=int)
```

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

```
np.ones((3, 5), dtype=float)
```

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

```
np.full((3, 5), 5.128)
```

```
array([[5.128, 5.128, 5.128, 5.128, 5.128],
       [5.128, 5.128, 5.128, 5.128, 5.128],
       [5.128, 5.128, 5.128, 5.128, 5.128]])
```

```
np.linspace(0, 1, 5)
```

By default: n=50 points

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

```
np.eye(3)
```

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

Indexing

Similar to python's list indexing:

— access value of an element;
— modify value of an element

1D

```
x1=np.arange(10)
```

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

```
x1[3]
```

```
3
```

```
x1[-1]
```

```
9
```

2D

```
x2 = np.array([[1,2,3],[3,4,6]])
```

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

```
print(x2[0, 0])  # 2d-array works like     matrices
```

```
1
```

```
print(x2[1, -1]) # You can also use negative indices here
```

```
6
```

```
x2[0, 0] = 12
x2
```

Modify a value

```
array([[12,  2,  3],
       [ 3,  4,  6]])
```

Indexing

Similar to python's list indexing:

— access value of an element;
— modify value of an element

1D

```
x1=np.arange(10)
```

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

```
x1[3]
```

```
3
```

```
x1[-1]
```

```
9
```

2D

```
x2 = np.array([[1,2,3],[3,4,6]])
```

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

```
print(x2[0, 0])  # 2d-array works like     matrices
```

```
1
```

```
print(x2[1, -1]) # You can also use negative indices here
```

```
6
```

```
x2[0,0]=12.56
x2
```

Modify a value

Indexing

Similar to python's list indexing:

— access value of an element;
— modify value of an element

1D

```
x1=np.arange(10)
```

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

```
x1[3]
```

```
3
```

```
x1[-1]
```

```
9
```

2D

```
x2 = np.array([[1,2,3],[3,4,6]])
```

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

```
print(x2[0, 0])  # 2d-array works like     matrices
```

```
1
```

```
print(x2[1, -1]) # You can also use negative indices here
```

```
6
```

```
x2[0,0]=12.56
x2
```

Type is fixed!!!

```
array([[12,  2,  3],
       [ 3,  4,  6]])
```

Slicing

Similar to python's list slicing: give a name to a sub-array **x [start : stop : step]**

Ex. 1D

```
x1=np.arange(10)
```

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

```
print(x1[1:9])
```

Elements from index 1 to index 9 excluded

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

```
print(x1[::-1])
```

Elements in reverse order

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

Slicing

Similar to python's list slicing: give a name to a sub-array **x [start : stop : step]**

Ex. 2D

```
x2=np.array([[12,  5,  2,  4], [ 7,  6,  8,  8], [ 1,  6,  7,  7]])
```

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

```
x2[:2, :3]
```

2 first row; 3 first columns

```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

```
x2[:,::2]
```

All raws; every other columns

```
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

>>> Q5

Combining Slicing and Indexing

To extract a single row or a single column

Ex. 2D

```
x2=np.array([[12,  5,  2,  4], [ 7,  6,  8,  8], [ 1,  6,  7,  7]])
```

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

```
# first column of x2:
print(x2[:, 0])
```

```
[12  7  1]
```

```
# first row of x2:
print(x2[0, :])
print(x2[0])   # equivalent more compact notation, only for row access
```

```
[12  5  2  4]
[12  5  2  4]
```

Mutable elements

Important: the elements of an array are **mutable**

>>> Q6

```
a0=np.ones((3,3), dtype=float)
a1=a0
a1[1][1]=8
print(a0)
```

Mutable elements

Important: the elements of an array are **mutable**

>>> Q6

```
a0=np.ones((3,3), dtype=float)
a1=a0
a1[1][1]=8
print(a0)
```

```
[[1. 1. 1.]
 [1. 8. 1.]
 [1. 1. 1.]]
```

Mutable elements

Important: the elements of an array are **mutable**

>>> Q6

```
a0=np.ones((3,3), dtype=float)
a1=a0
a1[1][1]=8
print(a0)
```

```
[[1. 1. 1.]
 [1. 8. 1.]
 [1. 1. 1.]]
```

Make a copy:

```
a0=np.ones((3,3), dtype=float)
a1=np.copy(a0)
a1[1][1]=8
print(a0)
```

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

Copy

Slices = views of an Array

Important: Array slices return **views** of the array and not copies!!!!

>>> Ex2

```
x2=np.array([[12,  5,  2,  4], [ 7,  6,  8,  8], [ 1,  6,  7,  7]])
print(x2)
```

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

```
# Let's extract a 2×2 subarray from this:
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[12  5]
 [ 7  6]]
```

```
# Now if we modify this subarray:
x2_sub[0, 0] = 99
print(x2_sub)
```

```
[[99  5]
 [ 7  6]]
```

```
# the original array is changed!
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

If you modify the sub-array, you modify the original array itself!

Copy an Array

Important: Array slices return **views** of the array and not copies!!!!

To create a copy of x: **x.copy()**

>>> Q7

Copy an Array

Important: Array slices return **views** of the array and not copies!!!!

To create a copy of x: x.copy()

>>> Q7

```
# Let's copy a 2×2 subarray from x2:
x2_sub_copy = x2[:2, :2].copy()
print(x2_sub_copy)
```

```
[[99  5]
 [ 7  6]]
```

```
# modification of the sub-array:
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

```
# check that the original array is not touch:
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Part 2
Operations on NumPy Arrays

Why Numpy? , UFunc

Python loops are very slow

In python, variable types are flexible.

Each time python performs an operation on objects, it first has to **check the type of the objects**, and check for the appropriate operations for that type.

Flexible, but very slow…

Huge loss in performance, when the same small operation has to be repeated, ex. looping over arrays to operate on each elements.

```
# Example of a function that is applied to all the elements of a vector:
def reciprocal(x):
    x_reciprocal = np.empty(len(x))
    for i in range(len(x)):
        x_reciprocal[i] = 1.0 / x[i]
    return x_reciprocal

# Test:
np.random.seed(0)
x = np.random.randint(1, 10, size=5)
reciprocal(x)
```

```
array([0.16666667, 1.        , 0.25      , 0.25      , 0.125     ])
```

```
big_array = np.random.randint(1, 100, size=1000000)
%timeit reciprocal(big_array)
```

>>> Ex3

```
2.19 s ± 25.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Very slow!!! —> smartphones 10^9 operations/s

Vectorized operations are much faster!

A vectorized operation: performing an operation directly on the array, which will then be applied to each element.

This vectorized approach is designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution.

```
# The execution time for our big array is
# several orders of magnitude faster than the Python loop:
%timeit (1.0 / big_array)
```

>>> Ex4

```
2.77 ms ± 93 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

—> Much faster!!!

This was with the Python loop: super slow…

```
big_array = np.random.randint(1, 100, size=1000000)
%timeit reciprocal(big_array)
```

```
2.19 s ± 25.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

UFunc

NumPy's universal functions (ufuncs): Vectorized operations are implemented via Ufunc.

Ufuncs are extremely flexible, you can perform operations:
— between a scalar and an array,
— between two arrays of the same shape.

Simple arithmetics:

```
x = np.arange(4)
print(x+5)
print(np.add(x,5))
```

```
[5 6 7 8]
[5 6 7 8]
```

>>> Q8 - Q9

python	equivalent numpy operator	operation
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

UFunc

NumPy's universal functions (ufuncs): Vectorized operations are implemented via Ufunc.

Ufuncs are extremely flexible, you can perform operations:
— between a scalar and an array,
— between two arrays of the same shape.

>>> Q10

Boolean comparison:

```
x=np.array(range(0, 100, 10))
print(x)

# Using boolean operator on arrays:
x>=50
```

```
[ 0 10 20 30 40 50 60 70 80 90]

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

… and masks:

```
#boolean array as mask:
# You can use a list of booleans to define which elements to keep or not
x[[False, False, False, False, False,  True,  True,  True,  True,  True]]
```

```
array([50, 60, 70, 80, 90])
```

UFunc

NumPy's universal functions (ufuncs): Vectorized operations are implemented via Ufunc.

Ufuncs are extremely flexible, you can perform operations:
— between a scalar and an array,
— between two arrays of the same shape.

Other useful Ufunc:

nb.abs( ) , trigonometric functions, exponentials and logarithms, other special functions

Ex. Sampling function: >>> Q11

```
xmin=0
xmax=10
n=30
```

UFunc

NumPy's universal functions (ufuncs): Vectorized operations are implemented via Ufunc.

Ufuncs are extremely flexible, you can perform operations:
— between a scalar and an array,
— between two arrays of the same shape.

Other useful Ufunc:

nb.abs( ) , trigonometric functions, exponentials and logarithms, other special functions

Ex. Sampling function: >>> Q11

```
xmin=0
xmax=10
n=30
```

```
# Example with exponential function
np.exp(np.linspace(xmin, xmax, n))
```

Part 3

Simple uses of NumPy Arrays

Simple data statistics

Advanced UFunc features:

Build-in functions in NumPy that are useful when working with large amount of data.

x.min()
x.max()
x.mean()
x.std()

```
%timeit min(big_array)
%timeit np.min(big_array)
```

```
65.9 ms ± 886 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
339 µs ± 16.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Linear algebra

A two-dimensional array is one representation of a matrix, and NumPy knows how to efficiently do typical matrix operations. For example, you can:

- compute the transpose using `M.T` ;
- compute the dot product (for vector-vector product, matrix-vector product, or matrix-matrix product) using `np.dot(M1,M2)` ;
- compute the trace using `M.trace()` ;
- extract a vector of the diagonal elements, with `M.diagonal()` ;
- flatten the matrix using `M.flatten()` ;
- compute the cross product of two vectors, using `numpy.cross(V1, V2)` .

You can also do more sophisticated operations from the numpy `np.linalg` package, such as:

- get the matrix norm (or vector norm), with `numpy.linalg.norm(M)` ;
- get the rank of the matrix, with `np.linalg.matrix_rank(M)` ;
- get the determinant of a square matrix, with `np.linalg.det(M)` ;
- get the inverse of non-singular square matrix (with determinant different from 0), with `np.linalg.inv(M)` ;
- eigenvalue decomposition using `np.linalg.eigvals(M)` ;
- get a tuple (eigenvalues, eigenvectors) using `np.linalg.eig(M)` ;

>>> Q15

Random Arrays

>>> Q16

Link

Part 4
Introduction to Matplotlib