CS101 Project (2015) : Matrices Documentation

Release beta

Lilian Besson

December 14, 2015

Contents

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

Todo

Conclude this index page!

Todo

Explain a little bit how it was created.

CHAPTER 1

Contents:

1.1 Readme

This Python 2 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).

1.1.1 matrix

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

1.1.2 tests

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

—

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,
- 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,

—

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.

1.2 How to use this project

This project does not require any extra modules. It needs Python 2 (v2.7 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.

—

1.2.1 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).

1.3 License

1.4 Things to do for this project

This project is entirely concluded, there is nothing else to do.

—

1.4.1 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).

1.5 Authors

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

—

1.5.1 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.

1.6 Documentation for the matrix module

This module *matrix* defines the *matrix.Matrix* class, as asked for the project. 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 defines a class Matrix, to be extended as much as possible.

1.6.1 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]]
```

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

- @date: Tue Apr 07 14:09:03 2015.
- @author: Lilian Besson for CS101 course at Mahindra Ecole Centrale 2015.
- @licence: MIT Licence (http://lbesson.mit-license.org).

class `matrix.Decimal`

Extended fractions.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.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.

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

- *Experimental* 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 max size, and k and l is a step of 1.
- TODO: clean up the code, improve it.
- TODO: Write more doctests!

```
>>> 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*) → 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*) → 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 shallow copy of the matrix A.

```
>>> 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.

```
>>> 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.

```
>>> 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 O(n * m) 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 O(n * m) 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
```

__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 O(n * m) 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 O(n * m) 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 O(n * m) 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  # 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 O(n * m) 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
```

__pos__()

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

•Returns a new matrix!

•Useless?

•Time and memory complexity is O(n * m) 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 O(n * m) 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  # 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 O(n * m * p) 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.

```
>>> 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 O(n * m * p) 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.

```
>>> 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 (Hadamard product).

•Returns a new matrix!

•Time and memory complexity is O(n * m * p) 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 O(n * m * p * 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.

These examples are failing in a simple Python console, but work fine in an IPython console... **But I do not know why!**

__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 O(n * m * p) 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 O(n * m) 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]]
```

__rdiv__(*B*)

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

•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 O(n * m * p) for a matrix A of size (n, m) and B of size (m, p).

These examples are failing in a simple Python console, but work fine in an IPython console... **But I do not know why!**

__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 O(n^3 log(k)) for a matrix A of size (n, n).

•Memory complexity is O(n^2).

•Use A.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=100*)

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 100 (it should be enough for any matrix).

```
>>> from math import e
>>> import math
>>> 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.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 or 'd' for decimal numbers.
- 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' <– L_i - gamma * 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 https://en.wikipedia.org/wiki/Gaussian_elimination#Computing_determinants.

```
>>> 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.

- FIXME: the Gauss process has to be changed, and improved for singular matrices (when the rank is not maximum!).

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 Matrix.gauss method...

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

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

map (*f*, **args*, ***kwargs*)
Apply the function f to each of the coefficient of the matrix A (returns a new matrix).

round ([*ndigits=8*]) → rounds every coefficient to ndigits digits after the comma.

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

- 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.

__next__ ()
Python 3 compatibility (FIXME ?).

next()
Generator for iterating the matrix A.

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

real
Real part of the matrix, coefficient wise.

imag
Imaginary part of the matrix, coefficient wise.

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

dot(*v*)
A.dot(v) <-> A . v computes the dot multiplication of the vector v.

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

norm(*p=2*)
A.norm() 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.
- Reference is https://en.wikipedia.org/wiki/Matrix_norm#.22Entrywise.22_norms.

normalized(*fnorm=None*)
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 simply not modified).
- Reference is https://en.wikipedia.org/wiki/Orthogonalization.

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

trace()
A.trace() computes the trace of A.

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

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

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

is_diagonal
A.is_symetric tests if A is anti-symetric or not.

is_hermitian
A.is_hermitian tests if A is symetric or not.

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

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

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

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

- Computes the determinant by using the Gauss elimination process.

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

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

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 O(n^2), time is O(n^3) (1 determinant of size n-1).

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.

- Complexities: memory is O(n^2), time is O(n^3) (1 determinant of size n-1).

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

- Reference is https://en.wikipedia.org/wiki/Adjugate_matrix#Inverses.
- Complexities: memory is O(n^2), time is 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 O(n^3).

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.

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

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

matrix.**diag**(*d*)
diag(d) creates a matrix from a list of diagonal values.

matrix.**mat_from_f**(*f*, *n*, *m=None*)
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 two arguments i, j.
- Remark: 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 O(n^3)).

matrix.**rank**(*A*)
rank(A) <-> A.rank computes the rank of A (in O(n^3)).

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

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

matrix.**inv**(*A*)
inv(A) <-> A.inv() tries to compute the inverse of A (in O(n^3)).

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

matrix.**PLUdecomposition**(*A*, *mode=None*)
matrix.PLUdecomposition(A) computes the permuted LU decomposition for the matrix A.

- Operates in time complexity of O(n^3), memory of O(n^2).
- mode can be None (default), or 'f' for fractions or 'd' for decimal numbers.
- Returned P, L, U that satisfy 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 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' <– L_i - gamma * 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.

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.

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. Empty product is 1.

matrix.**_ifnone**(*a*, *b*)
b if a is None else a.

- Useful for converting a slice object to a xrange object.

matrix.**_slice_to_xrange**(*sliceobject*)
Get a xrange of indeces from a slice object.

- Thanks to http://stackoverflow.com/a/13855369 !

matrix.**innerproduct**(*vx*, *vy*)
Dot product of the two vectors vx and vy (real numbers ONLY).

matrix.**norm_square**(*u*)
Shortcut for the square of the norm of the vector u : ||u||^2 = <u, u>.

matrix.**vect_const_multi**(*vx*, *c*)
Multiply the vector vx by the (real) constant c.

matrix.**proj**(*u*, *v*)
Projection of the vector v into the vector u (proj_u(v) as written on Wikipedia).

matrix.**gram_schmidt**(*V*, *normalized=False*)
Basic implementation of the Gram-Schmidt process, in the easy case of Rn with the usual <.,.>.

- The matrix is interpreted as a family of *column* vectors.
- Reference for notations, concept and proof is https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process.
- If normalized is True, the vectors are normalized before being returned.

matrix.**minor**(*A*, *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 O(n^2), time is O(n^3) (1 determinant of size n-1).

matrix.**cofactor**(*A*, *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.

- Complexities: memory is O(n^2), time is O(n^3) (1 determinant of size n-1).

matrix.**adjugate**(*A*)
adjugate(A) returns the adjugate matrix of A.

- Reference is https://en.wikipedia.org/wiki/Adjugate_matrix#Inverses.
- Complexities: memory is O(n^2), time is 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 O(n^3).

1.7 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: Tue Apr 07 14:09:03 2015.
- @author: Lilian Besson for CS101 course at Mahindra Ecole Centrale 2015.
- @licence: MIT Licence (http://lbesson.mit-license.org).

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Copyrights

m

t

Symbols

A

C

D

E

F

G

I