(continued from previous page)

```
>>> B = B.as_explicit()
>>> P.T * B * P == A
True
```

table(*printer*, *rowstart='['*, *rowend=']'*, *rowsep='\n'*, *colsep=', '*, *align='right'*)

String form of Matrix as a table.

`printer` is the printer to use for on the elements (generally something like StrPrinter())

`rowstart` is the string used to start each row (by default '[').

`rowend` is the string used to end each row (by default ']').

`rowsep` is the string used to separate rows (by default a newline).

`colsep` is the string used to separate columns (by default ', ').

`align` defines how the elements are aligned. Must be one of 'left', 'right', or 'center'. You can also use '<', '>', and '^' to mean the same thing, respectively.

This is used by the string printer for Matrix.

Examples

```
>>> from sympy import Matrix, StrPrinter
>>> M = Matrix([[1, 2], [-33, 4]])
>>> printer = StrPrinter()
>>> M.table(printer)
'[  1, 2]\n[-33, 4]'
>>> print(M.table(printer))
[  1, 2]
[-33, 4]
>>> print(M.table(printer, rowsep=',\n'))
[  1, 2],
[-33, 4]
>>> print('[%s]' % M.table(printer, rowsep=',\n'))
[[  1, 2],
[-33, 4]]
>>> print(M.table(printer, colsep=' '))
[  1 2]
[-33 4]
>>> print(M.table(printer, align='center'))
[ 1 , 2]
[-33, 4]
>>> print(M.table(printer, rowstart='{', rowend='}'))
{  1, 2}
{-33, 4}
```

upper_hessenberg_decomposition()

Converts a matrix into Hessenberg matrix H

Returns 2 matrices H, P s.t. $PHP^T = A$, where H is an upper hessenberg matrix and P is an orthogonal matrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([
...     [1,2,3],
...     [-3,5,6],
...     [4,-8,9],
... ])
>>> H, P = A.upper_hessenberg_decomposition()
>>> H
Matrix([
[1,    6/5,    17/5],
[5, 213/25, -134/25],
[0, 216/25,  137/25]])
>>> P
Matrix([
[1,    0,   0],
[0, -3/5, 4/5],
[0,  4/5, 3/5]])
>>> P * H * P.H == A
True
```

References

upper_triangular_solve(*rhs*)

Solves Ax = B, where A is an upper triangular matrix.

See also:

lower_triangular_solve (page 1304), *gauss_jordan_solve* (page 1297), *cholesky_solve* (page 1292), *diagonal_solve* (page 1295), *LDLsolve* (page 1282), *LUsolve* (page 1287), *QRsolve* (page 1291), *pinv_solve* (page 1306)

Matrix Exceptions Reference

class sympy.matrices.matrices.**MatrixError**

class sympy.matrices.matrices.**ShapeError**

Wrong matrix shape

class sympy.matrices.matrices.**NonSquareMatrixError**

Matrix Functions Reference

`sympy.matrices.dense.`**`matrix_multiply_elementwise`**`(`*A*, *B*`)`

Return the Hadamard product (elementwise product) of A and B

```
>>> from sympy import Matrix, matrix_multiply_elementwise
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> matrix_multiply_elementwise(A, B)
Matrix([
[  0, 10, 200],
[300, 40,   5]])
```

See also:

sympy.matrices.common.MatrixCommon.__mul__ (page 1327)

`sympy.matrices.dense.`**`zeros`**`(`**args*, ***kwargs*`)`

Returns a matrix of zeros with `rows` rows and `cols` columns; if `cols` is omitted a square matrix will be returned.

See also:

ones (page 1319), *eye* (page 1319), *diag* (page 1319)

`sympy.matrices.dense.`**`ones`**`(`**args*, ***kwargs*`)`

Returns a matrix of ones with `rows` rows and `cols` columns; if `cols` is omitted a square matrix will be returned.

See also:

zeros (page 1319), *eye* (page 1319), *diag* (page 1319)

`sympy.matrices.dense.`**`eye`**`(`**args*, ***kwargs*`)`

Create square identity matrix n x n

See also:

diag (page 1319), *zeros* (page 1319), *ones* (page 1319)

`sympy.matrices.dense.`**`diag`**`(`**values*, *strict=True*, *unpack=False*, ***kwargs*`)`

Returns a matrix with the provided values placed on the diagonal. If non-square matrices are included, they will produce a block-diagonal matrix.

Examples

This version of diag is a thin wrapper to Matrix.diag that differs in that it treats all lists like matrices – even when a single list is given. If this is not desired, either put a $*$ before the list or set $unpack = True$.

```
>>> from sympy import diag
```

```
>>> diag([1, 2, 3], unpack=True)  # = diag(1,2,3) or diag(*[1,2,3])
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

```
>>> diag([1, 2, 3])  # a column vector
Matrix([
[1],
[2],
[3]])
```

See also:

common.MatrixCommon.eye (page 1334), *common.MatrixCommon.diagonal* (page 1332), *common.MatrixCommon.diag* (page 1331), *expressions.blockmatrix.BlockMatrix* (page 1380)

sympy.matrices.dense.**jordan_cell**(*eigenval, n*)

Create a Jordan block:

Examples

```
>>> from sympy import jordan_cell
>>> from sympy.abc import x
>>> jordan_cell(x, 4)
Matrix([
[x, 1, 0, 0],
[0, x, 1, 0],
[0, 0, x, 1],
[0, 0, 0, x]])
```

sympy.matrices.dense.**hessian**(*f, varlist, constraints=()*)

Compute Hessian matrix for a function f wrt parameters in varlist which may be given as a sequence or a row/column vector. A list of constraints may optionally be given.

Examples

```
>>> from sympy import Function, hessian, pprint
>>> from sympy.abc import x, y
>>> f = Function('f')(x, y)
>>> g1 = Function('g')(x, y)
>>> g2 = x**2 + 3*y
>>> pprint(hessian(f, (x, y), [g1, g2]))
[                    d               d            ]
[     0        0    --(g(x, y))     --(g(x, y))  ]
[                    dx              dy           ]
[                                                 ]
[     0        0        2*x              3        ]
[                                                 ]
[                      2               2          ]
[d                    d               d           ]
[--(g(x, y))  2*x   ---(f(x, y))   -----(f(x, y))]
[dx                    2           dy dx          ]
[                    dx                           ]
[                                                 ]
```

(continues on next page)

(continued from previous page)

```
[                      2                2          ]
[d                    d                d           ]
[--(g(x, y))   3   -----(f(x, y))   ---(f(x, y))   ]
[dy                dy dx               2           ]
[                                    dy            ]
```

See also:

sympy.matrices.matrices.MatrixCalculus.jacobian (page 1279), *wronskian* (page 1321)

References

[R596]

`sympy.matrices.dense.`**`GramSchmidt`**`(`*vlist, orthonormal=False*`)`

Apply the Gram-Schmidt process to a set of vectors.

Parameters

vlist : List of Matrix

Vectors to be orthogonalized for.

orthonormal : Bool, optional

If true, return an orthonormal basis.

Returns

vlist : List of Matrix

Orthogonalized vectors

Notes

This routine is mostly duplicate from `Matrix.orthogonalize`, except for some difference that this always raises error when linearly dependent vectors are found, and the keyword `normalize` has been named as `orthonormal` in this function.

See also:

matrices.MatrixSubspaces.orthogonalize (page 1236)

References

[R597]

`sympy.matrices.dense.`**`wronskian`**`(`*functions, var, method='bareiss'*`)`

Compute Wronskian for [] of functions

```
                 | f1       f2        ...   fn      |
                 | f1'      f2'       ...   fn'     |
                 |  .        .        .      .      |
W(f1, ..., fn) = |  .        .         .     .      |
                 |  .        .          .    .      |
```

(continues on next page)

(continued from previous page)

```
                         |  (n)      (n)            (n)     |
                         | D    (f1) D    (f2)  ...  D    (fn) |
```

see: https://en.wikipedia.org/wiki/Wronskian

See also:

sympy.matrices.matrices.MatrixCalculus.jacobian (page 1279), *hessian* (page 1320)

`sympy.matrices.dense.`**`casoratian`**(*seqs*, *n*, *zero=True*)

Given linear difference operator L of order 'k' and homogeneous equation Ly = 0 we want to compute kernel of L, which is a set of 'k' sequences: a(n), b(n), … z(n).

Solutions of L are linearly independent iff their Casoratian, denoted as C(a, b, …, z), do not vanish for n = 0.

Casoratian is defined by k x k determinant:

```
+  a(n)     b(n)     . . . z(n)     +
|  a(n+1)   b(n+1)   . . . z(n+1)   |
|    .         .     .        .     |
|    .         .       .      .     |
|    .         .         .    .     |
+  a(n+k-1) b(n+k-1) . . . z(n+k-1) +
```

It proves very useful in rsolve_hyper() where it is applied to a generating set of a recurrence to factor out linearly dependent solutions and return a basis:

```
>>> from sympy import Symbol, casoratian, factorial
>>> n = Symbol('n', integer=True)
```

Exponential and factorial are linearly independent:

```
>>> casoratian([2**n, factorial(n)], n) != 0
True
```

`sympy.matrices.dense.`**`randMatrix`**(*r*, *c=None*, *min=0*, *max=99*, *seed=None*, *symmetric=False*, *percent=100*, *prng=None*)

Create random matrix with dimensions `r x c`. If `c` is omitted the matrix will be square. If `symmetric` is True the matrix must be square. If `percent` is less than 100 then only approximately the given percentage of elements will be non-zero.

The pseudo-random number generator used to generate matrix is chosen in the following way.

- If `prng` is supplied, it will be used as random number generator. It should be an instance of `random.Random`, or at least have `randint` and `shuffle` methods with same signatures.
- if `prng` is not supplied but `seed` is supplied, then new `random.Random` with given `seed` will be created;
- otherwise, a new `random.Random` with default seed will be used.

Examples

```
>>> from sympy import randMatrix
>>> randMatrix(3)
[25, 45, 27]
[44, 54,  9]
[23, 96, 46]
>>> randMatrix(3, 2)
[87, 29]
[23, 37]
[90, 26]
>>> randMatrix(3, 3, 0, 2)
[0, 2, 0]
[2, 0, 1]
[0, 0, 1]
>>> randMatrix(3, symmetric=True)
[85, 26, 29]
[26, 71, 43]
[29, 43, 57]
>>> A = randMatrix(3, seed=1)
>>> B = randMatrix(3, seed=2)
>>> A == B
False
>>> A == randMatrix(3, seed=1)
True
>>> randMatrix(3, symmetric=True, percent=50)
[77, 70,  0],
[70,  0,  0],
[ 0,  0, 88]
```

Numpy Utility Functions Reference

`sympy.matrices.dense.`**`list2numpy`**(*l*, *dtype=<class 'object'>*)

Converts Python list of SymPy expressions to a NumPy array.

See also:

`matrix2numpy` (page 1323)

`sympy.matrices.dense.`**`matrix2numpy`**(*m*, *dtype=<class 'object'>*)

Converts SymPy's matrix to a NumPy array.

See also:

`list2numpy` (page 1323)

`sympy.matrices.dense.`**`symarray`**(*prefix*, *shape*, ***kwargs*)

Create a numpy ndarray of symbols (as an object array).

The created symbols are named `prefix_i1_i2_`... You should thus provide a non-empty prefix if you want your symbols to be unique for different output arrays, as SymPy symbols with identical names are the same object.

Parameters

prefix : string

A prefix prepended to the name of every symbol.

shape : int or tuple

Shape of the created array. If an int, the array is one-dimensional; for more than one dimension the shape must be a tuple.

****kwargs** : dict

keyword arguments passed on to Symbol

Examples

These doctests require numpy.

```
>>> from sympy import symarray
>>> symarray('', 3)
[_0 _1 _2]
```

If you want multiple symarrays to contain distinct symbols, you *must* provide unique prefixes:

```
>>> a = symarray('', 3)
>>> b = symarray('', 3)
>>> a[0] == b[0]
True
>>> a = symarray('a', 3)
>>> b = symarray('b', 3)
>>> a[0] == b[0]
False
```

Creating symarrays with a prefix:

```
>>> symarray('a', 3)
[a_0 a_1 a_2]
```

For more than one dimension, the shape must be given as a tuple:

```
>>> symarray('a', (2, 3))
[[a_0_0 a_0_1 a_0_2]
 [a_1_0 a_1_1 a_1_2]]
>>> symarray('a', (2, 3, 2))
[[[a_0_0_0 a_0_0_1]
  [a_0_1_0 a_0_1_1]
  [a_0_2_0 a_0_2_1]]

 [[a_1_0_0 a_1_0_1]
  [a_1_1_0 a_1_1_1]
  [a_1_2_0 a_1_2_1]]]
```

For setting assumptions of the underlying Symbols:

```
>>> [s.is_real for s in symarray('a', 2, real=True)]
[True, True]
```

sympy.matrices.dense.**rot_axis1**(*theta*)

Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.

Examples

```
>>> from sympy import pi, rot_axis1
```

A rotation of pi/3 (60 degrees):

```
>>> theta = pi/3
>>> rot_axis1(theta)
Matrix([
[1,          0,          0],
[0,        1/2, sqrt(3)/2],
[0, -sqrt(3)/2,        1/2]])
```

If we rotate by pi/2 (90 degrees):

```
>>> rot_axis1(pi/2)
Matrix([
[1,  0, 0],
[0,  0, 1],
[0, -1, 0]])
```

See also:

rot_axis2 **(page 1325)**
Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

rot_axis3 **(page 1326)**
Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

sympy.matrices.dense.**rot_axis2**(*theta*)

Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.

Examples

```
>>> from sympy import pi, rot_axis2
```

A rotation of pi/3 (60 degrees):

```
>>> theta = pi/3
>>> rot_axis2(theta)
Matrix([
[      1/2, 0, -sqrt(3)/2],
[        0, 1,          0],
[sqrt(3)/2, 0,        1/2]])
```

If we rotate by pi/2 (90 degrees):

```
>>> rot_axis2(pi/2)
Matrix([
[0, 0, -1],
[0, 1,  0],
[1, 0,  0]])
```

See also:

rot_axis1 **(page 1324)**
Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

rot_axis3 **(page 1326)**
Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

sympy.matrices.dense.**rot_axis3**(*theta*)

Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.

Examples

```
>>> from sympy import pi, rot_axis3
```

A rotation of pi/3 (60 degrees):

```
>>> theta = pi/3
>>> rot_axis3(theta)
Matrix([
[       1/2, sqrt(3)/2, 0],
[-sqrt(3)/2,       1/2, 0],
[         0,         0, 1]])
```

If we rotate by pi/2 (90 degrees):

```
>>> rot_axis3(pi/2)
Matrix([
[ 0, 1, 0],
[-1, 0, 0],
[ 0, 0, 1]])
```

See also:

rot_axis1 **(page 1324)**
Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

rot_axis2 **(page 1325)**
Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

sympy.matrices.matrices.**a2idx**(*j*, *n=None*)

Common Matrices

MatrixCommon Class Reference

class sympy.matrices.common.**MatrixCommon**

All common matrix operations including basic arithmetic, shaping, and special matrices like $zeros$, and eye.

property C

By-element conjugation

property H

Return Hermite conjugate.

Examples

```
>>> from sympy import Matrix, I
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m
Matrix([
[    0],
[1 + I],
[    2],
[    3]])
>>> m.H
Matrix([[0, 1 - I, 2, 3]])
```

See also:

conjugate **(page 1330)**
By-element conjugation

sympy.matrices.matrices.MatrixBase.D **(page 1280)**
Dirac conjugation

property T

Matrix transposition

__abs__()

Returns a new matrix with entry-wise absolute values.

__add__(*other*)

Return self + other, raising ShapeError if shapes do not match.

__getitem__(*key*)

Implementations of __getitem__ should accept ints, in which case the matrix is indexed as a flat list, tuples (i,j) in which case the (i,j) entry is returned, slices, or mixed tuples (a,b) where a and b are any combination of slices and integers.

__len__()

The total number of entries in the matrix.

__mul__(*other*)

Return self*other where other is either a scalar or a matrix of compatible dimensions.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> 2*A == A*2 == Matrix([[2, 4, 6], [8, 10, 12]])
True
>>> B = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> A*B
Matrix([
[30, 36, 42],
[66, 81, 96]])
>>> B*A
Traceback (most recent call last):
...
ShapeError: Matrices size mismatch.
>>>
```

See also:

matrix_multiply_elementwise (page 1319)

__pow__(*exp*)
Return self**exp a scalar or symbol.

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

adjoint()
Conjugate transpose or Hermitian conjugation.

applyfunc(*f*)
Apply a function to each element of the matrix.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, lambda i, j: i*2+j)
>>> m
Matrix([
[0, 1],
[2, 3]])
>>> m.applyfunc(lambda i: 2*i)
Matrix([
[0, 2],
[4, 6]])
```

as_real_imag(*deep=True*, ***hints*)
Returns a tuple containing the (real, imaginary) part of matrix.

atoms(**types*)
Returns the atoms that form the current object.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import Matrix
>>> Matrix([[x]])
Matrix([[x]])
>>> _.atoms()
{x}
>>> Matrix([[x, y], [y, x]])
Matrix([
[x, y],
[y, x]])
>>> _.atoms()
{x, y}
```

col(*j*)

Elementary column selector.

Examples

```
>>> from sympy import eye
>>> eye(2).col(0)
Matrix([
[1],
[0]])
```

See also:

row (page 1352), *col_del* (page 1329), *col_join* (page 1329), *col_insert* (page 1329)

col_del(*col*)

Delete the specified column.

col_insert(*pos, other*)

Insert one or more columns at the given column position.

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.col_insert(1, V)
Matrix([
[0, 1, 0, 0],
[0, 1, 0, 0],
[0, 1, 0, 0]])
```

See also:

col (page 1329), *row_insert* (page 1352)

col_join(*other*)

Concatenates two matrices along self's last and other's first row.

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.col_join(V)
Matrix([
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[1, 1, 1]])
```

See also:

col (page 1329), *row_join* (page 1353)

classmethod companion(*poly*)

Returns a companion matrix of a polynomial.

Examples

```
>>> from sympy import Matrix, Poly, Symbol, symbols
>>> x = Symbol('x')
>>> c0, c1, c2, c3, c4 = symbols('c0:5')
>>> p = Poly(c0 + c1*x + c2*x**2 + c3*x**3 + c4*x**4 + x**5, x)
>>> Matrix.companion(p)
Matrix([
[0, 0, 0, 0, -c0],
[1, 0, 0, 0, -c1],
[0, 1, 0, 0, -c2],
[0, 0, 1, 0, -c3],
[0, 0, 0, 1, -c4]])
```

conjugate()

Return the by-element conjugation.

Examples

```
>>> from sympy import SparseMatrix, I
>>> a = SparseMatrix(((1, 2 + I), (3, 4), (I, -I)))
>>> a
Matrix([
[1, 2 + I],
[3,     4],
[I,    -I]])
>>> a.C
```

(continues on next page)

(continued from previous page)

```
Matrix([
[ 1, 2 - I],
[ 3,     4],
[-I,     I]])
```

See also:

transpose **(page 1355)**
    Matrix transposition

H **(page 1327)**
    Hermite conjugation

sympy.matrices.matrices.MatrixBase.D **(page 1280)**
    Dirac conjugation

classmethod diag(**args, strict=False, unpack=True, rows=None, cols=None, **kwargs*)

Returns a matrix with the specified diagonal. If matrices are passed, a block-diagonal matrix is created (i.e. the "direct sum" of the matrices).

Kwargs

rows
    [rows of the resulting matrix; computed if] not given.

cols
    [columns of the resulting matrix; computed if] not given.

cls : class for the resulting matrix

unpack : bool which, when True (default), unpacks a single sequence rather than interpreting it as a Matrix.

strict : bool which, when False (default), allows Matrices to have variable-length rows.

Examples

```
>>> from sympy import Matrix
>>> Matrix.diag(1, 2, 3)
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

The current default is to unpack a single sequence. If this is not desired, set $unpack = False$ and it will be interpreted as a matrix.

```
>>> Matrix.diag([1, 2, 3]) == Matrix.diag(1, 2, 3)
True
```

When more than one element is passed, each is interpreted as something to put on the diagonal. Lists are converted to matrices. Filling of the diagonal always continues from the bottom right hand corner of the previous item: this will create a block-diagonal matrix whether the matrices are square or not.

```
>>> col = [1, 2, 3]
>>> row = [[4, 5]]
>>> Matrix.diag(col, row)
Matrix([
[1, 0, 0],
[2, 0, 0],
[3, 0, 0],
[0, 4, 5]])
```

When *unpack* is False, elements within a list need not all be of the same length. Setting *strict* to True would raise a ValueError for the following:

```
>>> Matrix.diag([[1, 2, 3], [4, 5], [6]], unpack=False)
Matrix([
[1, 2, 3],
[4, 5, 0],
[6, 0, 0]])
```

The type of the returned matrix can be set with the `cls` keyword.

```
>>> from sympy import ImmutableMatrix
>>> from sympy.utilities.misc import func_name
>>> func_name(Matrix.diag(1, cls=ImmutableMatrix))
'ImmutableDenseMatrix'
```

A zero dimension matrix can be used to position the start of the filling at the start of an arbitrary row or column:

```
>>> from sympy import ones
>>> r2 = ones(0, 2)
>>> Matrix.diag(r2, 1, 2)
Matrix([
[0, 0, 1, 0],
[0, 0, 0, 2]])
```

See also:

eye (page 1334), *diagonal* (page 1332), *dense.diag* (page 1319), *expressions.blockmatrix.BlockMatrix* (page 1380), *sparsetools.banded* (page 1366)

diagonal(*k=0*)

Returns the kth diagonal of self. The main diagonal corresponds to $k = 0$; diagonals above and below correspond to $k > 0$ and $k < 0$, respectively. The values of $self[i, j]$ for which $j - i = k$, are returned in order of increasing $i + j$, starting with $i + j = |k|$.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(3, 3, lambda i, j: j - i); m
Matrix([
[ 0,  1, 2],
[-1,  0, 1],
[-2, -1, 0]])
>>> _.diagonal()
Matrix([[0, 0, 0]])
>>> m.diagonal(1)
Matrix([[1, 1]])
>>> m.diagonal(-2)
Matrix([[-2]])
```

Even though the diagonal is returned as a Matrix, the element retrieval can be done with a single index:

```
>>> Matrix.diag(1, 2, 3).diagonal()[1]  # instead of [0, 1]
2
```

See also:

`diag` (page 1331)

`evalf`(*n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False*)

Apply evalf() to each element of self.

`expand`(*deep=True, modulus=None, power_base=True, power_exp=True, mul=True, log=True, multinomial=True, basic=True, **hints*)

Apply core.function.expand to each entry of the matrix.

Examples

```
>>> from sympy.abc import x
>>> from sympy import Matrix
>>> Matrix(1, 1, [x*(x+1)])
Matrix([[x*(x + 1)]])
>>> _.expand()
Matrix([[x**2 + x]])
```

`extract`(*rowsList, colsList*)

Return a submatrix by specifying a list of rows and columns. Negative indices can be given. All indices must be in the range $-n \leq i < n$ where n is the number of rows or columns.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(4, 3, range(12))
>>> m
Matrix([
[0,  1,  2],
[3,  4,  5],
[6,  7,  8],
[9, 10, 11]])
>>> m.extract([0, 1, 3], [0, 1])
Matrix([
[0,  1],
[3,  4],
[9, 10]])
```

Rows or columns can be repeated:

```
>>> m.extract([0, 0, 1], [-1])
Matrix([
[2],
[2],
[5]])
```

Every other row can be taken by using range to provide the indices:

```
>>> m.extract(range(0, m.rows, 2), [-1])
Matrix([
[2],
[8]])
```

RowsList or colsList can also be a list of booleans, in which case the rows or columns corresponding to the True values will be selected:

```
>>> m.extract([0, 1, 2, 3], [True, False, True])
Matrix([
[0,  2],
[3,  5],
[6,  8],
[9, 11]])
```

classmethod eye(*rows*, *cols=None*, ***kwargs*)

Returns an identity matrix.

Args

rows : rows of the matrix cols : cols of the matrix (if None, cols=rows)

Kwargs

cls : class of the returned matrix

property free_symbols

Returns the free symbols within the matrix.

Examples

```
>>> from sympy.abc import x
>>> from sympy import Matrix
>>> Matrix([[x], [1]]).free_symbols
{x}
```

get_diag_blocks()

Obtains the square sub-matrices on the main diagonal of a square matrix.

Useful for inverting symbolic matrices or solving systems of linear equations which may be decoupled by having a block diagonal structure.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z
>>> A = Matrix([[1, 3, 0, 0], [y, z*z, 0, 0], [0, 0, x, 0], [0, 0, 0,␣
↪0]])
>>> a1, a2, a3 = A.get_diag_blocks()
>>> a1
Matrix([
[1,    3],
[y, z**2]])
>>> a2
Matrix([[x]])
>>> a3
Matrix([[0]])
```

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from sympy import Matrix, SparseMatrix, Float
>>> from sympy.abc import x, y
>>> A = Matrix(((1, x), (0.2, 3)))
>>> B = SparseMatrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
>>> B.has(x)
True
>>> B.has(y)
False
>>> B.has(Float)
True
```

classmethod hstack(**args*)

Return a matrix formed by joining args horizontally (i.e. by repeated application of row_join).

Examples

```
>>> from sympy import Matrix, eye
>>> Matrix.hstack(eye(2), 2*eye(2))
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2]])
```

is_anti_symmetric(*simplify=True*)

Check if matrix M is an antisymmetric matrix, that is, M is a square matrix with all M[i, j] == -M[j, i].

When `simplify=True` (default), the sum M[i, j] + M[j, i] is simplified before testing to see if it is zero. By default, the SymPy simplify function is used. To use a custom function set simplify to a function that accepts a single argument which returns a simplified expression. To skip simplification, set simplify to False but note that although this will be faster, it may induce false negatives.

Examples

```
>>> from sympy import Matrix, symbols
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[ 0, 1],
[-1, 0]])
>>> m.is_anti_symmetric()
```

(continues on next page)

(continued from previous page)

```
True
>>> x, y = symbols('x y')
>>> m = Matrix(2, 3, [0, 0, x, -y, 0, 0])
>>> m
Matrix([
[ 0, 0, x],
[-y, 0, 0]])
>>> m.is_anti_symmetric()
False
```

```
>>> from sympy.abc import x, y
>>> m = Matrix(3, 3, [0, x**2 + 2*x + 1, y,
...                   -(x + 1)**2, 0, x*y,
...                   -y, -x*y, 0])
```

Simplification of matrix elements is done by default so even though two elements which should be equal and opposite would not pass an equality test, the matrix is still reported as anti-symmetric:

```
>>> m[0, 1] == -m[1, 0]
False
>>> m.is_anti_symmetric()
True
```

If simplify=False is used for the case when a Matrix is already simplified, this will speed things up. Here, we see that without simplification the matrix does not appear anti-symmetric:

```
>>> m.is_anti_symmetric(simplify=False)
False
```

But if the matrix were already expanded, then it would appear anti-symmetric and simplification in the is_anti_symmetric routine is not needed:

```
>>> m = m.expand()
>>> m.is_anti_symmetric(simplify=False)
True
```

is_diagonal()

Check if matrix is diagonal, that is matrix in which the entries outside the main diagonal are all zero.

Examples

```
>>> from sympy import Matrix, diag
>>> m = Matrix(2, 2, [1, 0, 0, 2])
>>> m
Matrix([
[1, 0],
[0, 2]])
```

(continues on next page)

(continued from previous page)

```
>>> m.is_diagonal()
True
```

```
>>> m = Matrix(2, 2, [1, 1, 0, 2])
>>> m
Matrix([
[1, 1],
[0, 2]])
>>> m.is_diagonal()
False
```

```
>>> m = diag(1, 2, 3)
>>> m
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> m.is_diagonal()
True
```

See also:

is_lower (page 1338), *is_upper* (page 1342), *sympy.matrices.matrices.MatrixEigen.is_diagonalizable* (page 1241), *diagonalize* (page 1238)

property is_hermitian

Checks if the matrix is Hermitian.

In a Hermitian matrix element i,j is the complex conjugate of element j,i.

Examples

```
>>> from sympy import Matrix
>>> from sympy import I
>>> from sympy.abc import x
>>> a = Matrix([[1, I], [-I, 1]])
>>> a
Matrix([
[ 1, I],
[-I, 1]])
>>> a.is_hermitian
True
>>> a[0, 0] = 2*I
>>> a.is_hermitian
False
>>> a[0, 0] = x
>>> a.is_hermitian
>>> a[0, 1] = a[1, 0]*I
>>> a.is_hermitian
False
```

property is_lower

Check if matrix is a lower triangular matrix. True can be returned even if the matrix is not square.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_lower
True
```

```
>>> m = Matrix(4, 3, [0, 0, 0, 2, 0, 0, 1, 4, 0, 6, 6, 5])
>>> m
Matrix([
[0, 0, 0],
[2, 0, 0],
[1, 4, 0],
[6, 6, 5]])
>>> m.is_lower
True
```

```
>>> from sympy.abc import x, y
>>> m = Matrix(2, 2, [x**2 + y, y**2 + x, 0, x + y])
>>> m
Matrix([
[x**2 + y, x + y**2],
[       0,    x + y]])
>>> m.is_lower
False
```

See also:

is_upper (page 1342), *is_diagonal* (page 1337), *is_lower_hessenberg* (page 1339)

property is_lower_hessenberg

Checks if the matrix is in the lower-Hessenberg form.

The lower hessenberg matrix has zero entries above the first superdiagonal.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[1, 2, 0, 0], [5, 2, 3, 0], [3, 4, 3, 7], [5, 6, 1,␣
↪1]])
>>> a
Matrix([
[1, 2, 0, 0],
[5, 2, 3, 0],
[3, 4, 3, 7],
[5, 6, 1, 1]])
>>> a.is_lower_hessenberg
True
```

See also:

is_upper_hessenberg (page 1343), *is_lower* (page 1338)

property is_square

Checks if a matrix is square.

A matrix is square if the number of rows equals the number of columns. The empty matrix is square by definition, since the number of rows and the number of columns are both zero.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[1, 2, 3], [4, 5, 6]])
>>> b = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> c = Matrix([])
>>> a.is_square
False
>>> b.is_square
True
>>> c.is_square
True
```

property is_strongly_diagonally_dominant

Tests if the matrix is row strongly diagonally dominant.

Explanation

A n, n matrix A is row strongly diagonally dominant if

$$|A_{i,i}| > \sum_{j=0, j \neq i}^{n-1} |A_{i,j}| \quad \text{for all } i \in \{0, ..., n-1\}$$

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[3, -2, 1], [1, -3, 2], [-1, 2, 4]])
>>> A.is_strongly_diagonally_dominant
False
```

```
>>> A = Matrix([[-2, 2, 1], [1, 3, 2], [1, -2, 0]])
>>> A.is_strongly_diagonally_dominant
False
```

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

Notes

If you want to test whether a matrix is column diagonally dominant, you can apply the test after transposing the matrix.

is_symbolic()

Checks if any elements contain Symbols.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.is_symbolic()
True
```

is_symmetric(*simplify=True*)

Check if matrix is symmetric matrix, that is square matrix and is equal to its transpose.

By default, simplifications occur before testing symmetry. They can be skipped using 'simplify=False'; while speeding things a bit, this may however induce false negatives.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [0, 1, 1, 2])
>>> m
Matrix([
[0, 1],
[1, 2]])
>>> m.is_symmetric()
True
```

```
>>> m = Matrix(2, 2, [0, 1, 2, 0])
>>> m
Matrix([
[0, 1],
[2, 0]])
>>> m.is_symmetric()
False
```

```
>>> m = Matrix(2, 3, [0, 0, 0, 0, 0, 0])
>>> m
Matrix([
[0, 0, 0],
[0, 0, 0]])
>>> m.is_symmetric()
False
```

```
>>> from sympy.abc import x, y
>>> m = Matrix(3, 3, [1, x**2 + 2*x + 1, y, (x + 1)**2, 2, 0, y, 0,␣
↪3])
>>> m
Matrix([
[         1, x**2 + 2*x + 1, y],
[(x + 1)**2,              2, 0],
[         y,              0, 3]])
>>> m.is_symmetric()
True
```

If the matrix is already simplified, you may speed-up is_symmetric() test by using ‘simplify=False’.

```
>>> bool(m.is_symmetric(simplify=False))
False
>>> m1 = m.expand()
>>> m1.is_symmetric(simplify=False)
True
```

property is_upper

Check if matrix is an upper triangular matrix. True can be returned even if the matrix is not square.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_upper
True
```

```
>>> m = Matrix(4, 3, [5, 1, 9, 0, 4, 6, 0, 0, 5, 0, 0, 0])
>>> m
Matrix([
[5, 1, 9],
[0, 4, 6],
[0, 0, 5],
[0, 0, 0]])
>>> m.is_upper
True
```

```
>>> m = Matrix(2, 3, [4, 2, 5, 6, 1, 1])
>>> m
Matrix([
[4, 2, 5],
[6, 1, 1]])
>>> m.is_upper
False
```

See also:

is_lower (page 1338), *is_diagonal* (page 1337), *is_upper_hessenberg* (page 1343)

property is_upper_hessenberg

Checks if the matrix is the upper-Hessenberg form.

The upper hessenberg matrix has zero entries below the first subdiagonal.

Examples

```
>>> from sympy import Matrix
>>> a = Matrix([[1, 4, 2, 3], [3, 4, 1, 7], [0, 2, 3, 4], [0, 0, 1,␣
↪3]])
>>> a
Matrix([
[1, 4, 2, 3],
[3, 4, 1, 7],
[0, 2, 3, 4],
[0, 0, 1, 3]])
>>> a.is_upper_hessenberg
True
```

See also:

is_lower_hessenberg (page 1339), *is_upper* (page 1342)

property is_weakly_diagonally_dominant

Tests if the matrix is row weakly diagonally dominant.

Explanation

A n, n matrix A is row weakly diagonally dominant if

$$|A_{i,i}| \ge \sum_{j=0, j\neq i}^{n-1} |A_{i,j}| \quad \text{for all } i \in \{0, ..., n-1\}$$

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[3, -2, 1], [1, -3, 2], [-1, 2, 4]])
>>> A.is_weakly_diagonally_dominant
True
```

```
>>> A = Matrix([[-2, 2, 1], [1, 3, 2], [1, -2, 0]])
>>> A.is_weakly_diagonally_dominant
False
```

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

Notes

If you want to test whether a matrix is column diagonally dominant, you can apply the test after transposing the matrix.

property is_zero_matrix

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

Examples

```
>>> from sympy import Matrix, zeros
>>> from sympy.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero_matrix
True
>>> b.is_zero_matrix
True
>>> c.is_zero_matrix
```

(continues on next page)

(continued from previous page)

```
False
>>> d.is_zero_matrix
True
>>> e.is_zero_matrix
```

classmethod jordan_block(*size=None*, *eigenvalue=None*, ***, *band='upper'*, ***kwargs*)

Returns a Jordan block

Parameters

size : Integer, optional

Specifies the shape of the Jordan block matrix.

eigenvalue : Number or Symbol

Specifies the value for the main diagonal of the matrix.

Note: The keyword `eigenval` is also specified as an alias of this keyword, but it is not recommended to use.

We may deprecate the alias in later release.

band : 'upper' or 'lower', optional

Specifies the position of the off-diagonal to put 1 s on.

cls : Matrix, optional

Specifies the matrix class of the output form.

If it is not specified, the class type where the method is being executed on will be returned.

rows, cols : Integer, optional

Specifies the shape of the Jordan block matrix. See Notes section for the details of how these key works.

Deprecated since version 1.4: The rows and cols parameters are deprecated and will be removed in a future version.

Returns

Matrix

A Jordan block matrix.

Raises

ValueError

If insufficient arguments are given for matrix size specification, or no eigenvalue is given.

Examples

Creating a default Jordan block:

```
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> Matrix.jordan_block(4, x)
Matrix([
[x, 1, 0, 0],
[0, x, 1, 0],
[0, 0, x, 1],
[0, 0, 0, x]])
```

Creating an alternative Jordan block matrix where 1 is on lower off-diagonal:

```
>>> Matrix.jordan_block(4, x, band='lower')
Matrix([
[x, 0, 0, 0],
[1, x, 0, 0],
[0, 1, x, 0],
[0, 0, 1, x]])
```

Creating a Jordan block with keyword arguments

```
>>> Matrix.jordan_block(size=4, eigenvalue=x)
Matrix([
[x, 1, 0, 0],
[0, x, 1, 0],
[0, 0, x, 1],
[0, 0, 0, x]])
```

Notes

Deprecated since version 1.4: This feature is deprecated and will be removed in a future version.

The keyword arguments `size`, `rows`, `cols` relates to the Jordan block size specifications.

If you want to create a square Jordan block, specify either one of the three arguments.

If you want to create a rectangular Jordan block, specify `rows` and `cols` individually.

Arguments Given			Matrix Shape	
size	rows	cols	rows	cols
size	Any		size	size
None	None		ValueError	
	rows	None	rows	rows
	None	cols	cols	cols
	rows	cols	rows	cols

References

[R562]

lower_triangular(*k=0*)

returns the elements on and below the kth diagonal of a matrix. If k is not specified then simply returns lower-triangular portion of a matrix

Examples

```
>>> from sympy import ones
>>> A = ones(4)
>>> A.lower_triangular()
Matrix([
[1, 0, 0, 0],
[1, 1, 0, 0],
[1, 1, 1, 0],
[1, 1, 1, 1]])
```

```
>>> A.lower_triangular(-2)
Matrix([
[0, 0, 0, 0],
[0, 0, 0, 0],
[1, 0, 0, 0],
[1, 1, 0, 0]])
```

```
>>> A.lower_triangular(1)
Matrix([
[1, 1, 0, 0],
[1, 1, 1, 0],
[1, 1, 1, 1],
[1, 1, 1, 1]])
```

multiply(*other, dotprodsimp=None*)

Same as __mul__() but with optional simplification.

Parameters

dotprodsimp : bool, optional

Specifies whether intermediate term algebraic simplification is used during matrix multiplications to control expression blowup and thus speed up calculation. Default is off.

multiply_elementwise(*other*)

Return the Hadamard product (elementwise product) of A and B

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> A.multiply_elementwise(B)
Matrix([
[  0, 10, 200],
[300, 40,   5]])
```

See also:

sympy.matrices.matrices.MatrixBase.cross (page 1295), *sympy.matrices.matrices.MatrixBase.dot* (page 1296), *multiply* (page 1347)

n(**args, **kwargs*)

Apply evalf() to each element of self.

classmethod ones(*rows, cols=None, **kwargs*)

Returns a matrix of ones.

Args

rows : rows of the matrix cols : cols of the matrix (if None, cols=rows)

Kwargs

cls : class of the returned matrix

permute(*perm, orientation='rows', direction='forward'*)

Permute the rows or columns of a matrix by the given list of swaps.

Parameters

perm : Permutation, list, or list of lists

A representation for the permutation.

If it is `Permutation`, it is used directly with some resizing with respect to the matrix size.

If it is specified as list of lists, (e.g., `[[0, 1], [0, 2]]`), then the permutation is formed from applying the product of cycles. The direction how the cyclic product is applied is described in below.

If it is specified as a list, the list should represent an array form of a permutation. (e.g., `[1, 2, 0]`) which would would form the swapping function $0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0$.

orientation : 'rows', 'cols'

A flag to control whether to permute the rows or the columns

direction : 'forward', 'backward'

A flag to control whether to apply the permutations from the start of the list first, or from the back of the list first.

For example, if the permutation specification is `[[0, 1], [0, 2]]`, If the flag is set to 'forward', the cycle would be formed as $0 \mapsto 2, 2 \mapsto 1, 1 \mapsto 0$.

If the flag is set to 'backward', the cycle would be formed as $0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 0$.

If the argument perm is not in a form of list of lists, this flag takes no effect.

Examples

```
>>> from sympy import eye
>>> M = eye(3)
>>> M.permute([[0, 1], [0, 2]], orientation='rows', direction='forward
↪')
Matrix([
[0, 0, 1],
[1, 0, 0],
[0, 1, 0]])
```

```
>>> from sympy import eye
>>> M = eye(3)
>>> M.permute([[0, 1], [0, 2]], orientation='rows', direction=
↪'backward')
Matrix([
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]])
```

Notes

If a bijective function $\sigma : \mathbb{N}_0 \to \mathbb{N}_0$ denotes the permutation.

If the matrix A is the matrix to permute, represented as a horizontal or a vertical stack of vectors:

$$A = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_{n-1} \end{bmatrix}$$

If the matrix B is the result, the permutation of matrix rows is defined as:

$$B := \begin{bmatrix} a_{\sigma(0)} \\ a_{\sigma(1)} \\ \vdots \\ a_{\sigma(n-1)} \end{bmatrix}$$

And the permutation of matrix columns is defined as:

$$B := \begin{bmatrix} \alpha_{\sigma(0)} & \alpha_{\sigma(1)} & \cdots & \alpha_{\sigma(n-1)} \end{bmatrix}$$

permute_cols(*swaps, direction='forward'*)

Alias for `self.permute(swaps, orientation='cols', direction=direction)`

See also:

permute (page 1348)

permute_rows(*swaps, direction='forward'*)

Alias for `self.permute(swaps, orientation='rows', direction=direction)`

See also:

permute (page 1348)

pow(*exp, method=None*)

Return self**exp a scalar or symbol.

Parameters

method : multiply, mulsimp, jordan, cayley

If multiply then it returns exponentiation using recursion. If jordan then Jordan form exponentiation will be used. If cayley then the exponentiation is done using Cayley-Hamilton theorem. If mulsimp then the exponentiation is done using recursion with dotprodsimp. This specifies whether intermediate term algebraic simplification is used during naive matrix power to control expression blowup and thus speed up calculation. If None, then it heuristically decides which method to use.

refine(*assumptions=True*)

Apply refine to each element of the matrix.

Examples

```
>>> from sympy import Symbol, Matrix, Abs, sqrt, Q
>>> x = Symbol('x')
>>> Matrix([[Abs(x)**2, sqrt(x**2)],[sqrt(x**2), Abs(x)**2]])
Matrix([
[ Abs(x)**2, sqrt(x**2)],
[sqrt(x**2),  Abs(x)**2]])
>>> _.refine(Q.real(x))
Matrix([
[  x**2, Abs(x)],
[Abs(x),   x**2]])
```

replace(*F, G, map=False, simultaneous=True, exact=None*)

Replaces Function F in Matrix entries with Function G.

Examples

```
>>> from sympy import symbols, Function, Matrix
>>> F, G = symbols('F, G', cls=Function)
>>> M = Matrix(2, 2, lambda i, j: F(i+j)) ; M
Matrix([
[F(0), F(1)],
[F(1), F(2)]])
>>> N = M.replace(F,G)
>>> N
Matrix([
[G(0), G(1)],
[G(1), G(2)]])
```

reshape(*rows, cols*)

Reshape the matrix. Total number of elements must remain the same.

Examples

```
>>> from sympy import Matrix
>>> m = Matrix(2, 3, lambda i, j: 1)
>>> m
Matrix([
[1, 1, 1],
[1, 1, 1]])
>>> m.reshape(1, 6)
Matrix([[1, 1, 1, 1, 1, 1]])
>>> m.reshape(3, 2)
Matrix([
[1, 1],
[1, 1],
[1, 1]])
```

rmultiply(*other, dotprodsimp=None*)

Same as __rmul__() but with optional simplification.

Parameters

dotprodsimp : bool, optional

Specifies whether intermediate term algebraic simplification is used during matrix multiplications to control expression blowup and thus speed up calculation. Default is off.

rot90(*k=1*)

Rotates Matrix by 90 degrees

Parameters

k : int

Specifies how many times the matrix is rotated by 90 degrees (clockwise when positive, counter-clockwise when negative).

Examples

```
>>> from sympy import Matrix, symbols
>>> A = Matrix(2, 2, symbols('a:d'))
>>> A
Matrix([
[a, b],
[c, d]])
```

Rotating the matrix clockwise one time:

```
>>> A.rot90(1)
Matrix([
[c, a],
[d, b]])
```

Rotating the matrix anticlockwise two times:

```
>>> A.rot90(-2)
Matrix([
[d, c],
[b, a]])
```

row(*i*)

Elementary row selector.

Examples

```
>>> from sympy import eye
>>> eye(2).row(0)
Matrix([[1, 0]])
```

See also:

col (page 1329), *row_del* (page 1352), *row_join* (page 1353), *row_insert* (page 1352)

row_del(*row*)

Delete the specified row.

row_insert(*pos, other*)

Insert one or more rows at the given row position.

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.row_insert(1, V)
Matrix([
[0, 0, 0],
[1, 1, 1],
[0, 0, 0],
[0, 0, 0]])
```

See also:

row (page 1352), *col_insert* (page 1329)

row_join(*other*)

Concatenates two matrices along self's last and rhs's first column

Examples

```
>>> from sympy import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.row_join(V)
Matrix([
[0, 0, 0, 1],
[0, 0, 0, 1],
[0, 0, 0, 1]])
```

See also:

row (page 1352), *col_join* (page 1329)

property shape

The shape (dimensions) of the matrix as the 2-tuple (rows, cols).

Examples

```
>>> from sympy import zeros
>>> M = zeros(2, 3)
>>> M.shape
(2, 3)
>>> M.rows
2
>>> M.cols
3
```

simplify(***kwargs*)

Apply simplify to each element of the matrix.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import SparseMatrix, sin, cos
>>> SparseMatrix(1, 1, [x*sin(y)**2 + x*cos(y)**2])
Matrix([[x*sin(y)**2 + x*cos(y)**2]])
>>> _.simplify()
Matrix([[x]])
```

subs(**args, **kwargs*)

Return a new matrix with subs applied to each entry.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.subs(x, y)
Matrix([[y]])
>>> Matrix(_).subs(y, x)
Matrix([[x]])
```

todod()

Returns matrix as dict of dicts containing non-zero elements of the Matrix

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[0, 1],[0, 3]])
>>> A
Matrix([
[0, 1],
[0, 3]])
>>> A.todod()
{0: {1: 1}, 1: {1: 3}}
```

todok()

Return the matrix as dictionary of keys.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix.eye(3)
>>> M.todok()
{(0, 0): 1, (1, 1): 1, (2, 2): 1}
```

tolist()

Return the Matrix as a nested Python list.

Examples

```
>>> from sympy import Matrix, ones
>>> m = Matrix(3, 3, range(9))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8]])
>>> m.tolist()
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> ones(3, 0).tolist()
[[], [], []]
```

When there are no rows then it will not be possible to tell how many columns were in the original matrix:

```
>>> ones(0, 3).tolist()
[]
```

trace()

Returns the trace of a square matrix i.e. the sum of the diagonal elements.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.trace()
5
```

transpose()

Returns the transpose of the matrix.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.transpose()
Matrix([
[1, 3],
[2, 4]])
```

```
>>> from sympy import Matrix, I
>>> m=Matrix(((1, 2+I), (3, 4)))
>>> m
Matrix([
[1, 2 + I],
[3,     4]])
>>> m.transpose()
Matrix([
[    1, 3],
[2 + I, 4]])
>>> m.T == m.transpose()
True
```

See also:

conjugate **(page 1330)**
By-element conjugation

upper_triangular(*k=0*)

returns the elements on and above the kth diagonal of a matrix. If k is not specified then simply returns upper-triangular portion of a matrix

Examples

```
>>> from sympy import ones
>>> A = ones(4)
>>> A.upper_triangular()
Matrix([
[1, 1, 1, 1],
[0, 1, 1, 1],
[0, 0, 1, 1],
[0, 0, 0, 1]])
```

```
>>> A.upper_triangular(2)
Matrix([
[0, 0, 1, 1],
[0, 0, 0, 1],
[0, 0, 0, 0],
[0, 0, 0, 0]])
```