CS101 Project (2015): Matrices Documentation

Release public

Lilian Besson

June 18, 2016

Contents

This documentation is an example of an *automatically* generated documentation for a Python programming project.

Its main purpose is to be a solution for the CS programming project, given at MEC in April 2015.

The most important page is matrix.html, which documents completely the *matrix* module (main aspect of that project).

Note: This documentation was initially written as a solution for a programming project that I (Lilian Besson) gave to my students in April 2015. It can be interesting to anyone so I published it under an open-source license.

Please feel free to use this documentation and this project, and do not hesitate to contact me if needed.

CHAPTER 1

About the project

- Common instruction paper: CS101_Programming_Projects__What_to_do__and_how_to_do_it.pdf.
- Project paper: CS101_Programming_Project_5__Matrices_operations.pdf.
- The source code can be found here: mec-cs101-matrices/src/ (matrix.py and tests.py).
- More information about the documentation are here: readthedocs.org/projects/mec-cs101-matrices.

Note: How was this documentation created?

In one sentence: I used bitbucket.org to host the Python programs and text files for the documentation, and readthedocs.org takes care of automatically reading the files and (re-)generating this website, whenever something changes in a `.py` or a `.txt` file.

More information about readthedocs can be found here https://docs.readthedocs.io/en/latest/index.html.

CHAPTER 2

Contents:

2.1 Readme

This Python (v2 or v3) project gives an (almost) complete solution for the CS101 programming project, subject #5, about **Matrices and Linear Operations**. This project took place at Mahindra Ecole Centrale in April 2015.

Inside this directory, you will find two Python files (`matrix.py` and `tests.py`).

2.1.1 matrix

Defines the `Matrix` class (*`matrix.Matrix`*), with all its operations and methods. Defines utility functions, like `eye()`, `ones()`, `diag()`, `zeros()` etc.

2.1.2 tests

Performs many tests and examples, by using the matrix module.

2.1.3 Documentation

The documentation is hosted on ReadTheDocs: https://mec-cs101-integrals.readthedocs.io/.

2.1.4 Other files

Please read:

- `INSTALL.txt` : for details about using or installing these files.
- the report, `Matrices_and_Linear_Operations__Project_CS101_2015.pdf`, gives more details about the Python programs, and theoretical explanations about the algorithms we decided to implement, and more small things.
- `AUTHORS.txt` : gives a complete list of authors (only one).
- `TODO.txt` : gives details about un-finished tasks, if you want to conclude the project yourself.
- `LICENSE.txt` : for details about the license under which this project is publicly released.

2.1.5 About this file

It quickly explains what your project was about. It should sum up in a few lines what was the task, and how you solved it.

Imagine that someone downloaded your project and want to understand it, well then this file should be as helpful as possible (while not being too long or verbous). It should be the starting point for a new user.

2.2 How to use this project?

This project does not require any extra modules. It requires Python 2 (v2.7 or more recent) or Python 3 (v3.4 or more recent).

Each of the 2 programs can be executed directly from the command line environnement, either with `python` or with `ipython`, or within `Spyder` (or any IDE). They should work out-of-the-box, without any user interaction.

2.2.1 How to install the project?

If you have installed git on your laptop, you can clone the git repository for this project : `git clone https://bitbucket.org/lbesson/mec-cs101-matrices.git`.

You can also download the entire project as a zip file, then extract the archive, and run the scripts in the extracted directory.

2.2.2 About this file

It quickly explains how to use your project. Any required modules/packages have to be specified here, and if one of your program expect an input from a user, please say it so here.

Imagine that someone downloaded your project and want to use it, well then this file should be as helpful as possible (while not being too long or verbous).

2.3 Documentation for the matrix module

This module *matrix* defines the *matrix.Matrix* class, as asked for the project.

Below is included a documentation (automatically generated from the docstrings present in the source file). Complete solution for the CS101 Programming Project about matrices.

This file defines a class *Matrix*, designed to be as complete as possible. *Do not worry, I was not asking you to do as much.*

2.3.1 Examples

Importing the module:

```
>>> from matrix import *
>>> from matrix import Matrix as M  # shortcut
```

Defining a matrix by giving its list of rows:

```
>>> A = M([[1, 0], [0, 1]])
>>> A == eye(A.n)
True
>>> B = 2*(A**2) + 4*A + eye(A.n)
>>> B
[[7, 0], [0, 7]]
>>> B == 7 * eye(A.n)
True
```

Indexing and slicing:

```
>>> A[1,:] = 2; A
[[1, 0], [2, 2]]
>>> A[0, 0] = -5; A
[[-5, 0], [2, 2]]
```

Addition, multiplication, power etc:

```
>>> C = eye(2); C
[[1, 0], [0, 1]]
>>> C + (3 * C) - C
[[3, 0], [0, 3]]
>>> (4 * C) ** 2
[[16, 0], [0, 16]]
```

Many more examples are given below:

2.3.2 Things that could still be worked on for this solution

Todo

Implement the **QR**, **SVD** and other **matrix decompositions**.

Todo

Try to add a randomized matrix decomposition (or any *less-original* matrix decomposition method)? Note: I worked on this aspect, for a project in January 2016 for my M.Sc. : https://bitbucket.org/lbesson/mva15-project-parcimonie-compressed-sensing/.

Todo

Implement a nice wrapper for a linear equations solver (with LU).

Todo

More doctests for *PLUdecomposition()*, and implement the non-permuted LU decomposition?

Todo

Add more doctests and examples for Gauss, Gauss-Jordan, Gram-Schmidt (*gauss()*, *gauss_jordan()*, *gram_schmidt()*)?

Note: Interactive examples?

See the other file tests.py for *many* examples.

- *Date:* Saturday 18 juin 2016, 10:31:25.
- *Author:* Lilian Besson for the CS101 course at Mahindra Ecole Centrale, 2015,
- *Licence:* MIT Licence.

See also:

I also wrote a complete solution for the other project I was in charge of, about numerical algorithms to compute integrals.

class `matrix.`**`Decimal`**

Extended `decimal.Decimal` class to improve the `str` and `repr` methods.

If there is not digit after the comma, print it as an integer.

`__weakref__`

list of weak references to the object (if defined)

class `matrix.`**`Fraction`**

Extended `fractions.Fraction` class to improve the `str` and `repr` methods.

If the denominator is 1, print it as an integer.

`__weakref__`

list of weak references to the object (if defined)

class `matrix.`**`Matrix`**(*listrows*)

A class to represent matrices of size `(n, m)`.

`M = Matrix(listrows)` will have three attributes:

- `M.listrows` list of rows vectors (as list),
- `M.n` or `M.rows` number of rows,
- `M.` or `M.cols` number of columns (ie. length of the rows).

All the required special methods are implemented, so *Matrix* objects can be used as numbers, with a very natural syntax.

Warning: All the rows should have the same size.

`__init__`(*listrows*)

Create a *Matrix* object from the list of row vectors `M`.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.listrows
[[1, 2, 3], [4, 5, 6]]
```

listrows = None
self.listrows is the list of rows for self

n
Getter for the read-only attribute `A.n` (number of rows).

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.n
2
>>> A.rows == A.n
True
```

rows
Getter for the read-only attribute `A.n` (number of rows).

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.n
2
>>> A.rows == A.n
True
```

m
Getter for the read-only attribute `A.m` (size of the rows, ie. number of columns).

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.m
3
>>> A.cols == A.m
True
```

cols
Getter for the read-only attribute `A.m` (size of the rows, ie. number of columns).

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.m
3
>>> A.cols == A.m
True
```

__getitem__(*ij*)
`A[i, j]` <-> `A.listrows[i][j]` reads the (`i, j`) element of the matrix `A`.

- *Experimental* support of slices: `A[a:b:k, j]`, or `A[i, c:d:l]` or `A[a:b:k, c:d:l]`.
- Default values for `a` and `c` is a **start point** of `0`, `b` and `d` is a **end point** of maximum size, and `k` and `l` is a **step** of `1`.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A[0, 0]
1
>>> A[0, :]
[[1, 2, 3]]
>>> A[-1, :]
[[4, 5, 6]]
>>> A[:, 0]
[[1], [4]]
>>> A[1:, 1:]
```

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

__setitem__(*ij*, *value*)

`A[i, j] = value`: will update the `(i, j)` element of the matrix `A`.

- Support for slice arguments: `A[a:b:k, j] = sub_row`, or `A[i, c:d:l] = sub_column` or `A[a:b:k, c:d:l] = submatrix`.
- Default values for `a` and `c` is a **start point** of `0`, `b` and `d` is a **end point** of maximum size, and `k` and `l` is a **step** of `1`.

```
        >>> A = Matrix([[1, 2, 3], [4, 5, 6]])
        >>> A[0, 0] = 4; A
        [[4, 2, 3], [4, 5, 6]]
        >>> A[:, 0]
        [[4], [4]]
        >>> A[-1, :] = 9; A
        [[4, 2, 3], [9, 9, 9]]
        >>> A[1, 1] = 3; A
        [[4, 2, 3], [9, 3, 9]]
        >>> A[0, :] = [3, 2, 1]; A
        [[3, 2, 1], [9, 3, 9]]
        >>> A[1:, 1:] = -1; A
        [[3, 2, 1], [9, -1, -1]]
        >>> A[1:, 1:] *= -8; A
        [[3, 2, 1], [9, 8, 8]]
```

row(*i*)

`A.row(i)` <-> *extracts* the `i`-th row of `A`, as a *new* matrix.

> **Warning:** Modifying `A.row(i)` does NOT modify the matrix `A`.

```
        >>> A = Matrix([[1, 2, 3], [4, 5, 6]])
        >>> A.row(0)
        [[1, 2, 3]]
        >>> A.row(1)
        [[4, 5, 6]]
        >>> r = A.row(0); r *= 3
        >>> A  # it has not been modified!
        [[1, 2, 3], [4, 5, 6]]
```

col(*j*)

`A.col(j)` <-> *extracts* the `j`-th column of `A`, as a new matrix.

> **Warning:** Modifying `A.col(j)` does NOT modify the matrix A.

```
        >>> A = Matrix([[1, 2, 3], [4, 5, 6]])
        >>> A.col(0)
        [[1], [4]]
        >>> A.col(2)
        [[3], [6]]
        >>> c = A.col(1); c *= 6
        >>> A  # it has not been modified!
        [[1, 2, 3], [4, 5, 6]]
```

copy()

A.copy() <-> a shallow copy of the matrix A (ie. a new and fresh matrix with same values).

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = A.copy()
>>> A[0, 0] = -10; A
[[-10, 2, 3], [4, 5, 6]]
>>> B  # It has not been modified!
[[1, 2, 3], [4, 5, 6]]
```

__len__()

len(A) returns A.n * A.m, the number of values in the matrix.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> len(A)
6
>>> len(A) == A.n * A.m
True
```

shape

A.shape is (A.n, A.m) (similar to the shape attribute of NumPy arrays).

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.shape
(2, 3)
```

transpose()

A.transpose() is the transposition of the matrix A.

- Returns a new matrix!
- Definition: if B = A.transpose(), then B[i, j] is A[j, i].

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.transpose()
[[1, 4], [2, 5], [3, 6]]
>>> A.transpose().transpose() == A
True
```

T

A.T <-> A.transpose() is the transposition of the matrix A, useful shortcut as in NumPy.

```
>>> B = Matrix([[1, 4], [2, 5], [3, 6]])
>>> B.T
[[1, 2, 3], [4, 5, 6]]
>>> B == B.T.T
True
```

__str__()

str(A) <-> A.__str__() converts the matrix A to a string (showing the list of rows vectors).

```
>>> B = Matrix([[1, 4], [2, 5], [3, 6]])
>>> str(B)
'[[1, 4], [2, 5], [3, 6]]'
```

__repr__()

repr(A) <-> A.__repr__() converts the matrix A to a string (showing the list of rows vectors).

```
>>> B = Matrix([[1, 4], [2, 5], [3, 6]])
>>> repr(B)
'[[1, 4], [2, 5], [3, 6]]'
```

__eq__(*B*)

`A == B` <-> `A.__eq__(B)` compares the matrix `A` with `B`.

- Time complexity is $\mathcal{O}(nm)$ for matrices of size `(n, m)`.

```
>>> B = Matrix([[1, 4], [2, 5], [3, 6]])
>>> B == B
True
>>> B + B + B == 3*B == B + 2*B == 2*B + B
True
>>> B - B + B == 1*B == -B + 2*B == 2*B - B == 2*B + (-B)
True
>>> B != B
False
```

almosteq(*B*, *epsilon=1e-10*)

`A.almosteq(B)` compares the matrix `A` with `B`, numerically with an error threshold of `epsilon`.

- Default epsilon is 10^{-10}.
- Time complexity is $\mathcal{O}(nm)$ for matrices of size `(n, m)`.

```
>>> B = Matrix([[1, 4], [2, 5], [3, 6]])
>>> C = B.copy(); C[0,0] += 4*1e-6
>>> B == C
False
>>> B.almosteq(C)
False
>>> B.almosteq(C, epsilon=1e-4)
True
>>> B.almosteq(C, epsilon=1e-5)
True
>>> B.almosteq(C, epsilon=1e-6)
False
```

__lt__(*B*)

`A < B` <-> $A_{i,j} < B_{i,j} \forall i, j$ compares the matrix `A` with `B`.

- Time complexity is $\mathcal{O}(nm)$ for matrices of size `(n, m)`.
- Time complexity is $\mathcal{O}(nm)$ for matrices of size `(n, m)`.
- `A > B`, `A <= B`, `A >= B` are all computed automatically with *__eq__()* and *__lt__()*.

```
>>> B = Matrix([[1, 4], [2, 5], [3, 6]])
>>> B < B
False
>>> B < B + 4
True
>>> B > B
False
>>> B > B - 12
True
```

__add__(*B*)

`A + B` <-> `A.__add__(B)` computes the sum of the matrix `A` and `B`.

- Returns a new matrix!
- Time and memory complexity is $\mathcal{O}(nm)$ for matrices of size `(n, m)`.
- If `B` is a number, the sum is done coefficient wise.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A + A
[[2, 4, 6], [8, 10, 12]]
>>> B = ones(A.n, A.m); B
[[1, 1, 1], [1, 1, 1]]
>>> A + B
[[2, 3, 4], [5, 6, 7]]
>>> B + A
[[2, 3, 4], [5, 6, 7]]
>>> B + B + B + B + B + B + B
[[7, 7, 7], [7, 7, 7]]
>>> B + 4  # Coefficient wise!
[[5, 5, 5], [5, 5, 5]]
>>> B + (-2)  # Coefficient wise!
[[-1, -1, -1], [-1, -1, -1]]
>>> B + (-1.0)  # Coefficient wise!
[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
```

__radd__(*B*)

`B + A` <-> `A.__radd__(B)` computes the sum of `B` and the matrix `A`.

- Returns a new matrix!
- Time and memory complexity is $\mathcal{O}(nm)$ for matrices of size `(n, m)`.
- If `B` is a number, the sum is done coefficient wise.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> 1 + A
[[2, 3, 4], [5, 6, 7]]
>>> B = ones(A.n, A.m)
>>> 4 + B  # Coefficient wise!
[[5, 5, 5], [5, 5, 5]]
>>> (-2) + B  # Coefficient wise!
[[-1, -1, -1], [-1, -1, -1]]
>>> (-1.0) + B  # Coefficient wise!
[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
```

__sub__(*B*)

`A - B` <-> `A.__sub__(B)` computes the difference of the matrix `A` and `B`.

- Returns a new matrix!
- Time and memory complexity is $\mathcal{O}(nm)$ for matrices of size `(n, m)`.
- If `B` is a number, the sum is done coefficient wise.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = ones(A.n, A.m)
>>> A - B
[[0, 1, 2], [3, 4, 5]]
>>> B - A
[[0, -1, -2], [-3, -4, -5]]
>>> A - 1  # Coefficient wise!
[[0, 1, 2], [3, 4, 5]]
>>> B - 2  # Coefficient wise!
[[-1, -1, -1], [-1, -1, -1]]
>>> (A - 3.14).round()  # Coefficient wise!
[[-2.14, -1.14, -0.14], [0.86, 1.86, 2.86]]
```

__neg__()

`-A` <-> `A.__neg__()` computes the opposite of the matrix `A`.

- Returns a new matrix!
- Time and memory complexity is $\mathcal{O}(nm)$ for a matrix of size `(n, m)`.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> -A
[[-1, -2, -3], [-4, -5, -6]]
>>> A - A == A + (-A)
True
>>> -(-A)  == A
True
>>> -------A == -A  # Crazy syntax!
True
>>> s = '-------'
>>> len(s) % 2 == 1  # We check that we had an od number of minus symbol
True
```

__pos__()

`+` <-> `A.__pos__()` computes the positive of the matrix A.

- Returns a new matrix!
- Useless?
- Time and memory complexity is $\mathcal{O}(nm)$ for a matrix of size `(n, m)`.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> +A == A
True
>>> +-+-+-+-+++----+-+-+----++++A == A  # Crazy syntax, again!
True
>>> s = '+-+-+-+-+++----+-+-+----++++'
>>> s.count('-') % 2 == 0  # We check that we had an even number of minus symbol
True
```

__rsub__(*B*)

`B - A` <-> `A.__rsub__(B)` computes the difference of `B` and the matrix `A`.

- Returns a new matrix!
- Time and memory complexity is $\mathcal{O}(nm)$ for matrices of size `(n, m)`.
- If `B` is a number, the sum is done coefficient wise.
- If `B` is a *Matrix* object, `B - A` will in fact be `B.__sub__(A)` and not `A.__rsub__(B)`.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> 1 - A  # Coefficient wise!
[[0, -1, -2], [-3, -4, -5]]
>>> B = ones(A.n, A.m)
>>> (-1) - B  # Coefficient wise!
[[-2, -2, -2], [-2, -2, -2]]
>>> ((-1) - B) == -(1 + B) == -(B + B)
True
```

__mul__(*B*)

`A * B` <-> `A.__mul__(B)` computes the product of the matrix `A` and `B`.

- Returns a new matrix!
- Time and memory complexity is $\mathcal{O}(nmp)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.

•If `B` is a number, the product is done coefficient wise.

Warning: Matrix product is not commutative!

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = eye(A.n); B
[[1, 0], [0, 1]]
>>> A * B == B * A == A
True
>>> A * A
[[7, 10], [15, 22]]
>>> A * (A * A) == (A * A) * A
True
>>> A * 1 == A  # Coefficient wise!
True
>>> A * 12.011993  # Coefficient wise!
[[12.011993, 24.023986], [36.035979, 48.047972]]
```

__rmul__(*B*)

`B * A` <-> `A.__rmul__(B)` computes the product of `B` and the matrix `A`.

•Returns a new matrix!

•Time and memory complexity is $\mathcal{O}(nmp)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.

•If B is a number, the product is done coefficient wise.

•If `B` is a *Matrix* object, `B * A` will in fact be `B.__mul__(A)` and not `A.__rmul__(B)`.

Warning: Matrix product is not commutative!

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> 1 * A == A  # Coefficient wise!
True
>>> 12.011993 * A  # Coefficient wise!
[[12.011993, 24.023986], [36.035979, 48.047972]]
```

multiply_elementwise(*B*)

`A.multiply_elementwise(B)` computes the product of the matrix `A` and `B`, element-wise (it is called a **Hadamard product**).

•Returns a new matrix!

•Time and memory complexity is $\mathcal{O}(nmp)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = eye(A.n)
>>> A.multiply_elementwise(B)
[[1, 0], [0, 4]]
>>> A.multiply_elementwise(A)  # A .^ 2 in Matlab?
[[1, 4], [9, 16]]
```

__div__(*B*)

`A / B` <-> `A * (B ** (-1))` computes the division of the matrix `A` by `B`.

•Returns a new matrix!

•Performs **true division**!

- Time and memory complexity is $\mathcal{O}(nmp \max(m,p)^2)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.
- If `B` is a number, the division is done coefficient wise.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = eye(A.n)
>>> B.almosteq(A / A)
True
>>> C = B.map(float)
>>> A / C == A * C == A
True
>>> A / B == A * B == A
True
>>> A / 2  # Coefficient wise!
[[0.5, 1.0], [1.5, 2.0]]
>>> A / 2.0  # Coefficient wise!
[[0.5, 1.0], [1.5, 2.0]]
```

__truediv__(*B*)

`A / B` <-> `A * (B ** (-1))` computes the division of the matrix `A` by `B`.

- Returns a new matrix!
- Performs **true division**!
- Time and memory complexity is $\mathcal{O}(nmp \max(m,p)^2)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.
- If `B` is a number, the division is done coefficient wise.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = eye(A.n)
>>> B.almosteq(A / A)
True
>>> C = B.map(float)
>>> A / C == A * C == A
True
>>> A / B == A * B == A
True
>>> A / 2  # Coefficient wise!
[[0.5, 1.0], [1.5, 2.0]]
>>> A / 2.0  # Coefficient wise!
[[0.5, 1.0], [1.5, 2.0]]
```

__floordiv__(*B*)

`A // B` <-> `A * (B ** (-1))` computes the division of the matrix `A` by `B`.

- Returns a new matrix!
- Time and memory complexity is $\mathcal{O}(nmp)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.
- If `B` is a number, the division is done coefficient wise with an **integer division** `//`.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = eye(A.n); C = B.map(float)
>>> A // C == A * C == A
True
>>> A // B == A * B == A
True
>>> A // 2  # Coefficient wise!
[[0, 1], [1, 2]]
```

```
>>> A // 2.0  # Coefficient wise!
[[0.0, 1.0], [1.0, 2.0]]
```

__mod__(*b*)

`A % b` <-> `A.__mod__(b)` computes the modulus coefficient-wise of the matrix `A` by `b`.

•Returns a new matrix!

•Time and memory complexity is $\mathcal{O}(nm)$ for a matrix `A` of size `(n, m)`.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> A % 2
[[1, 0], [1, 0]]
>>> (A*100) % 31
[[7, 14], [21, 28]]
>>> (A*100) % 33 == A  # Curious property
True
>>> (A*100) % 35
[[30, 25], [20, 15]]
```

Warning: `A % B` for two matrices means the coefficient-wise modulus.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = Matrix([[2, 3], [2, 2]])
>>> A % B
[[1, 2], [1, 0]]
```

__rdiv__(*B*)

`B / A` <-> `A.__rdiv__(B)` computes the division of `B` by `A`.

Warning: If `B` is `1` (`B == 1`), `1 / A` is `A.inv()` (special case!)

•If `B` is a number, the division is done coefficient wise.

•Returns a new matrix!

•Time and memory complexity is $\mathcal{O}(nmp)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> Ainv = Matrix([[-2.0, 1.0], [1.5, -0.5]])
>>> B = eye(A.n)
>>> B == A * Ainv == Ainv * A
True
>>> 1 / B == B == B / 1
True
>>> C = B.map(float)
>>> 1 / B == B == B / 1
True
>>> A.inv() == 1 / A  # special case!
True
>>> 1 / A  # This is like 1 / A
[[-2.0, 1.0], [1.5, -0.5]]
>>> 2 / (2*A)  # Warning This is coefficient wise !
[[1.0, 0.5], [0.333333..., 0.25]]
```

__rtruediv__(*B*)

`B / A` <-> `A.__rdiv__(B)` computes the division of `B` by `A`.

Warning: If `B` is `1` (`B == 1`), `1 / A` is `A.inv()` (special case!)

•If `B` is a number, the division is done coefficient wise.

•Returns a new matrix!

•Time and memory complexity is $\mathcal{O}(nmp)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> Ainv = Matrix([[-2.0, 1.0], [1.5, -0.5]])
>>> B = eye(A.n)
>>> B == A * Ainv == Ainv * A
True
>>> 1 / B == B == B / 1
True
>>> C = B.map(float)
>>> 1 / B == B == B / 1
True
>>> A.inv() == 1 / A  # special case!
True
>>> 1 / A  # This is like 1 / A
[[-2.0, 1.0], [1.5, -0.5]]
>>> 2 / (2*A)  # Warning This is coefficient wise !
[[1.0, 0.5], [0.333333..., 0.25]]
```

__rfloordiv__(*B*)

`B // A` <-> `A.__rdiv__(B)` computes the division of `B` by `A`.

> **Warning:** If `B` is `1` (`B == 1`), `1 / A` is `A.inv()` (special case!)

•If `B` is a number, the division is done coefficient wise.

•Returns a new matrix!

•Time and memory complexity is $\mathcal{O}(nmp)$ for a matrix `A` of size `(n, m)` and `B` of size `(m, p)`.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> B = eye(A.n)
>>> 1 // B == B == B // 1
True
>>> C = B.map(float)
>>> 1 // B == B == B // 1
True
>>> A.inv() == 1 // A  # special case!
True
>>> 2 // (2*A)  # XXX This is coefficient wise !
[[1, 0], [0, 0]]
```

__pow__(*k*)

`A ** k` <-> `A.__pow__(k)` to compute the product of the square matrix `A` (with the quick exponentation trick).

•Returns a new matrix!

•`k` has to be an integer (`ValueError` will be returned otherwise).

•Time complexity is $\mathcal{O}(n^3 \log(k))$ for a matrix `A` of size (n, n).

•Memory complexity is $\mathcal{O}(n^2)$.

•It uses `A.inv()` (*inv()*) to (try to) compute the inverse if `k < 0`.

•More details are in the solution for the Problem II of the 2nd Mid-Term Exam for CS101.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> A ** 1 == A
True
>>> A ** 2
[[7, 10], [15, 22]]
>>> A * A == A ** 2
True
>>> B = eye(A.n)
>>> B == B ** 1 == A ** 0 == B ** 0
True
>>> divmod(2015, 2)
(1007, 1)
>>> 2015 == 1007*2 + 1
True
>>> A ** 2015 == ((A ** 1007) ** 2 ) * A
True
>>> C = diag([1, 4])
>>> C ** 100
[[1, 0], [0, 160693804425899027554196209234116260252220299378279283530137 6]]
>>> C ** 100 == diag([1**100, 4**100])
True
```

It also accept negative integers:

```
>>> A ** (-1) == A.inv()
True
>>> C = (A ** (-1)); C
[[-2.0, 1.0], [1.5, -0.5]]
>>> C * A == eye(A.n) == A * C
True
>>> C.listrows  # Rounding mistakes can happen (but not here)
[[-2.0, 1.0], [1.5, -0.5]]
>>> D = C.round(); D.listrows
[[-2.0, 1.0], [1.5, -0.5]]
>>> D * A == eye(A.n) == A * D  # No rounding mistake!
True
>>> (C * A).almosteq(eye(A.n))
True
>>> (A ** (-5)) == (A ** 5).inv() == (A.inv()) ** 5
False
>>> (A ** (-5)).round() == ((A ** 5).inv()).round() == ((A.inv()) ** 5).round()  # No roundi
True
```

exp(*limit=30*)

`A.exp()` computes a numerical approximation of the exponential of the square matrix `A`.

- Raise a ValueError exception if `A` is not square.
- Note: $\exp(A) = \mathrm{e}^A$ is defined as the series $\sum_{k=0}^{+\infty} \frac{A^k}{k!}$.
- We only compute the first `limit` terms of this series, hopping that the partial sum will be close to the entire series.
- Default value for `limit` is 30 (it should be enough for any matrix).

```
>>> import math
>>> e = math.e
>>> I = eye(10); I[0, :]
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
>>> I * e == I.exp() == diag([e] * I.n)  # Rounding mistakes!
False
>>> (I * e).round() == I.exp().round() == diag([e] * I.n).round()  # No more rounding mistak
True
>>> C = diag([1, 4])
>>> C.exp() == diag([e ** 1, e ** 4]) == diag([math.exp(1), math.exp(4)])  # Rounding mistak
False
>>> C.exp().almosteq(diag([e ** 1, e ** 4]))  # No more rounding mistakes!
True
>>> diag([e ** 1, e ** 4]).almosteq(diag([math.exp(1), math.exp(4)]))
True
```

inv()

`A.inv()` computes the inverse of the square matrix `A` (if possible), with the Gauss-Jordan algorithm.

- Raise a `ValueError` exception if `A` is not square.
- Raise a `ValueError` exception if `A` is singular.

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> A.inv()
[[-2.0, 1.0], [1.5, -0.5]]
>>> A * A.inv() == A.inv() * A == eye(A.n)  # Rounding mistake can happen (but not here)
True
>>> Ai = A.inv().round()  # No more rounding mistake!
>>> A * Ai == Ai * A == eye(A.n)
True
>>> A.det
-2
>>> O = Matrix([[1, 2], [0, 0]])  # O and not 0
>>> O.is_singular
True
>>> O.inv()  # O is singular!
Traceback (most recent call last):
    ...
ValueError: A.inv() on a singular matrix (ie. non inversible).
>>> O.det
0
```

gauss(*det=False*, *verb=False*, *mode=None*, *maxpivot=False*)

`A.gauss()` implements the Gauss elimination process on matrix `A`.

When possible, the Gauss elimination process produces a row echelon form by applying linear operations to `A`.

- If `maxpivot` is True, we look for the pivot with higher absolute value (can help reducing rounding mistakes).
- If `verb` is True, some details are printed at each steps of the algorithm.
- `mode` can be `None` (default), or `'f'` for fractions (*Fraction*) or `'d'` for decimal numbers (*Decimal*).
- Reference is https://en.wikipedia.org/wiki/Gaussian_elimination#Definitions_and_example_of_algorithm
- We chosed to apply rows operations only: it uses elementary operations on lines/rows: $L'_i \longrightarrow L_i - \gamma \times L_k$ (method *swap_rows()*).
- Can swap two columns in order to select the bigger pivot (increases the numerical stability).
- The function will raise a `ValueError` if the matrix `A` is singular (ie. Gauss process cannot conclude).

- If `det` is `True`, the returned value is `c, d` with `c` the row echelon form, and `d` the determinant. Reference for this part is this wikipedia page.

```
>>> Matrix([[1, 2], [3, 4]]).gauss()
[[1, 2], [0, -2]]
>>> Matrix([[1, 2], [1, 2]]).gauss()
[[1, 2], [0, 0]]
>>> Matrix([[1, 2], [-1, -0.5]]).gauss()
[[1, 2], [0, 1.5]]
>>> Matrix([[1, 2], [3, 4]]).gauss(maxpivot=True)
[[2, 1], [0, 1]]
>>> Matrix([[1, 2], [1, 2]]).gauss(maxpivot=True)
[[2, 1], [0, 0]]
>>> Matrix([[1, 2], [3, 4]]).gauss(det=True)
([[1, 2], [0, -2]], -2)
>>> Matrix([[1, 2], [1, 2]]).gauss(det=True)
([[1, 2], [0, 0]], 0)
```

gauss_jordan(*inv=False*, *verb=False*, *mode=None*, *maxpivot=False*)

`A.gauss_jordan()` implements the Gauss elimination process on matrix `A`.

- If `inv` is `True`, the returned value is `J_n, A**(-1)` with `J_n` the reduced row echelon form of `A`, and `A**(-1)` the computed inverse of A.
- If `maxpivot` is `True`, we look for the pivot with higher absolute value (can help reducing rounding mistakes).

rank

`A.rank` uses the Gauss elimination process to compute the rank of the matrix `A`, by simply counting the number of non-zero elements on the diagonal of the echelon form.

Todo

The Gauss process (*`gauss()`*) has to be changed, and improved for singular matrices (when the rank is not maximum!).

```
>>> Matrix([[1, 2], [3, 4]]).rank
2
>>> Matrix([[1, 2], [1, 2]]).rank
1
>>> zeros(7).rank
0
>>> eye(19).rank
19
```

det

`A.det` uses the Gauss elimination process to compute the determinant of the matrix `A`.

Note: Because it depends of the number of elementary operations performed in the Gauss method, we had to modify the *`gauss()`* method...

```
>>> Matrix([[1, 2], [3, 4]]).det
-2
>>> Matrix([[1, 2], [1, 2]]).det
0
>>> zeros(7).det
0
```

```
>>> eye(19).det
1
```

count (*value*)

`A.count(value)` counts how many times the element `value` is in the matrix `A`.

```
>>> Matrix([[1, 2], [3, 4]]).count(2)
1
>>> Matrix([[1, 2], [1, 2]]).count(2)
2
>>> zeros(7).count(2)
0
>>> zeros(7).count(0)
49
>>> eye(19).count(1)
19
>>> eye(19).count(0)
342
```

__contains__ (*value*)

`value in A <-> A.__contains__(value)` tells if the element `value` is present in the matrix `A`.

```
>>> 4 in Matrix([[1, 2], [3, 4]])
True
>>> 4 in Matrix([[1, 2], [1, 2]])
False
>>> O, I = zeros(7), eye(7)
>>> 3 * I**2 + 2 * I + O ** 0
[[6, 0, 0, 0, 0, 0, 0], [0, 6, 0, 0, 0, 0, 0], [0, 0, 6, 0, 0, 0, 0], [0, 0, 0, 6, 0, 0, 0],
>>> 6 in (3 * I**2 + 2 * I + O ** 0)
True
```

map (*f*, **args*, ***kwargs*)

Apply the function `f` to each of the coefficient of the matrix `A` (returns a new matrix).

```
>>> O, I = zeros(2), eye(2)
>>> I.map(lambda x: x * 4)
[[4, 0], [0, 4]]
>>> O.map(lambda x: x + 6)
[[6, 6], [6, 6]]
>>> A = Matrix([[-1j, -2j], [-2j, -1j]])
>>> A.map(lambda z: abs(z))
[[1.0, 2.0], [2.0, 1.0]]
>>> A.map(lambda z: int(abs(z)))
[[1, 2], [2, 1]]
>>> A.map(lambda z: z + 1j)
[[0j, -1j], [-1j, 0j]]
>>> A.map(lambda z: '"%s"' % str(z))
[["-1j", "-2j"], ["-2j", "-1j"]]
>>> A.map(lambda z: "Look: %s" % str(z))
[[Look: -1j, Look: -2j], [Look: -2j, Look: -1j]]
```

•If `f` needs arguments or key-words arguments, use the `*args` and `**kwargs` :

```
>>> def f(x, n, offset=0):
...     return (x ** n) + offset
>>> A = Matrix([[1, 2], [2, 1]])
>>> A.map(f, 2)
```

```
[[1, 4], [4, 1]]
>>> A.map(f, 2, offset=4)
[[5, 8], [8, 5]]
```

round(*ndigits=8*)

`A.round([ndigits=8])` <-> rounds every coefficient of `A` to `ndigits` digits after the comma.

```
>>> A = (1. / 3.) * eye(2) + 4
>>> A.round(0)
[[4.0, 4.0], [4.0, 4.0]]
>>> A.round(2)
[[4.33, 4.0], [4.0, 4.33]]
>>> A.round(7)
[[4.3333333, 4.0], [4.0, 4.3333333]]
```

__iter__()

`iter(A)` <-> `A.__iter__()` is used to create an iterator from the matrix `A`.

- The values are looped rows by rows, then columns then columns.
- This method is called when an iterator is required for a container. This method should return a new iterator object that can iterate over all the objects in the container.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> list(A)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

__next__()

For Python 3 compatibility.

next()

Generator for iterating the matrix `A`.

- The values are looped rows by rows, then columns then columns.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> for x in A:
...     print(x)
1
2
3
4
5
6
7
8
9
>>> for i, x in enumerate(A):
...     print(i, "th value of A is", x)
0 th value of A is 1
1 th value of A is 2
2 th value of A is 3
3 th value of A is 4
4 th value of A is 5
5 th value of A is 6
6 th value of A is 7
7 th value of A is 8
8 th value of A is 9
```

real

Real part of the matrix `A`, coefficient wise.

```
>>> A = Matrix([[1j, 2j], [2j, 1j]])
>>> A.real
[[0.0, 0.0], [0.0, 0.0]]
>>> A = Matrix([[1+6j, 2], [-1+2j, 1+9j]])
>>> A.real
[[1.0, 2], [-1.0, 1.0]]
```

imag
Imaginary part of the matrix `A`, coefficient wise.

```
>>> A = Matrix([[-1j, -2j], [-2j, -1j]])
>>> A.imag
[[-1.0, -2.0], [-2.0, -1.0]]
```

conjugate()
Conjugate part of the matrix `A`, coefficient wise.

```
>>> A = Matrix([[-1j, -2j], [-2j, -1j]])
>>> A.conjugate()
[[1j, 2j], [2j, 1j]]
```

dot(*v*)
`A.dot(v)` computes the dot multiplication of the matrix `A` and the vector `v` ($A\dot{v}$).

- `v` can be a matrix (*Matrix*) of size `(m, 1)`, or a list of size `m`.

```
>>> A = Matrix([[1, 1], [1, -1]])
>>> v = [2, 3]
>>> A.dot(v)
[[5], [-1]]
>>> v = Matrix([[2], [-3]])
>>> A.dot(v)
[[-1], [5]]
```

Warning: An exception `ValueError` is raised if the sizes does not allow the dot product:

```
>>> A.dot(v.T)  # v.T is not a column vector!
Traceback (most recent call last):
    ...
ValueError: A.dot(v): the vector v = [[2, -3]] is not a vector: v.m = 2 != 1.
>>> v = Matrix([[2], [-3], [7]])
>>> A.dot(v)
Traceback (most recent call last):
    ...
ValueError: A.dot(v): the size of the vector v = [[2], [-3], [7]] should be compatible with
>>> v = [1, 2, 3, 4, 5]
>>> A.dot(v)
Traceback (most recent call last):
    ...
ValueError: A.dot(v): the size of the vector v = [[1], [2], [3], [4], [5]] should be compati
```

norm(*p=2*)
`A.norm(p)` computes the p-norm of the matrix `A`, default is `p = 2`.

- Mathematically defined as p-root of the sum of the p-power of *modulus* of its coefficients :

$$\|A\|_p := \left(\sum_{1 \leq i \leq n, 1 \leq j \leq m} |A_{i,j}|^p \right)^{\frac{1}{p}}$$

- If `p = 'inf'`, the max norm is returned (ie. infinity norm), defined by $\|A\|_\infty := \max_{i,j} |A_{i,j}|$.
- Reference is Matrix norm (on Wikipedia).

```
>>> A = Matrix([[1, 2], [-3, -1]])
>>> A.norm()  # (1)**2 + (2)**2 + (-3)**2 + (-1)**2
3.872983346207417
>>> 15**0.5
3.872983346207417
>>> A.norm('inf')
3
>>> A.norm(1) == 7  # (1) + (2) + (3) + (1)
True
>>> A.norm(3)
3.332221851645953
```

normalized(*fnorm=None*, **args*, ***kwargs*)

`A.normalized()` return a new matrix, which **columns vectors are normalized** by using the norm `2` (or the given function `fnorm`).

- Will **not fail** if a vector has norm `0` (it is just not modified).
- Reference is Orthogonalization (on Wikipedia).
- Any extra arguments `args`, `kwargs` are given to the function `fnorm`.

```
>>> A = Matrix([[1, 2], [-3, -1]])
>>> A.normalized(p='inf')
[[0.333333..., 1.0], [-1.0, -0.5]]
>>> eye(5).normalized(p='inf').map(int)  # normalize then round to an int
[[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]]
>>> B = -eye(5)
>>> (2*B).normalized()  # each vector is divided by its norm = 2
[[-1.0, 0.0, 0.0, 0.0, 0.0], [0.0, -1.0, 0.0, 0.0, 0.0], [0.0, 0.0, -1.0, 0.0, 0.0], [0.0, 0
>>> B.normalized(p='inf')
[[-1.0, 0.0, 0.0, 0.0, 0.0], [0.0, -1.0, 0.0, 0.0, 0.0], [0.0, 0.0, -1.0, 0.0, 0.0], [0.0, 0
```

It works also for a simple vector:

```
>>> v = Matrix([[1], [-2], [3]])
>>> v.normalized()
[[0.267261...], [-0.534522...], [0.801783...]]
>>> v.normalized(p=2)
[[0.267261...], [-0.534522...], [0.801783...]]
>>> v.normalized() * (14**0.5)
[[1.0], [-2.0], [3.0]]
>>> v.normalized(p=1)
[[0.166666...], [-0.333333...], [0.5]]
>>> v.normalized(p=1) * 6
[[1.0], [-2.0], [3.0]]
>>> 6 * v.normalized(p=1)
[[1.0], [-2.0], [3.0]]
```

__abs__()

`abs(A)` <-> `A.abs()` <-> `A.__abs__()` computes the absolute value / modulus of `A` coefficient-wise.

```
>>> A = Matrix([[-4, 2+2j], [0, 4j]])
>>> abs(A)
[[4, 2.828427...], [0, 4.0]]
>>> B = -eye(2)
>>> B.abs()
[[1, 0], [0, 1]]
```

abs()

abs(A) <-> A.abs() <-> A.__abs__() computes the absolute value / modulus of A coefficient-wise.


```
>>> A = Matrix([[-4, 2+2j], [0, 4j]])
>>> abs(A)
[[4, 2.828427...], [0, 4.0]]
>>> B = -eye(2)
>>> B.abs()
[[1, 0], [0, 1]]
```


trace()

A.trace() computes the trace of A :

$$\mathrm{Tr}(A) := \sum_{1 \leq i \leq \min(n,m)} A_{i,i}$$

```
>>> A = Matrix([[-4, 2+2j], [0, 4j]])
>>> A.trace()
(-4+4j)
>>> eye(19).trace()
19
>>> zeros(20).trace()
0
>>> ones(100).trace()
100
```

is_square

A.is_square tests if A is **square** or not.

```
>>> A = Matrix([[-4, 2+2j], [0, 4j]])
>>> A.is_square
True
>>> v = Matrix([[-4], [0]])
>>> v.is_square
False
```

is_symetric

A.is_symetric tests if A is **symetric** or not.

```
>>> A = Matrix([[-4, 2+2j], [0, 4j]])
>>> A.is_symetric
False
>>> eye(30).is_symetric
True
```

is_anti_symetric

A.is_anti_symetric tests if A is **anti-symetric** or not.

```
>>> A = Matrix([[0, 1], [-1, 0]])
>>> A.is_anti_symetric
True
```

```
>>> eye(30).is_anti_symetric
False
```

is_diagonal

`A.is_diagonal` tests if A is **diagonal** or not.

```
>>> eye(40).is_diagonal
True
>>> A = Matrix([[0, 1], [-1, 0]])
>>> A.is_diagonal
False
>>> A = diag(range(30))
>>> A.is_diagonal
True
```

is_hermitian

`A.is_hermitian` tests if `A` is **Hermitian** or not (tests if $A^* = A$, ie. `conjugate(A.T) == A`).

```
>>> A = Matrix([[1, 2j], [-2j, 1]])
>>> A.is_hermitian
True
>>> eye(30).is_hermitian
True
>>> (1j * ones(3)).is_hermitian
False
```

is_lower

`A.is_lower` tests if `A` is **lower triangular** or not.

```
>>> A = Matrix([[8, 1], [0, 7]])
>>> A.is_lower
False
>>> A.T.is_lower
True
```

is_upper

`A.is_upper` tests if `A` is **upper triangular** or not.

```
>>> A = Matrix([[2, 0], [3, 4]])
>>> A.is_upper
False
>>> A.T.is_upper
True
```

is_zero

`A.is_zero` tests if `A` is the **zero matrix** or not.

```
>>> A = Matrix([[2, 0], [3, 4]])
>>> A.is_zero
False
>>> zeros(30).is_zero
True
>>> (0 * A).is_zero
True
```

is_singular

`A.is_singular` tests if `A` is **singular** (ie. non-invertible) or not.

Note: It computes the determinant by using the Gauss elimination process (*det()*).

```
>>> A = Matrix([[2, 0], [3, 4]])
>>> A.is_singular
False
>>> zeros(3).is_singular
True
>>> (0 * A).is_singular
True
>>> Matrix([[2, 0], [4, 0]]).is_singular
True
```

swap_cols(*j1*, *j2*)

`A.swap_cols(j1, j2)` changes *in place* the `j1`-th and `j2`-th *columns* of the matrix `A`.

```
>>> A = Matrix([[2, 0], [3, 4]]); A
[[2, 0], [3, 4]]
>>> A.swap_cols(0, 1); A
[[0, 2], [4, 3]]
```

swap_rows(*i1*, *i2*)

`A.swap_rows(i1, i2)` changes *in place* the `i1`-th and `i2`-th *rows* of the matrix `A`.

```
>>> A = Matrix([[2, 0], [3, 4]]); A
[[2, 0], [3, 4]]
>>> A.swap_rows(0, 1); A
[[3, 4], [2, 0]]
```

minor(*i*, *j*)

`A.minor(i, j)` <-> `minor(A, i, j)` returns the `(i, j)` minor of `A`, defined as the determinant of the submatrix `A[i0, j0]` for `i0 != i` and `j0 != j`.

•Complexities: memory is $\mathcal{O}(n^2)$, time is $\mathcal{O}(n^3)$ (1 determinant of size `n - 1`).

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> A.minor(0, 0)
4
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> A.minor(0, 0)  # | 5 6  8 9 | = 5 * 9 - 6 * 8 = -3
-3.000000000000007
>>> A.minor(1, 0)  # | 2 3  8 9 | = 2 * 9 - 3 * 8 = -6
-6
```

cofactor(*i*, *j*)

`A.cofactor(i, j)` <-> `cofactor(A, i, j)` returns the `(i, j)` cofactor of `A`, defined as the `(-1)**(i + j)` times to `(i, j)` minor of `A` (cf. *`minor()`*).

•Complexities: memory is $\mathcal{O}(n^2)$, time is $\mathcal{O}(n^3)$ (1 determinant of size `n - 1`).

```
>>> A = Matrix([[1, 2], [3, 4]])
>>> A.cofactor(0, 0)
4
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> A.cofactor(0, 0)  # (-1)**0 * | 5 6  8 9 | = 5 * 9 - 6 * 8 = -3
-3.000000000000007
>>> A.cofactor(1, 0)  # (-1)**1 * | 2 3  8 9 | = -(2 * 9 - 3 * 8) = 6
6
```

adjugate()

`A.adjugate()` <-> `adjugate(A)` returns the **adjugate matrix** of `A`.

•Reference is https://en.wikipedia.org/wiki/Adjugate_matrix#Inverses.

•Complexities: memory is $\mathcal{O}(n^2)$, time is $\mathcal{O}(n^5)$ (n^2 determinants of size `n - 1`).

•Using the adjugate matrix for computing the inverse is a BAD method : too time-consuming ! LU or Gauss-elimination is only $\mathcal{O}(n^3)$.

```
>>> A = Matrix([[2, 0], [3, 4]])
>>> A.adjugate()
[[4, -3], [0, 2]]
>>> A * A.adjugate() == A.det * eye(A.n)
False
>>> A * A.adjugate().T == A.det * eye(A.n)
True
```

type()
`A.type()` returns the matrix of types of coefficients of `A`.

__weakref__
list of weak references to the object (if defined)

matrix.**ones**(*n*, *m=None*)
`ones(n, m)` is a matrix of size `(n, m)` filled with `1`.

```
>>> ones(3, 2)
[[1, 1], [1, 1], [1, 1]]
>>> ones(2, 3)
[[1, 1, 1], [1, 1, 1]]
```

•It works with only one dimension, or with a tuple `(n, m)` :

```
>>> ones(2)
[[1, 1], [1, 1]]
>>> ones((2, 3))
[[1, 1, 1], [1, 1, 1]]
```

matrix.**zeros**(*n*, *m=None*)
`zeros(n, m)` is a matrix of size `(n, m)` filled with `0`.

```
>>> zeros(3, 2)
[[0, 0], [0, 0], [0, 0]]
>>> zeros(2, 3)
[[0, 0, 0], [0, 0, 0]]
>>> ones(2, 3) == zeros(2, 3) + 1
True
>>> zeros(2, 3) == ones(2, 3) * 0
True
```

•It works with only one dimension, or with a tuple `(n, m)` :

```
>>> zeros(2)
[[0, 0], [0, 0]]
>>> zeros((2, 3))
[[0, 0, 0], [0, 0, 0]]
```

matrix.**eye**(*n*)
`eye(n)` is the (square) identity matrix of size `(n, n)` (`1` on the diagonal, `0` outside).

```
>>> eye(2)
[[1, 0], [0, 1]]
>>> zeros(18) == eye(18) * 0
```

```
    True
    >>> eye(60).is_diagonal
    True
    >>> eye(40).is_square
    True
    >>> eye(20).is_singular
    False
    >>> eye(5).det
    1
    >>> eye(7).trace()
    7
```

matrix.**diag**(*d*, *n=None*)

diag(d) creates a matrix from a list d (or iterator) of diagonal values, or with n-times the value d if d is not an iterator and n is an integer.

```
    >>> D = diag(range(1,6))
    >>> D[2, :]
    [[0, 0, 3, 0, 0]]
```

We can check the usual properties of diagonal matrices:

```
    >>> D.trace()
    15
    >>> D.trace() == sum(range(1,6))
    True
    >>> D.det
    120
    >>> from math import factorial
    >>> D.det == factorial(5)
    True
```

Other examples:

```
    >>> diag([-1, 1])
    [[-1, 0], [0, 1]]
    >>> diag([-4, 1]) + 3
    [[-1, 3], [3, 4]]
```

We can also use the optional argument n:

```
    >>> diag(3.14, 3)
    [[3.14, 0, 0], [0, 3.14, 0], [0, 0, 3.14]]
    >>> diag([3.14]*3)  # Same !
    [[3.14, 0, 0], [0, 3.14, 0], [0, 0, 3.14]]
```

matrix.**mat_from_f**(*f*, *n*, *m=None*, **args*, ***kwargs*)

mat_from_f(f, n, m=None) creates a matrix of size (n, m) initialized with the function f : A[i, j] = f(i, j).

- Default value for m is n (square matrix).

> **Warning:** f has to accept (at least) two arguments i, j.

```
    >>> mat_from_f(lambda i, j: 1 if i == j else 0, 3) == eye(3)
    True
    >>> mat_from_f(lambda i, j: 1, 3) == ones(3)
    True
    >>> mat_from_f(lambda i, j: i+j, 3)
```

```
    [[0, 1, 2], [1, 2, 3], [2, 3, 4]]
    >>> mat_from_f(lambda i, j: i*j, 3)
    [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

•Any extra arguments `args`, `kwargs` are given to the function `f`.

```
    >>> def f(i, j, e, offset=0):
    ...     return (i * e) + offset
    >>> mat_from_f(f, 2, 2, 4)  # n = 2, m = 2, e = 4
    [[0, 0], [4, 4]]
    >>> mat_from_f(f, 2, 2, 4, offset=10)  # n = 2, m = 2, e = 4, offset = 10
    [[10, 10], [14, 14]]
```

•Remark: it is similar to `Array.make` (or `Array.init`) in OCaml (v3.12+) or `String.create` (or `String.make`).

`matrix.`**`det`**`(`*A*`)`
`det(A)` <-> `A.det` computes the determinant of `A` (in $\mathcal{O}(n^3)$).

```
    >>> det(eye(2))
    1
    >>> det((-1) * eye(4))
    1
    >>> det((-1) * eye(5))
    -1
```

`matrix.`**`rank`**`(`*A*`)`
`rank(A)` <-> `A.rank` computes the rank of `A` (in $\mathcal{O}(n^3)$).

```
    >>> rank(eye(2))
    2
```

`matrix.`**`gauss`**`(`*A*, **args*, ***kwargs*`)`
`gauss(A)` <-> `A.gauss()` applies the Gauss elimination process on `A` (in $\mathcal{O}(n^3)$).

`matrix.`**`gauss_jordan`**`(`*A*, **args*, ***kwargs*`)`
`gauss_jordan(A)` <-> `A.gauss_jordan()` applies the Gauss-Jordan elimination process on `A` (in $\mathcal{O}(n^3)$).

`matrix.`**`inv`**`(`*A*`)`
`inv(A)` <-> `A.inv()` **tries** to compute the inverse of `A` (in $\mathcal{O}(n^3)$).

```
    >>> inv(eye(2)) == eye(2)
    True
```

`matrix.`**`exp`**`(`*A*, **args*, ***kwargs*`)`
`exp(A)` <-> `A.exp()` computes an approximation of the exponential of `A` (in $\mathcal{O}(n^3 * limit)$).

```
    >>> import math
    >>> e = math.exp(1.0)
    >>> C = diag([1, 4])
    >>> exp(C) == diag([e ** 1, e ** 4]) == diag([math.exp(1), math.exp(4)])  # Rounding mistakes!
    False
    >>> exp(C).almosteq(diag([e ** 1, e ** 4]))  # No more rounding mistakes!
    True
    >>> diag([e ** 1, e ** 4]).almosteq(diag([math.exp(1), math.exp(4)]))
    True
```

matrix.**PLUdecomposition**(*A*, *mode=None*)

PLUdecomposition(A) computes the **permuted LU decomposition** for the matrix A.

- Operates in time complexity of $\mathcal{O}(n^3)$, memory of $\mathcal{O}(n^2)$.
- mode can be None (default), or 'f' for fractions (Fractions) or 'd' for decimal (*Decimal*) numbers.
- Returned P, L, U that satisfies P*A = L*U, with P being a permutation matrix, L a lower triangular matrix, U an upper triangular matrix.
- Will raise a ValueError exception if A is singular.
- Reference is Gauss elimination (on Wikipedia).
- We chosed to apply rows operations only: it uses elementary operations on lines/rows: $L'_i \longrightarrow L_i - \gamma \times L_k$ (method swap_rows).
- Can swap two columns in order to select the bigger pivot (increases the numerical stability).

matrix.**norm**(*A*, *p=2*, **args*, ***kwargs*)

norm(A, p) <-> A.norm(p) computes the p-norm of A (default is p = 2).

matrix.**trace**(*A*, **args*, ***kwargs*)

trace(A) <-> A.trace() computes the trace of A.

matrix.**rand_matrix**(*n=1*, *m=1*, *k=10*)

rand_matrix(n, m, k) generates a new random matrix of size (n, m) with each coefficients being integers, randomly taken between -k and k (bound *included*).

```
>>> from random import seed
>>> seed(0)  # We want the examples to always be the same
>>> rand_matrix(2, 3)
[[7, 5, -2], [-5, 0, -2]]
>>> rand_matrix(3, 2, 40)
[[23, -16], [-2, 7], [33, 0]]
>>> rand_matrix(4, 4, 100)
[[-44, 51, 24, -50], [82, 97, 62, 81], [-38, 46, 80, 37], [-6, -80, -13, 22]]
```

matrix.**rand_matrix_float**(*n=1*, *m=1*, *k=10*)

rand_matrix_float(n, m, k) generates a new random matrix of size (n, m) with each coefficients being float numbers, randomly taken between -k and k (right bound excluded).

```
>>> from random import seed
>>> seed(0)  # We want the examples to always be the same
>>> rand_matrix_float(2, 3)
[[6.8884370305, 5.15908805881, -1.58856838338], [-4.82166499414, 0.225494427372, -1.90131725099]
>>> rand_matrix_float(3, 2, 1)
[[0.56759717807, -0.393374547842], [-0.0468060916953, 0.16676407891], [0.816225770391, 0.0093737
>>> rand_matrix_float(4, 4, 20)
[[-8.72648622401, 10.2321681663, 4.73475986701, -9.9797463455], [16.3898502387, 19.3114190415, 1
```

matrix.**_argmax**(*indexes*, *array*)

Compute the index i in indexes such that the array[i] is the bigger.

matrix.**_prod**(*iterator*)

Compute the product of the values in the iterator iterator. Empty product is 1.

matrix.**_ifnone**(*a*, *b*)

b if (a is None), else a.

- Useful for converting a slice object to a range object (slice, range).

matrix.**_slice_to_range**(*sliceobject*)
Get a range of indeces from a slice object (slice, range).

•Thanks to this answer on stackoverflow.com.

matrix.**innerproduct**(*vx*, *vy*)
(Hermitian) dot product of the two vectors vx and vy (sum of conjugate(vx[i]) * vy[i]):

$$\mathbf{x}.\mathbf{y} = \langle \mathbf{x}, \mathbf{y} \rangle := \sum_{1 \leq i \leq n} \overline{x_i} \times y_i.$$

```
>>> vx = [1, 2, 3]; vy = [-1, 0, 4]
>>> innerproduct(vx, vy)
11
```

Warning: The conjugate is on the first vector, as always for Hermite spaces and Hermitian inner product.

```
>>> vx = [1j, 2j, 3j]; vy = [-1, 0, 4]
>>> (-1j) * (-1) + (-2j) * (0) + (-3j) * (4)
-11j
>>> innerproduct(vx, vy)
-11j
```

matrix.**norm_square**(*u*)
Shortcut for the square of the norm of the vector u:

$$\|u\|^2 := \langle u, u \rangle.$$

```
>>> u = [1, 2, 3]
>>> norm_square(u)
14
```

•It works for imaginary valued vectors:

```
>>> u = [1j, -2j, 3j]
>>> norm_square(u)
14.0
```

•And it also works for complex valued vectors:

```
>>> u = [1+1j, 2-2j, 3+3j]
>>> norm_square(u)
28.0
```

matrix.**norm2**(*u*)
Shortcut for the canonical norm of the vector u:

```
>>> u = [1, 2, 3]
>>> norm2(u)
3.7416573867739413
```

•It works for imaginary valued vectors:

```
>>> u = [1j, -2j, 3j]
>>> norm2(u)
3.7416573867739413
```

•And it also works for complex valued vectors:

```
>>> u = [1+1j, 2-2j, 3+3j]
>>> norm2(u)
5.291502622129181
```

matrix.**vect_const_multi**(*vx*, *c*)

Multiply the vector `vx` by the constant `c` (scalar, ie. real or complex).

```
>>> vx = [1, 2, 3]; vy = [-1, 0, 4]
>>> vect_const_multi(vx, 2)
[2, 4, 6]
>>> vect_const_multi(vy, -4)
[4, 0, -16]
```

matrix.**proj**(*u*, *v*)

Projection of the vector `v` into the vector `u` ($\text{proj}_u(v)$ as written on Wikipedia).

```
>>> u = [1, 2, 3]; v = [-1, 0, 4]
>>> proj(u, v)  # 11/14 * u
[0.7857142857142857, 1.5714285714285714, 2.357142857142857]
>>> proj(u, v) == [(11/14) * x for x in u]
True
```

matrix.**gram_schmidt**(*V*, *normalized=False*)

Basic implementation of the Gram-Schmidt process for the column vectors of the matrix `V`, in the easy case of $\mathbb{R}^n$ with the usual dot product.

•The matrix is interpreted as a family of *column* vectors.

•Reference for notations, concept and proof is Gram-Schmidt process (on Wikipedia).

•If `normalized` is `True`, the vectors are normalized before being returned.

```
>>> V = Matrix([[1, 2, 3], [-1, 0, 4]])
>>> gram_schmidt(V)
[[1, 2, 3], [-1, 0, 4]]
```

matrix.**minor**(*A*, *i*, *j*)

`minor(A, i, j)` <-> `A.minor(i, j)` returns the `(i, j)` minor of `A`, defined as the determinant of the submatrix `A[i0,j0]` for `i0 != i` and `j0 != j`.

•Complexities: memory is $\mathcal{O}(n^2)$, time is $\mathcal{O}(n^3)$ (1 determinant of size `n - 1`).

matrix.**cofactor**(*A*, *i*, *j*)

`cofactor(A, i, j)` <-> `A.cofactor(i, j)` returns the `(i, j)` cofactor of `A`, defined as the `(-1)**(i + j)` times to `(i, j)` minor of `A` (cf. *`minor()`*).

•Complexities: memory is $\mathcal{O}(n^2)$, time is $\mathcal{O}(n^3)$ (1 determinant of size `n - 1`).

matrix.**adjugate**(*A*)

`adjugate(A)` <-> `A.adjugate()` returns the adjugate matrix of `A`.

•Reference is Adjugate matrix (on Wikipedia).

•Complexities: memory is $\mathcal{O}(n^2)$, time is $\mathcal{O}(n^5)$ (n^2 determinants of size `n - 1`).

- Using the adjugate matrix for computing the inverse is a BAD method : too time-consuming ! LU or Gauss-elimination is only $\mathcal{O}(n^3)$.

2.4 Documentation for the tests script

This script *tests* tests all the integration functions required for the project (written in matrix.html). Below is included an auto-generated documentation (from the docstrings present in the source file). Complete solution for the CS101 Programming Project about matrices.

This file uses the module `matrix`, and its class `matrix.Matrix`, to do many examples of matrices and linear operations.

Examples and naming conventions for all these functions and methods are mainly inspired of http://docs.sympy.org/dev/modules/matrices/matrices.html.

- *Date:* Saturday 18 juin 2016, 10:31:25.
- *Author:* Lilian Besson for the CS101 course at Mahindra Ecole Centrale, 2015,
- *Licence:* MIT Licence.

2.5 The MIT License (MIT)

2.6 Things to do for this project

This project is done, there is not much to do.

2.6.1 About this file

In case that some part of your project is not done (not completed yet), you can explain here what still has to be done.

Imagine that some other team would have to work on your project, and conclude it, well then this file should be as helpful for them as possible (while not being too long or verbous).

2.7 Authors

Lilian Besson, 14XJ00999, lilian.besson at crans dot org

2.7.1 About this file

It has to contain a list, line by line, of each member of your team, following this format: Name, Roll#, email ID. Adding your name and personal information in this file is like signing *numerically*: it proved that you participated.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Copyright

m

t

Symbols

A

C

D

E

F

G

I

L

M

N

O

P

R

S

T

V

Z