Matroid Theory

Release 10.6

The Sage Development Team

Jun 27, 2025

CONTENTS

CHAPTER

ONE

BASICS

1.1 Matroid construction

1.1.1 Theory

Matroids are combinatorial structures that capture the abstract properties of (linear/algebraic/…) dependence. Formally, a matroid is a pair $M = (E, I)$ of a finite set E, the *groundset*, and a collection of subsets I, the independent sets, subject to the following axioms:

- I contains the empty set
- If X is a set in I, then each subset of X is in I
- If two subsets X, Y are in I, and $|X| > |Y|$, then there exists $x \in X - Y$ such that $Y + \{x\}$ is in I.

See the Wikipedia article on matroids for more theory and examples. Matroids can be obtained from many types of mathematical structures, and Sage supports a number of them.

There are two main entry points to Sage's matroid functionality. The object *matroids.* contains a number of constructors for well-known matroids. The function *Matroid()* allows you to define your own matroids from a variety of sources. We briefly introduce both below; follow the links for more comprehensive documentation.

Each matroid object in Sage comes with a number of built-in operations. An overview can be found in the documentation of *the abstract matroid class*.

1.1.2 Built-in matroids

For built-in matroids, do the following:

- Within a Sage session, type `matroids.` (Do not press `Enter`, and do not forget the final period ".")
- Hit `Tab`.

You will see a list of methods which will construct matroids. For example:

```
sage: M = matroids.Wheel(4)
sage: M.is_connected()
True
```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(4))
>>> M.is_connected()
True
```

or:

```
sage: U36 = matroids.Uniform(3, 6)
sage: U36.equals(U36.dual())
True
```

```
>>> from sage.all import *
>>> U36 = matroids.Uniform(Integer(3), Integer(6))
>>> U36.equals(U36.dual())
True
```

A number of special matroids are collected under a `catalog` submenu. To see which, type `matroids.catalog.<tab>` as above:

```
sage: F7 = matroids.catalog.Fano()
sage: len(F7.nonspanning_circuits())
7
```

```
>>> from sage.all import *
>>> F7 = matroids.catalog.Fano()
>>> len(F7.nonspanning_circuits())
7
```

1.1.3 Constructing matroids

To define your own matroid, use the function *Matroid()*. This function attempts to interpret its arguments to create an appropriate matroid. The input arguments are documented in detail *below*.

EXAMPLES:

```
sage: A = Matrix(GF(2), [[1, 0, 0, 0, 1, 1, 1],
....:                    [0, 1, 0, 1, 0, 1, 1],
....:                    [0, 0, 1, 1, 1, 0, 1]])
sage: M = Matroid(A)
sage: M.is_isomorphic(matroids.catalog.Fano())
True

sage: M = Matroid(graphs.PetersenGraph())
↪# needs sage.graphs
sage: M.rank()
↪# needs sage.graphs
9
```

```
>>> from sage.all import *
>>> A = Matrix(GF(Integer(2)), [[Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)],
...                    [Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)],
...                    [Integer(0), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(0), Integer(1)]])
>>> M = Matroid(A)
>>> M.is_isomorphic(matroids.catalog.Fano())
True
```

(continues on next page)

(continued from previous page)

```
>>> M = Matroid(graphs.PetersenGraph())                                    #␣
↪needs sage.graphs
>>> M.rank()                                                               #␣
↪needs sage.graphs
9
```

AUTHORS:

- Rudi Pendavingh, Michael Welsh, Stefan van Zwam (2013-04-01): initial version

1.1.4 Functions

sage.matroids.constructor.**Matroid**(*groundset=None*, *data=None*, ***kwds*)

Construct a matroid.

Matroids are combinatorial structures that capture the abstract properties of (linear/algebraic/…) dependence. Formally, a matroid is a pair $M = (E, I)$ of a finite set E, the *groundset*, and a collection of subsets I, the independent sets, subject to the following axioms:

- I contains the empty set
- If X is a set in I, then each subset of X is in I
- If two subsets X, Y are in I, and $|X| > |Y|$, then there exists $x \in X - Y$ such that $Y + \{x\}$ is in I.

See the Wikipedia article on matroids for more theory and examples. Matroids can be obtained from many types of mathematical structures, and Sage supports a number of them.

There are two main entry points to Sage's matroid functionality. For built-in matroids, do the following:

- Within a Sage session, type "matroids." (Do not press `Enter`, and do not forget the final period ".")
- Hit `Tab`.

You will see a list of methods which will construct matroids. For example:

```
sage: F7 = matroids.catalog.Fano()
sage: len(F7.nonspanning_circuits())
7
```

```
>>> from sage.all import *
>>> F7 = matroids.catalog.Fano()
>>> len(F7.nonspanning_circuits())
7
```

or:

```
sage: U36 = matroids.Uniform(3, 6)
sage: U36.equals(U36.dual())
True
```

```
>>> from sage.all import *
>>> U36 = matroids.Uniform(Integer(3), Integer(6))
>>> U36.equals(U36.dual())
True
```

To define your own matroid, use the function `Matroid()`. This function attempts to interpret its arguments to create an appropriate matroid. The following named arguments are supported:

INPUT:

- `groundset` – (optional) the groundset of the matroid; if not provided, the function attempts to determine a groundset from the data

Exactly one of the following inputs must be given (where `data` must be a positional argument and anything else must be a keyword argument):

- `data` – a graph or a matrix or a RevLex-Index string or a list of independent sets containing all bases or a matroid
- `bases` – the list of bases (maximal independent sets) of the matroid
- `independent_sets` – the list of independent sets of the matroid
- `circuits` – the list of circuits of the matroid
- `nonspanning_circuits` – the list of nonspanning circuits of the matroid
- `flats` – the dictionary, list, or lattice of flats of the matroid
- `graph` – a graph, whose edges form the elements of the matroid
- `matrix` – a matrix representation of the matroid
- `reduced_matrix` – a reduced representation of the matroid: if `reduced_matrix = A` then the matroid is represented by $[I \ \ A]$ where I is an appropriately sized identity matrix
- `morphism` – a morphism representation of the matroid
- `reduced_morphism` – a reduced morphism representation of the matroid
- `rank_function` – a function that computes the rank of each subset; can only be provided together with a groundset
- `circuit_closures` – either a list of tuples `(k, C)` with `C` the closure of a circuit, and `k` the rank of `C`, or a dictionary `D` with `D[k]` the set of closures of rank-`k` circuits
- `revlex` – the encoding as a string of `0` and `*` symbols; used by [Mat2012] and explained in [MMIB2012]
- `matroid` – an object that is already a matroid; useful only with the `regular` option

Further options:

- `regular` – boolean (default: `False`); if `True`, output a *RegularMatroid* instance such that, *if* the input defines a valid regular matroid, then the output represents this matroid. Note that this option can be combined with any type of input.
- `ring` – any ring. If provided, and the input is a `matrix` or `reduced_matrix`, output will be a linear matroid over the ring or field `ring`.
- `field` – any field. Same as `ring`, but only fields are allowed
- `check` – boolean (default: `True`); if `True` and `regular` is `True`, the output is checked to make sure it is a valid regular matroid

Warning

Except for regular matroids, the input is not checked for validity. If your data does not correspond to an actual matroid, the behavior of the methods is undefined and may cause strange errors. To ensure you have a matroid, run *M.is_valid()*.

Note

The `Matroid()` method will return instances of type *BasisMatroid*, *CircuitsMatroid*, *FlatsMatroid*, *CircuitClosuresMatroid*, *LinearMatroid*, *BinaryMatroid*, *TernaryMatroid*, *QuaternaryMatroid*, *RegularMatroid*, or *RankMatroid*. To import these classes (and other useful functions) directly into Sage's main namespace, type:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See *sage.matroids.advanced*.

EXAMPLES:

Note that in these examples we will often use the fact that strings are iterable in these examples. So we type `'abcd'` to denote the list `['a', 'b', 'c', 'd']`.

1. List of bases:

   All of the following inputs are allowed, and equivalent:

```
sage: M1 = Matroid(groundset='abcd', bases=['ab', 'ac', 'ad',
....:                                       'bc', 'bd', 'cd'])
sage: M2 = Matroid(bases=['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
sage: M3 = Matroid(['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
sage: M4 = Matroid('abcd', ['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
sage: M5 = Matroid('abcd', bases=[['a', 'b'], ['a', 'c'],
....:                             ['a', 'd'], ['b', 'c'],
....:                             ['b', 'd'], ['c', 'd']])
sage: M1 == M2
True
sage: M1 == M3
True
sage: M1 == M4
True
sage: M1 == M5
True
```

```
>>> from sage.all import *
>>> M1 = Matroid(groundset='abcd', bases=['ab', 'ac', 'ad',
...                                       'bc', 'bd', 'cd'])
>>> M2 = Matroid(bases=['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
>>> M3 = Matroid(['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
>>> M4 = Matroid('abcd', ['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
>>> M5 = Matroid('abcd', bases=[['a', 'b'], ['a', 'c'],
...                             ['a', 'd'], ['b', 'c'],
...                             ['b', 'd'], ['c', 'd']])
>>> M1 == M2
True
>>> M1 == M3
True
>>> M1 == M4
True
```

(continues on next page)

(continued from previous page)

```
>>> M1 == M5
True
```

We do not check if the provided input forms an actual matroid:

```
sage: M1 = Matroid(groundset='abcd', bases=['ab', 'cd'])
sage: M1.full_rank()
2
sage: M1.is_valid()
False
```

```
>>> from sage.all import *
>>> M1 = Matroid(groundset='abcd', bases=['ab', 'cd'])
>>> M1.full_rank()
2
>>> M1.is_valid()
False
```

Bases may be repeated:

```
sage: M1 = Matroid(['ab', 'ac'])
sage: M2 = Matroid(['ab', 'ac', 'ab'])
sage: M1 == M2
True
```

```
>>> from sage.all import *
>>> M1 = Matroid(['ab', 'ac'])
>>> M2 = Matroid(['ab', 'ac', 'ab'])
>>> M1 == M2
True
```

2. List of independent sets:

```
sage: M1 = Matroid(groundset='abcd',
....:              independent_sets=['', 'a', 'b', 'c', 'd', 'ab',
....:                                'ac', 'ad', 'bc', 'bd', 'cd'])
```

```
>>> from sage.all import *
>>> M1 = Matroid(groundset='abcd',
...              independent_sets=['', 'a', 'b', 'c', 'd', 'ab',
...                                'ac', 'ad', 'bc', 'bd', 'cd'])
```

We only require that the list of independent sets contains each basis of the matroid; omissions of smaller independent sets and repetitions are allowed:

```
sage: M1 = Matroid(bases=['ab', 'ac'])
sage: M2 = Matroid(independent_sets=['a', 'ab', 'b', 'ab', 'a',
....:                                'b', 'ac'])
sage: M1 == M2
True
```

```
>>> from sage.all import *
>>> M1 = Matroid(bases=['ab', 'ac'])
>>> M2 = Matroid(independent_sets=['a', 'ab', 'b', 'ab', 'a',
...                                'b', 'ac'])
>>> M1 == M2
True
```

3. List of circuits:

```
sage: M1 = Matroid(groundset='abc', circuits=['bc'])
```

```
>>> from sage.all import *
>>> M1 = Matroid(groundset='abc', circuits=['bc'])
```

A matroid specified by a list of circuits gets converted to a *CircuitsMatroid* internally:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M2 = CircuitsMatroid(Matroid(bases=['ab', 'ac']))
sage: M1 == M2
True

sage: M = Matroid(groundset='abcd', circuits=['abc', 'abd', 'acd',
....:                                         'bcd'])
sage: type(M)
<class 'sage.matroids.circuits_matroid.CircuitsMatroid'>
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M2 = CircuitsMatroid(Matroid(bases=['ab', 'ac']))
>>> M1 == M2
True

>>> M = Matroid(groundset='abcd', circuits=['abc', 'abd', 'acd',
...                                         'bcd'])
>>> type(M)
<class 'sage.matroids.circuits_matroid.CircuitsMatroid'>
```

Strange things can happen if the input does not satisfy the circuit axioms, and these can be caught by the *is_valid()* method. So please check that your input makes sense!

```
sage: M = Matroid('abcd', circuits=['ab', 'acd'])
sage: M.is_valid()
False
```

```
>>> from sage.all import *
>>> M = Matroid('abcd', circuits=['ab', 'acd'])
>>> M.is_valid()
False
```

4. Flats:

Given a dictionary of flats indexed by their rank, we get a *FlatsMatroid*:

```
sage: M = Matroid(flats={0: [''], 1: ['a', 'b'], 2: ['ab']})
sage: M.is_isomorphic(matroids.Uniform(2, 2)) and M.is_valid()
True
sage: type(M)
<class 'sage.matroids.flats_matroid.FlatsMatroid'>
```

```
>>> from sage.all import *
>>> M = Matroid(flats={Integer(0): [''], Integer(1): ['a', 'b'], Integer(2): [
↪'ab']})
>>> M.is_isomorphic(matroids.Uniform(Integer(2), Integer(2))) and M.is_valid()
True
>>> type(M)
<class 'sage.matroids.flats_matroid.FlatsMatroid'>
```

If instead we simply provide a list of flats, then the class computes and stores the lattice of flats upon definition. This can be time-consuming, but after it's done we benefit from some faster methods (e.g., *is_valid()*):

```
sage: M = Matroid(flats=['', 'a', 'b', 'ab'])
sage: for i in range(M.rank() + 1):  # print flats by rank
....:     print(f'{i}: {sorted([sorted(F) for F in M.flats(i)], key=str)}')
0: [[]]
1: [['a'], ['b']]
2: [['a', 'b']]
sage: M.is_valid()
True
sage: type(M)
<class 'sage.matroids.flats_matroid.FlatsMatroid'>
```

```
>>> from sage.all import *
>>> M = Matroid(flats=['', 'a', 'b', 'ab'])
>>> for i in range(M.rank() + Integer(1)):  # print flats by rank
...     print(f'{i}: {sorted([sorted(F) for F in M.flats(i)], key=str)}')
0: [[]]
1: [['a'], ['b']]
2: [['a', 'b']]
>>> M.is_valid()
True
>>> type(M)
<class 'sage.matroids.flats_matroid.FlatsMatroid'>
```

Finally, we can also directly provide a lattice of flats:

```
sage: from sage.combinat.posets.lattices import LatticePoset
sage: flats = [frozenset(F) for F in powerset('ab')]
sage: L_M = LatticePoset((flats, lambda x, y: x < y))
sage: M = Matroid(L_M)
sage: M.is_isomorphic(matroids.Uniform(2, 2)) and M.is_valid()
True
sage: type(M)
<class 'sage.matroids.flats_matroid.FlatsMatroid'>
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> from sage.combinat.posets.lattices import LatticePoset
>>> flats = [frozenset(F) for F in powerset('ab')]
>>> L_M = LatticePoset((flats, lambda x, y: x < y))
>>> M = Matroid(L_M)
>>> M.is_isomorphic(matroids.Uniform(Integer(2), Integer(2))) and M.is_valid()
True
>>> type(M)
<class 'sage.matroids.flats_matroid.FlatsMatroid'>
```

5. Graph:

   Sage has great support for graphs, see `sage.graphs.graph`.

```
sage: G = graphs.PetersenGraph()                                              #␣
↪needs sage.graphs
sage: Matroid(G)                                                              #␣
↪needs sage.graphs
Graphic matroid of rank 9 on 15 elements
```

```
>>> from sage.all import *
>>> G = graphs.PetersenGraph()                                              #␣
↪needs sage.graphs
>>> Matroid(G)                                                              #␣
↪needs sage.graphs
Graphic matroid of rank 9 on 15 elements
```

   If each edge has a unique label, then those are used as the ground set labels:

```
sage: G = Graph([(0, 1, 'a'), (0, 2, 'b'), (1, 2, 'c')])                      #␣
↪needs sage.graphs
sage: M = Matroid(G)                                                          #␣
↪needs sage.graphs
sage: sorted(M.groundset())                                                   #␣
↪needs sage.graphs
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> G = Graph([(Integer(0), Integer(1), 'a'), (Integer(0), Integer(2), 'b'),␣
↪(Integer(1), Integer(2), 'c')])                    # needs sage.graphs
>>> M = Matroid(G)                                                          #␣
↪needs sage.graphs
>>> sorted(M.groundset())                                                   #␣
↪needs sage.graphs
['a', 'b', 'c']
```

   If there are parallel edges, then integers are used for the ground set. If there are no edges in parallel, and is not a complete list of labels, or the labels are not unique, then vertex tuples are used:

```
sage: # needs sage.graphs
sage: G = Graph([(0, 1, 'a'), (0, 2, 'b'), (1, 2, 'b')])
sage: M = Matroid(G)
sage: sorted(M.groundset())
[(0, 1), (0, 2), (1, 2)]
```

(continues on next page)

(continued from previous page)

```
sage: H = Graph([(0, 1, 'a'), (0, 2, 'b'), (1, 2, 'b'), (1, 2, 'c')],
....:           multiedges=True)
sage: N = Matroid(H)
sage: sorted(N.groundset())
[0, 1, 2, 3]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> G = Graph([(Integer(0), Integer(1), 'a'), (Integer(0), Integer(2), 'b'),␣
↪(Integer(1), Integer(2), 'b')])
>>> M = Matroid(G)
>>> sorted(M.groundset())
[(0, 1), (0, 2), (1, 2)]
>>> H = Graph([(Integer(0), Integer(1), 'a'), (Integer(0), Integer(2), 'b'),␣
↪(Integer(1), Integer(2), 'b'), (Integer(1), Integer(2), 'c')],
...           multiedges=True)
>>> N = Matroid(H)
>>> sorted(N.groundset())
[0, 1, 2, 3]
```

The GraphicMatroid object forces its graph to be connected. If a disconnected graph is used as input, it will connect the components:

```
sage: # needs sage.graphs
sage: G1 = graphs.CycleGraph(3); G2 = graphs.DiamondGraph()
sage: G = G1.disjoint_union(G2)
sage: M = Matroid(G); M
Graphic matroid of rank 5 on 8 elements
sage: M.graph()
Looped multi-graph on 6 vertices
sage: M.graph().is_connected()
True
sage: M.is_connected()
False
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> G1 = graphs.CycleGraph(Integer(3)); G2 = graphs.DiamondGraph()
>>> G = G1.disjoint_union(G2)
>>> M = Matroid(G); M
Graphic matroid of rank 5 on 8 elements
>>> M.graph()
Looped multi-graph on 6 vertices
>>> M.graph().is_connected()
True
>>> M.is_connected()
False
```

If the keyword `regular` is set to `True`, the output will instead be an instance of *RegularMatroid*.

```
sage: G = Graph([(0, 1), (0, 2), (1, 2)])                                    #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
sage: M = Matroid(G, regular=True); M                                          #␣
↪needs sage.graphs
Regular matroid of rank 2 on 3 elements with 3 bases
```

```
>>> from sage.all import *
>>> G = Graph([(Integer(0), Integer(1)), (Integer(0), Integer(2)),␣
↪(Integer(1), Integer(2))])                                    # needs sage.
↪graphs
>>> M = Matroid(G, regular=True); M                                            #␣
↪needs sage.graphs
Regular matroid of rank 2 on 3 elements with 3 bases
```

Note: if a groundset is specified, we assume it is in the same order as `G.edge_iterator()` provides:

```
sage: G = Graph([(0, 1), (0, 2), (0, 2), (1, 2)], multiedges=True)             #␣
↪needs sage.graphs
sage: M = Matroid('abcd', G)                                                   #␣
↪needs sage.graphs
sage: M.rank(['b', 'c'])                                                       #␣
↪needs sage.graphs
1
```

```
>>> from sage.all import *
>>> G = Graph([(Integer(0), Integer(1)), (Integer(0), Integer(2)),␣
↪(Integer(0), Integer(2)), (Integer(1), Integer(2))], multiedges=True)       ␣
↪    # needs sage.graphs
>>> M = Matroid('abcd', G)                                                     #␣
↪needs sage.graphs
>>> M.rank(['b', 'c'])                                                         #␣
↪needs sage.graphs
1
```

As before, if no edge labels are present and the graph is simple, we use the tuples `(i, j)` of endpoints. If that fails, we simply use a list `[0..m-1]`

```
sage: G = Graph([(0, 1), (0, 2), (1, 2)])                                      #␣
↪needs sage.graphs
sage: M = Matroid(G, regular=True)                                             #␣
↪needs sage.graphs
sage: sorted(M.groundset())                                                    #␣
↪needs sage.graphs
[(0, 1), (0, 2), (1, 2)]

sage: G = Graph([(0, 1), (0, 2), (0, 2), (1, 2)], multiedges=True)             #␣
↪needs sage.graphs
sage: M = Matroid(G, regular=True)                                             #␣
↪needs sage.graphs
sage: sorted(M.groundset())                                                    #␣
↪needs sage.graphs
[0, 1, 2, 3]
```

```
>>> from sage.all import *
>>> G = Graph([(Integer(0), Integer(1)), (Integer(0), Integer(2)),␣
↪(Integer(1), Integer(2))])                                    # needs sage.
↪graphs
>>> M = Matroid(G, regular=True)                                          #␣
↪needs sage.graphs
>>> sorted(M.groundset())                                                 #␣
↪needs sage.graphs
[(0, 1), (0, 2), (1, 2)]

>>> G = Graph([(Integer(0), Integer(1)), (Integer(0), Integer(2)),␣
↪(Integer(0), Integer(2)), (Integer(1), Integer(2))], multiedges=True)      ␣
↪   # needs sage.graphs
>>> M = Matroid(G, regular=True)                                          #␣
↪needs sage.graphs
>>> sorted(M.groundset())                                                 #␣
↪needs sage.graphs
[0, 1, 2, 3]
```

When the `graph` keyword is used, a variety of inputs can be converted to a graph automatically. The following uses a graph6 string (see the `Graph` method's documentation):

```
sage: Matroid(graph=':I`AKGsaOs`cI]Gb~')                                   #␣
↪needs sage.graphs
Graphic matroid of rank 9 on 17 elements
```

```
>>> from sage.all import *
>>> Matroid(graph=':I`AKGsaOs`cI]Gb~')                                    #␣
↪needs sage.graphs
Graphic matroid of rank 9 on 17 elements
```

However, this method is no more clever than `Graph()`:

```
sage: Matroid(graph=41/2)                                                  #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: This input cannot be turned into a graph
```

```
>>> from sage.all import *
>>> Matroid(graph=Integer(41)/Integer(2))                                  ␣
↪              # needs sage.graphs
Traceback (most recent call last):
...
ValueError: This input cannot be turned into a graph
```

6. Matrix:

   The basic input is a `Sage matrix`:

```
sage: A = Matrix(GF(2), [[1, 0, 0, 1, 1, 0],
....:                    [0, 1, 0, 1, 0, 1],
....:                    [0, 0, 1, 0, 1, 1]])
```

(continues on next page)

(continued from previous page)

```
sage: M = Matroid(matrix=A)
sage: M.is_isomorphic(matroids.CompleteGraphic(4))                              #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> A = Matrix(GF(Integer(2)), [[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(0)],
...                     [Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1)],
...                     [Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)]])
>>> M = Matroid(matrix=A)
>>> M.is_isomorphic(matroids.CompleteGraphic(Integer(4)))                       ␣
↪     # needs sage.graphs
True
```

Various shortcuts are possible:

```
sage: M1 = Matroid(matrix=[[1, 0, 0, 1, 1, 0],
....:                      [0, 1, 0, 1, 0, 1],
....:                      [0, 0, 1, 0, 1, 1]], ring=GF(2))
sage: M2 = Matroid(reduced_matrix=[[1, 1, 0],
....:                              [1, 0, 1],
....:                              [0, 1, 1]], ring=GF(2))
sage: M3 = Matroid(groundset=[0, 1, 2, 3, 4, 5],
....:              matrix=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
....:              ring=GF(2))
sage: A = Matrix(GF(2), [[1, 1, 0], [1, 0, 1], [0, 1, 1]])
sage: M4 = Matroid([0, 1, 2, 3, 4, 5], A)
sage: M1 == M2
True
sage: M1 == M3
True
sage: M1 == M4
True
```

```
>>> from sage.all import *
>>> M1 = Matroid(matrix=[[Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0)],
...                     [Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1)],
...                     [Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)]], ring=GF(Integer(2)))
>>> M2 = Matroid(reduced_matrix=[[Integer(1), Integer(1), Integer(0)],
...                              [Integer(1), Integer(0), Integer(1)],
...                              [Integer(0), Integer(1), Integer(1)]],␣
↪ring=GF(Integer(2)))
>>> M3 = Matroid(groundset=[Integer(0), Integer(1), Integer(2), Integer(3),␣
↪Integer(4), Integer(5)],
...              matrix=[[Integer(1), Integer(1), Integer(0)], [Integer(1),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)]],
```

(continues on next page)

(continued from previous page)

```
...                ring=GF(Integer(2)))
>>> A = Matrix(GF(Integer(2)), [[Integer(1), Integer(1), Integer(0)],␣
↪[Integer(1), Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)]])
>>> M4 = Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)], A)
>>> M1 == M2
True
>>> M1 == M3
True
>>> M1 == M4
True
```

However, with unnamed arguments the input has to be a `Matrix` instance, or the function will try to interpret it as a set of bases:

```
sage: Matroid([0, 1, 2], [[1, 0, 1], [0, 1, 1]])
Traceback (most recent call last):
...
ValueError: basis has wrong cardinality
```

```
>>> from sage.all import *
>>> Matroid([Integer(0), Integer(1), Integer(2)], [[Integer(1), Integer(0),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1)]])
Traceback (most recent call last):
...
ValueError: basis has wrong cardinality
```

If the groundset size equals number of rows plus number of columns, an identity matrix is prepended. Otherwise the groundset size must equal the number of columns:

```
sage: A = Matrix(GF(2), [[1, 1, 0], [1, 0, 1], [0, 1, 1]])
sage: M = Matroid([0, 1, 2], A)
sage: N = Matroid([0, 1, 2, 3, 4, 5], A)
sage: M.rank()
2
sage: N.rank()
3
```

```
>>> from sage.all import *
>>> A = Matrix(GF(Integer(2)), [[Integer(1), Integer(1), Integer(0)],␣
↪[Integer(1), Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)]])
>>> M = Matroid([Integer(0), Integer(1), Integer(2)], A)
>>> N = Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)], A)
>>> M.rank()
2
>>> N.rank()
3
```

We automatically create an optimized subclass, if available:

```
sage: Matroid([0, 1, 2, 3, 4, 5],
....:         matrix=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
....:         field=GF(2))
Binary matroid of rank 3 on 6 elements, type (2, 7)
sage: Matroid([0, 1, 2, 3, 4, 5],
....:         matrix=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
....:         field=GF(3))
Ternary matroid of rank 3 on 6 elements, type 0-
sage: Matroid([0, 1, 2, 3, 4, 5],                                    #␣
↪needs sage.rings.finite_rings
....:         matrix=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
....:         field=GF(4, 'x'))
Quaternary matroid of rank 3 on 6 elements
sage: Matroid([0, 1, 2, 3, 4, 5],                                    #␣
↪needs sage.graphs
....:         matrix=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
....:         field=GF(2), regular=True)
Regular matroid of rank 3 on 6 elements with 16 bases
```

```
>>> from sage.all import *
>>> Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)],
...         matrix=[[Integer(1), Integer(1), Integer(0)], [Integer(1),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)]],
...         field=GF(Integer(2)))
Binary matroid of rank 3 on 6 elements, type (2, 7)
>>> Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)],
...         matrix=[[Integer(1), Integer(1), Integer(0)], [Integer(1),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)]],
...         field=GF(Integer(3)))
Ternary matroid of rank 3 on 6 elements, type 0-
>>> Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)],                                          # needs sage.rings.
↪finite_rings
...         matrix=[[Integer(1), Integer(1), Integer(0)], [Integer(1),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)]],
...         field=GF(Integer(4), 'x'))
Quaternary matroid of rank 3 on 6 elements
>>> Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)],                                          # needs sage.graphs
...         matrix=[[Integer(1), Integer(1), Integer(0)], [Integer(1),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)]],
...         field=GF(Integer(2)), regular=True)
Regular matroid of rank 3 on 6 elements with 16 bases
```

Otherwise the generic LinearMatroid class is used:

```
sage: Matroid([0, 1, 2, 3, 4, 5],
....:         matrix=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
....:         field=GF(83))
Linear matroid of rank 3 on 6 elements represented over the Finite
Field of size 83
```

```
>>> from sage.all import *
>>> Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)],
...         matrix=[[Integer(1), Integer(1), Integer(0)], [Integer(1),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(1)]],
...         field=GF(Integer(83)))
Linear matroid of rank 3 on 6 elements represented over the Finite
Field of size 83
```

An integer matrix is automatically converted to a matrix over **Q**. If you really want integers, you can specify the ring explicitly:

```
sage: A = Matrix([[1, 1, 0], [1, 0, 1], [0, 1, -1]])
sage: A.base_ring()
Integer Ring
sage: M = Matroid([0, 1, 2, 3, 4, 5], A)
sage: M.base_ring()
Rational Field
sage: M = Matroid([0, 1, 2, 3, 4, 5], A, ring=ZZ)
sage: M.base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> A = Matrix([[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0),
↪ Integer(1)], [Integer(0), Integer(1), -Integer(1)]])
>>> A.base_ring()
Integer Ring
>>> M = Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)], A)
>>> M.base_ring()
Rational Field
>>> M = Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)], A, ring=ZZ)
>>> M.base_ring()
Integer Ring
```

A morphism representation of a *LinearMatroid* can also be used as input:

```
sage: M = matroids.catalog.Fano()
sage: A = M.representation(order=True); A
Generic morphism:
  From: Free module generated by {'a', 'b', 'c', 'd', 'e', 'f', 'g'} over
        Finite Field of size 2
  To:   Free module generated by {0, 1, 2} over Finite Field of size 2
sage: A._unicode_art_matrix()
  a b c d e f g
0⎛1 0 0 0 1 1 1⎞
1⎜0 1 0 1 0 1 1⎟
2⎝0 0 1 1 1 0 1⎠
sage: N = Matroid(A); N
Binary matroid of rank 3 on 7 elements, type (3, 0)
sage: N.groundset()
frozenset({'a', 'b', 'c', 'd', 'e', 'f', 'g'})
```

(continues on next page)

(continued from previous page)

```
sage: M == N
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> A = M.representation(order=True); A
Generic morphism:
  From: Free module generated by {'a', 'b', 'c', 'd', 'e', 'f', 'g'} over
        Finite Field of size 2
  To:   Free module generated by {0, 1, 2} over Finite Field of size 2
>>> A._unicode_art_matrix()
  a b c d e f g
0⎛1 0 0 0 1 1 1⎞
1⎜0 1 0 1 0 1 1⎟
2⎝0 0 1 1 1 0 1⎠
>>> N = Matroid(A); N
Binary matroid of rank 3 on 7 elements, type (3, 0)
>>> N.groundset()
frozenset({'a', 'b', 'c', 'd', 'e', 'f', 'g'})
>>> M == N
True
```

The keywords `morphism` and `reduced_morphism` are also available:

```
sage: M = matroids.catalog.RelaxedNonFano("abcdefg")
sage: A = M.representation(order=True, reduced=True); A
Generic morphism:
  From: Free module generated by {'d', 'e', 'f', 'g'} over
        Finite Field in w of size 2^2
  To:   Free module generated by {'a', 'b', 'c'} over
        Finite Field in w of size 2^2
sage: A._unicode_art_matrix()
  d e f g
a⎛1 1 0 1⎞
b⎜1 0 1 1⎟
c⎝0 1 w 1⎠
sage: N = Matroid(reduced_morphism=A); N
Quaternary matroid of rank 3 on 7 elements
sage: N.groundset()
frozenset({'a', 'b', 'c', 'd', 'e', 'f', 'g'})
sage: M == N
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.RelaxedNonFano("abcdefg")
>>> A = M.representation(order=True, reduced=True); A
Generic morphism:
  From: Free module generated by {'d', 'e', 'f', 'g'} over
        Finite Field in w of size 2^2
  To:   Free module generated by {'a', 'b', 'c'} over
        Finite Field in w of size 2^2
>>> A._unicode_art_matrix()
```

(continues on next page)

(continued from previous page)

```
  d e f g
a⎛1 1 0 1⎞
b⎜1 0 1 1⎟
c⎝0 1 w 1⎠
>>> N = Matroid(reduced_morphism=A); N
Quaternary matroid of rank 3 on 7 elements
>>> N.groundset()
frozenset({'a', 'b', 'c', 'd', 'e', 'f', 'g'})
>>> M == N
True
```

7. Rank function:

   Any function mapping subsets to integers can be used as input:

```
sage: def f(X):
....:     return min(len(X), 2)
sage: M = Matroid('abcd', rank_function=f)
sage: M
Matroid of rank 2 on 4 elements
sage: M.is_isomorphic(matroids.Uniform(2, 4))
True
```

```
>>> from sage.all import *
>>> def f(X):
...     return min(len(X), Integer(2))
>>> M = Matroid('abcd', rank_function=f)
>>> M
Matroid of rank 2 on 4 elements
>>> M.is_isomorphic(matroids.Uniform(Integer(2), Integer(4)))
True
```

8. Circuit closures:

   This is often a really concise way to specify a matroid. The usual way is a dictionary of lists:

```
sage: M = Matroid(circuit_closures={3: ['edfg', 'acdg', 'bcfg',
....:     'cefh', 'afgh', 'abce', 'abdf', 'begh', 'bcdh', 'adeh'],
....:     4: ['abcdefgh']})
sage: M.equals(matroids.catalog.P8())
True
```

```
>>> from sage.all import *
>>> M = Matroid(circuit_closures={Integer(3): ['edfg', 'acdg', 'bcfg',
...     'cefh', 'afgh', 'abce', 'abdf', 'begh', 'bcdh', 'adeh'],
...     Integer(4): ['abcdefgh']})
>>> M.equals(matroids.catalog.P8())
True
```

   You can also input tuples (k, X) where X is the closure of a circuit, and k the rank of X:

```
sage: M = Matroid(circuit_closures=[(2, 'abd'), (3, 'abcdef'),
....:                               (2, 'bce')])
```

(continues on next page)

(continued from previous page)

```
sage: M.equals(matroids.catalog.Q6())                                          #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> M = Matroid(circuit_closures=[(Integer(2), 'abd'), (Integer(3), 'abcdef'),
...                               (Integer(2), 'bce')])
>>> M.equals(matroids.catalog.Q6())                                            #␣
↪needs sage.rings.finite_rings
True
```

9. RevLex-Index:

   This requires the `groundset` to be given and also needs a additional keyword argument `rank` to specify the rank of the matroid:

```
sage: M = Matroid("abcdef", "000000******0**", rank=4); M
Matroid of rank 4 on 6 elements with 8 bases
sage: list(M.bases())
[frozenset({'a', 'b', 'd', 'f'}),
 frozenset({'a', 'c', 'd', 'f'}),
 frozenset({'b', 'c', 'd', 'f'}),
 frozenset({'a', 'b', 'e', 'f'}),
 frozenset({'a', 'c', 'e', 'f'}),
 frozenset({'b', 'c', 'e', 'f'}),
 frozenset({'b', 'd', 'e', 'f'}),
 frozenset({'c', 'd', 'e', 'f'})]
```

```
>>> from sage.all import *
>>> M = Matroid("abcdef", "000000******0**", rank=Integer(4)); M
Matroid of rank 4 on 6 elements with 8 bases
>>> list(M.bases())
[frozenset({'a', 'b', 'd', 'f'}),
 frozenset({'a', 'c', 'd', 'f'}),
 frozenset({'b', 'c', 'd', 'f'}),
 frozenset({'a', 'b', 'e', 'f'}),
 frozenset({'a', 'c', 'e', 'f'}),
 frozenset({'b', 'c', 'e', 'f'}),
 frozenset({'b', 'd', 'e', 'f'}),
 frozenset({'c', 'd', 'e', 'f'})]
```

   Only the `0` symbols really matter, any symbol can be used instead of `*`:

   sage: Matroid("abcdefg", revlex='0++++++++0++++0+++++0+-++—-+-++', rank=4) Matroid of rank 4 on 7 elements with 31 bases

   It is checked that the input makes sense (but not that it defines a matroid):

```
sage: Matroid("abcdef", "000000******0**")
Traceback (most recent call last):
...
TypeError: for RevLex-Index, the rank needs to be specified
sage: Matroid("abcdef", "000000******0**", rank=3)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: expected string of length 20 (6 choose 3), got 15
sage: M = Matroid("abcdef", "*0000000000000*", rank=4); M
Matroid of rank 4 on 6 elements with 2 bases
sage: M.is_valid()
False
```

```
>>> from sage.all import *
>>> Matroid("abcdef", "000000******0**")
Traceback (most recent call last):
...
TypeError: for RevLex-Index, the rank needs to be specified
>>> Matroid("abcdef", "000000******0**", rank=Integer(3))
Traceback (most recent call last):
...
ValueError: expected string of length 20 (6 choose 3), got 15
>>> M = Matroid("abcdef", "*0000000000000*", rank=Integer(4)); M
Matroid of rank 4 on 6 elements with 2 bases
>>> M.is_valid()
False
```

10. Matroid:

    Most of the time, the matroid itself is returned:

```
sage: M = matroids.catalog.Fano()
sage: N = Matroid(M)
sage: N is M
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> N = Matroid(M)
>>> N is M
True
```

    But it can be useful with the `regular` option:

```
sage: M = Matroid(circuit_closures={2:['adb', 'bec', 'cfa',
....:                                  'def'], 3:['abcdef']})
sage: N = Matroid(M, regular=True); N                                          #␣
↪needs sage.graphs
Regular matroid of rank 3 on 6 elements with 16 bases
sage: M == N                                                                   #␣
↪needs sage.graphs
False
sage: M.is_isomorphic(N)                                                       #␣
↪needs sage.graphs
True
sage: Matrix(N)  # random                                                      #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
[1 0 0 1 1 0]
[0 1 0 1 1 1]
[0 0 1 0 1 1]
```

```
>>> from sage.all import *
>>> M = Matroid(circuit_closures={Integer(2):['adb', 'bec', 'cfa',
...                                           'def'], Integer(3):['abcdef']})
>>> N = Matroid(M, regular=True); N                                          #␣
↪needs sage.graphs
Regular matroid of rank 3 on 6 elements with 16 bases
>>> M == N                                                                   #␣
↪needs sage.graphs
False
>>> M.is_isomorphic(N)                                                       #␣
↪needs sage.graphs
True
>>> Matrix(N)  # random                                                     #␣
↪needs sage.graphs
[1 0 0 1 1 0]
[0 1 0 1 1 1]
[0 0 1 0 1 1]
```

The `regular` option:

```
sage: M = Matroid(reduced_matrix=[[1, 1, 0],                                #␣
↪needs sage.graphs
....:                             [1, 0, 1],
....:                             [0, 1, 1]], regular=True); M
Regular matroid of rank 3 on 6 elements with 16 bases

sage: M.is_isomorphic(matroids.CompleteGraphic(4))                          #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> M = Matroid(reduced_matrix=[[Integer(1), Integer(1), Integer(0)],       ␣
↪                      # needs sage.graphs
...                            [Integer(1), Integer(0), Integer(1)],
...                            [Integer(0), Integer(1), Integer(1)]],␣
↪regular=True); M
Regular matroid of rank 3 on 6 elements with 16 bases

>>> M.is_isomorphic(matroids.CompleteGraphic(Integer(4)))                   ␣
↪     # needs sage.graphs
True
```

By default we check if the resulting matroid is actually regular. To increase speed, this check can be skipped:

```
sage: M = matroids.catalog.Fano()
sage: N = Matroid(M, regular=True)                                          #␣
↪needs sage.graphs
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: input is not a valid regular matroid
sage: N = Matroid(M, regular=True, check=False); N                              #␣
↪needs sage.graphs
Regular matroid of rank 3 on 7 elements with 32 bases

sage: N.is_valid()                                                              #␣
↪needs sage.graphs
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> N = Matroid(M, regular=True)                                                #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: input is not a valid regular matroid
>>> N = Matroid(M, regular=True, check=False); N                                #␣
↪needs sage.graphs
Regular matroid of rank 3 on 7 elements with 32 bases

>>> N.is_valid()                                                               #␣
↪needs sage.graphs
False
```

Sometimes the output is regular, but represents a different matroid from the one you intended:

```
sage: M = Matroid(Matrix(GF(3), [[1, 0, 1, 1], [0, 1, 1, 2]]))
sage: N = Matroid(Matrix(GF(3), [[1, 0, 1, 1], [0, 1, 1, 2]]),                  #␣
↪needs sage.graphs
....:             regular=True)
sage: N.is_valid()                                                              #␣
↪needs sage.graphs
True
sage: N.is_isomorphic(M)                                                        #␣
↪needs sage.graphs
False
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(3)), [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2)]]))
>>> N = Matroid(Matrix(GF(Integer(3)), [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2)]]),              ␣
↪  # needs sage.graphs
...             regular=True)
>>> N.is_valid()                                                               #␣
↪needs sage.graphs
True
>>> N.is_isomorphic(M)                                                         #␣
↪needs sage.graphs
False
```

1.2 The abstract Matroid class

Matroids are combinatorial structures that capture the abstract properties of (linear/algebraic/…) dependence. See the Wikipedia article on matroids for theory and examples. In Sage, various types of matroids are supported: *BasisMatroid*, *CircuitClosuresMatroid*, *LinearMatroid* (and some specialized subclasses), *RankMatroid*. To construct them, use the function *Matroid()*.

All these classes share a common interface, which includes the following methods (organized by category). Note that most subclasses (notably *LinearMatroids*) will implement additional functionality (e.g. linear extensions).

- **Groundset:**
  - *groundset()*
  - *size()*
- **Rank, bases, circuits, closure**
  - *rank()*
  - *full_rank()*
  - *basis()*
  - *max_independent()*
  - *circuit()*
  - *fundamental_circuit()*
  - *closure()*
  - *augment()*
  - *corank()*
  - *full_corank()*
  - *cobasis()*
  - *max_coindependent()*
  - *cocircuit()*
  - *fundamental_cocircuit()*
  - *coclosure()*
  - *is_independent()*
  - *is_dependent()*
  - *is_basis()*
  - *is_circuit()*
  - *is_closed()*
  - *is_coindependent()*
  - *is_codependent()*
  - *is_cobasis()*
  - *is_cocircuit()*
  - *is_coclosed()*
- **Verification**

  - *is_valid()*

- **Enumeration**
  - *circuits()*
  - *nonspanning_circuits()*
  - *cocircuits()*
  - *noncospanning_cocircuits()*
  - *circuit_closures()*
  - *nonspanning_circuit_closures()*
  - *bases()*
  - *independent_sets()*
  - *nonbases()*
  - *dependent_sets()*
  - *flats()*
  - *coflats()*
  - *hyperplanes()*
  - *f_vector()*
  - *whitney_numbers()*
  - *whitney_numbers2()*
  - *broken_circuits()*
  - *no_broken_circuits_sets()*
- **Comparison**
  - *is_isomorphic()*
  - *equals()*
  - *is_isomorphism()*
- **Minors, duality, truncation**
  - *minor()*
  - *contract()*
  - *delete()*
  - *dual()*
  - *truncation()*
  - *has_minor()*
  - *has_line_minor()*
- **Extension**
  - *extension()*
  - *coextension()*
  - *modular_cut()*

  - *linear_subclasses()*
  - *extensions()*
  - *coextensions()*

- **Connectivity, simplicity**
  - *loops()*
  - *coloops()*
  - *simplify()*
  - *cosimplify()*
  - *is_simple()*
  - *is_cosimple()*
  - *components()*
  - *is_connected()*
  - *is_3connected()*
  - *is_4connected()*
  - *is_kconnected()*
  - *connectivity()*
  - *is_paving()*
  - *is_sparse_paving()*
  - *girth()*

- **Representation**
  - *is_graphic()*
  - *is_regular()*
  - *binary_matroid()*
  - *is_binary()*
  - *ternary_matroid()*
  - *is_ternary()*
  - *relabel()*

- **Optimization**
  - *max_weight_independent()*
  - *max_weight_coindependent()*
  - *is_max_weight_independent_generic()*
  - *intersection()*
  - *intersection_unweighted()*

- **Invariants**
  - *tutte_polynomial()*
  - *characteristic_polynomial()*

  - *flat_cover()*

- **Visualization**

  - *show()*

  - *plot()*

- **Construction**

  - *union()*

  - *direct_sum()*

- **Misc**

  - *automorphism_group()*

  - *broken_circuit_complex()*

  - *chow_ring()*

  - *matroid_polytope()*

  - *independence_matroid_polytope()*

  - *lattice_of_flats()*

  - *orlik_solomon_algebra()*

  - *bergman_complex()*

  - *augmented_bergman_complex()*

In addition to these, all methods provided by `SageObject` are available, notably `save()` and `rename()`.

1.2.1 Advanced usage

Many methods (such as `M.rank()`) have a companion method whose name starts with an underscore (such as `M._rank()`). The method with the underscore does not do any checks on its input. For instance, it may assume of its input that

- It is a subset of the groundset. The interface is compatible with Python's `frozenset` type.
- It is a list of things, supports iteration, and recursively these rules apply to its members.

Using the underscored version could improve the speed of code a little, but will generate more cryptic error messages when presented with wrong input. In some instances, no error might occur and a nonsensical answer returned.

A subclass should always override the underscored method, if available, and as a rule leave the regular method alone.

These underscored methods are not documented in the reference manual. To see them, within Sage you can create a matroid `M` and type `M._` followed by `Tab`. Then `M._rank?` followed by `Enter` will bring up the documentation string of the `_rank()` method.

1.2.2 Creating new Matroid subclasses

Many mathematical objects give rise to matroids, and not all are available through the provided code. For incidental use, the *RankMatroid* subclass may suffice. If you regularly use matroids based on a new data type, you can write a subclass of `Matroid`. You only need to override the `__init__`, `_rank()` and `groundset()` methods to get a fully working class.

EXAMPLES:

In a partition matroid, a subset is independent if it has at most one element from each partition. The following is a very basic implementation, in which the partition is specified as a list of lists:

```
sage: import sage.matroids.matroid
sage: class PartitionMatroid(sage.matroids.matroid.Matroid):
....:     def __init__(self, partition):
....:         self.partition = partition
....:         E = set()
....:         for P in partition:
....:             E.update(P)
....:         self.E = frozenset(E)
....:     def groundset(self):
....:         return self.E
....:     def _rank(self, X):
....:         X2 = set(X)
....:         used_indices = set()
....:         r = 0
....:         while X2:
....:             e = X2.pop()
....:             for i in range(len(self.partition)):
....:                 if e in self.partition[i]:
....:                     if i not in used_indices:
....:                         used_indices.add(i)
....:                         r = r + 1
....:                     break
....:         return r
....:
sage: M = PartitionMatroid([[1, 2], [3, 4, 5], [6, 7]])
sage: M.full_rank()
3
sage: M.tutte_polynomial(var('x'), var('y'))                                #␣
↪needs sage.symbolic
x^2*y^2 + 2*x*y^3 + y^4 + x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

```
>>> from sage.all import *
>>> import sage.matroids.matroid
>>> class PartitionMatroid(sage.matroids.matroid.Matroid):
...     def __init__(self, partition):
...         self.partition = partition
...         E = set()
...         for P in partition:
...             E.update(P)
...         self.E = frozenset(E)
...     def groundset(self):
...         return self.E
...     def _rank(self, X):
...         X2 = set(X)
...         used_indices = set()
...         r = Integer(0)
...         while X2:
...             e = X2.pop()
...             for i in range(len(self.partition)):
...                 if e in self.partition[i]:
...                     if i not in used_indices:
...                         used_indices.add(i)
```

(continues on next page)

(continued from previous page)

```
...                         r = r + Integer(1)
...                     break
...         return r
....:
>>> M = PartitionMatroid([[Integer(1), Integer(2)], [Integer(3), Integer(4),␣
↪Integer(5)], [Integer(6), Integer(7)]])
>>> M.full_rank()
3
>>> M.tutte_polynomial(var('x'), var('y'))                                    #␣
↪needs sage.symbolic
x^2*y^2 + 2*x*y^3 + y^4 + x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

Note

The abstract base class has no idea about the data used to represent the matroid. Hence some methods need to be customized to function properly.

Necessary:

- `def __init__(self, ...)`
- `def groundset(self)`
- `def _rank(self, X)`

Representation:

- `def _repr_(self)`

Comparison:

- `def __hash__(self)`
- `def __eq__(self, other)`
- `def __ne__(self, other)`

In Cythonized classes, use `__richcmp__()` instead of `__eq__()`, `__ne__()`.

Copying, loading, saving:

- `def __copy__(self)`
- `def __deepcopy__(self, memo={})`
- `def __reduce__(self)`

See, for instance, *rank_matroid* or *circuit_closures_matroid* for sample implementations of these.

Note

The example provided does not check its input at all. You may want to make sure the input data are not corrupt.

1.2.3 Some examples

EXAMPLES:

Construction:

```
sage: M = Matroid(Matrix(QQ, [[1, 0, 0, 0, 1, 1, 1],
....:                         [0, 1, 0, 1, 0, 1, 1],
....:                         [0, 0, 1, 1, 1, 0, 1]]))
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
sage: M.rank([0, 1, 2])
3
sage: M.rank([0, 1, 5])
2
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(QQ, [[Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)],
...                         [Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(1)],
...                         [Integer(0), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(1)]]))
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
>>> M.rank([Integer(0), Integer(1), Integer(2)])
3
>>> M.rank([Integer(0), Integer(1), Integer(5)])
2
```

Minors:

```
sage: M = Matroid(Matrix(QQ, [[1, 0, 0, 0, 1, 1, 1],
....:                         [0, 1, 0, 1, 0, 1, 1],
....:                         [0, 0, 1, 1, 1, 0, 1]]))
sage: N = (M / [2]).delete([3, 4])
sage: sorted(N.groundset())
[0, 1, 5, 6]
sage: N.full_rank()
2
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(QQ, [[Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)],
...                         [Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(1)],
...                         [Integer(0), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(1)]]))
>>> N = (M / [Integer(2)]).delete([Integer(3), Integer(4)])
>>> sorted(N.groundset())
[0, 1, 5, 6]
>>> N.full_rank()
2
```

Testing. Note that the abstract base class does not support pickling:

```
sage: M = sage.matroids.matroid.Matroid()
sage: TestSuite(M).run(skip='_test_pickling')
```

```
>>> from sage.all import *
>>> M = sage.matroids.matroid.Matroid()
>>> TestSuite(M).run(skip='_test_pickling')
```

1.2.4 REFERENCES

- [BC1977]
- [Cun1986]
- [CMO2011]
- [CMO2012]
- [GG2012]
- [GR2001]
- [Hli2006]
- [Hoc]
- [Lyo2003]
- [Oxl1992]
- [Oxl2011]
- [Pen2012]
- [PvZ2010]
- [Raj1987]

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version
- Michael Welsh (2013-04-01): Added is_3connected(), using naive algorithm
- Michael Welsh (2013-04-03): Changed flats() to use SetSystem
- Giorgos Mousa (2024-02-15): Add Whitney numbers, characteristic polynomial

1.2.5 Methods

class sage.matroids.matroid.**Matroid**

Bases: SageObject

The abstract matroid class, from which all matroids are derived. Do not use this class directly!

To implement a subclass, the least you should do is implement the `__init__()`, `_rank()` and `groundset()` methods. See the source of *rank_matroid.py* for a bare-bones example of this.

EXAMPLES:

In a partition matroid, a subset is independent if it has at most one element from each partition. The following is a very basic implementation, in which the partition is specified as a list of lists:

```
sage: class PartitionMatroid(sage.matroids.matroid.Matroid):
....:     def __init__(self, partition):
....:         self.partition = partition
....:         E = set()
....:         for P in partition:
....:             E.update(P)
....:         self.E = frozenset(E)
....:     def groundset(self):
....:         return self.E
....:     def _rank(self, X):
....:         X2 = set(X)
....:         used_indices = set()
....:         r = 0
....:         while X2:
....:             e = X2.pop()
....:             for i in range(len(self.partition)):
....:                 if e in self.partition[i]:
....:                     if i not in used_indices:
....:                         used_indices.add(i)
....:                         r = r + 1
....:                     break
....:         return r
....:
sage: M = PartitionMatroid([[1, 2], [3, 4, 5], [6, 7]])
sage: M.full_rank()
3
sage: M.tutte_polynomial(var('x'), var('y'))                                    #␣
↪needs sage.symbolic
x^2*y^2 + 2*x*y^3 + y^4 + x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

```
>>> from sage.all import *
>>> class PartitionMatroid(sage.matroids.matroid.Matroid):
...     def __init__(self, partition):
...         self.partition = partition
...         E = set()
...         for P in partition:
...             E.update(P)
...         self.E = frozenset(E)
...     def groundset(self):
...         return self.E
...     def _rank(self, X):
...         X2 = set(X)
...         used_indices = set()
...         r = Integer(0)
...         while X2:
...             e = X2.pop()
...             for i in range(len(self.partition)):
...                 if e in self.partition[i]:
...                     if i not in used_indices:
...                         used_indices.add(i)
...                         r = r + Integer(1)
...                     break
```

(continues on next page)

(continued from previous page)

```
...         return r
....:
>>> M = PartitionMatroid([[Integer(1), Integer(2)], [Integer(3), Integer(4),␣
↪Integer(5)], [Integer(6), Integer(7)]])
>>> M.full_rank()
3
>>> M.tutte_polynomial(var('x'), var('y'))                                    #␣
↪needs sage.symbolic
x^2*y^2 + 2*x*y^3 + y^4 + x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

Note

The abstract base class has no idea about the data used to represent the matroid. Hence some methods need to be customized to function properly.

Necessary:

- `def __init__(self, ...)`
- `def groundset(self)`
- `def _rank(self, X)`

Representation:

- `def _repr_(self)`

Comparison:

- `def __hash__(self)`
- `def __eq__(self, other)`
- `def __ne__(self, other)`

In Cythonized classes, use `__richcmp__()` instead of `__eq__()`, `__ne__()`.

Copying, loading, saving:

- `def __copy__(self)`
- `def __deepcopy__(self, memo={})`
- `def __reduce__(self)`

See, for instance, *rank_matroid.py* or *circuit_closures_matroid.pyx* for sample implementations of these.

Note

Many methods (such as `M.rank()`) have a companion method whose name starts with an underscore (such as `M._rank()`). The method with the underscore does not do any checks on its input. For instance, it may assume of its input that

- Any input that should be a subset of the groundset, is one. The interface is compatible with Python's `frozenset` type.
- Any input that should be a list of things, supports iteration, and recursively these rules apply to its members.

Using the underscored version could improve the speed of code a little, but will generate more cryptic error messages when presented with wrong input. In some instances, no error might occur and a nonsensical answer returned.

A subclass should always override the underscored method, if available, and as a rule leave the regular method alone.

augment(*X*, *Y=None*)

Return a maximal subset I of $Y - X$ such that $r(X + I) = r(X) + r(I)$.

INPUT:

- `X` – a subset (or any iterable) of the groundset
- `Y` – (default: the groundset) a subset (or any iterable) of the groundset

OUTPUT: a subset of $Y - X$

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: X = set(['a']); Y = M.groundset()
sage: Z = M.augment(X, Y)
sage: M.is_independent(Z.union(X))
True
sage: W = Z.union(X)
sage: all(M.is_dependent(W.union([y])) for y in Y if y not in W)
True
sage: sorted(M.augment(['x']))
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
sage: sorted(M.augment(['a'], ['x']))
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> X = set(['a']); Y = M.groundset()
>>> Z = M.augment(X, Y)
>>> M.is_independent(Z.union(X))
True
>>> W = Z.union(X)
>>> all(M.is_dependent(W.union([y])) for y in Y if y not in W)
True
>>> sorted(M.augment(['x']))
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
>>> sorted(M.augment(['a'], ['x']))
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

augmented_bergman_complex()

Return the augmented Bergman complex of `self`.

Given a matroid M with groundset $E = \{1, 2, \ldots, n\}$, the *augmented Bergman complex* can be seen as a hybrid of the complex of independent sets of M and the Bergman complex of M. It is defined as the simplicial complex on vertex set

$$\{y_1, \ldots, y_n\} \cup \{x_F : \text{ proper flats } F \subsetneq E\},$$

with simplices given by

$$\{y_i\}_{i \in I} \cup \{x_{F_1}, \ldots, x_{F_\ell}\},$$

for which I is an independent set and $I \subseteq F_1 \subsetneq F_2 \subsetneq \cdots \subsetneq F_\ell$.

OUTPUT: a simplicial complex

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: A = M.augmented_bergman_complex(); A                                  #␣
↪needs sage.graphs
Simplicial complex with 22 vertices and 91 facets

sage: M = matroids.Uniform(2,3)
sage: A = M.augmented_bergman_complex(); A                                  #␣
↪needs sage.graphs
Simplicial complex with 7 vertices and 9 facets
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> A = M.augmented_bergman_complex(); A                                    #␣
↪needs sage.graphs
Simplicial complex with 22 vertices and 91 facets

>>> M = matroids.Uniform(Integer(2),Integer(3))
>>> A = M.augmented_bergman_complex(); A                                    #␣
↪needs sage.graphs
Simplicial complex with 7 vertices and 9 facets
```

Both the independent set complex of the matroid and the usual Bergman complex are subcomplexes of the augmented Bergman complex. The vertices of the complex are labeled by `L` when they belong to the independent set complex and `R` when they belong to the (cone of) the Bergman complex. The cone point is `'R[]'`:

```
sage: sorted(A.faces()[0])                                                  #␣
↪needs sage.graphs
[('L0',), ('L1',), ('L2',), ('R[0]',), ('R[1]',), ('R[2]',), ('R[]',)]
sage: sorted(map(sorted, A.faces()[1]))                                     #␣
↪needs sage.graphs
[['L0', 'L1'],
 ['L0', 'L2'],
 ['L0', 'R[0]'],
 ['L1', 'L2'],
 ['L1', 'R[1]'],
```

(continues on next page)

(continued from previous page)

```
 ['L2', 'R[2]'],
 ['R[0]', 'R[]'],
 ['R[1]', 'R[]'],
 ['R[2]', 'R[]']]
```

```
>>> from sage.all import *
>>> sorted(A.faces()[Integer(0)])
↪     # needs sage.graphs
[('L0',), ('L1',), ('L2',), ('R[0]',), ('R[1]',), ('R[2]',), ('R[]',)]
>>> sorted(map(sorted, A.faces()[Integer(1)]))
↪     # needs sage.graphs
[['L0', 'L1'],
 ['L0', 'L2'],
 ['L0', 'R[0]'],
 ['L1', 'L2'],
 ['L1', 'R[1]'],
 ['L2', 'R[2]'],
 ['R[0]', 'R[]'],
 ['R[1]', 'R[]'],
 ['R[2]', 'R[]']]
```

See also

M.bergman_complex()

Todo

It is possible that this method could be optimized by building up the maximal chains using a sort of dynamic programming approach.

REFERENCES:

- [BHMPW20a]
- [BHMPW20b]

automorphism_group()

Return the automorphism group of `self`.

For a matroid M, an automorphism is a permutation σ of $E(M)$ (the groundset) such that $r(X) = r(\sigma(X))$ for all $X \subseteq E(M)$. The set of automorphisms of M forms a group under composition. This automorphism group is transitive if, for every two elements x and y of M, there is an automorphism that maps x to y.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: G = M.automorphism_group()
sage: G.is_transitive()
True
sage: G.structure_description()
'PSL(3,2)'
```

(continues on next page)

(continued from previous page)

```
sage: M = matroids.catalog.P8pp()
sage: M.automorphism_group().is_transitive()
True
sage: M = matroids.catalog.ExtendedTernaryGolayCode()
sage: G = M.automorphism_group()
sage: G.is_transitive()
True
sage: G.structure_description()
'M12'
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> G = M.automorphism_group()
>>> G.is_transitive()
True
>>> G.structure_description()
'PSL(3,2)'
>>> M = matroids.catalog.P8pp()
>>> M.automorphism_group().is_transitive()
True
>>> M = matroids.catalog.ExtendedTernaryGolayCode()
>>> G = M.automorphism_group()
>>> G.is_transitive()
True
>>> G.structure_description()
'M12'
```

REFERENCES:

[Oxl2011], p. 189.

bases()

Return the bases of the matroid.

A *basis* is a maximal independent set.

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: M = matroids.Uniform(2, 4)
sage: sorted([sorted(X) for X in M.bases()])
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2), Integer(4))
>>> sorted([sorted(X) for X in M.bases()])
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

ALGORITHM:

Test all subsets of the groundset of cardinality `self.full_rank()`

See also

M.independent_sets()

bases_iterator()

Return an iterator over the bases of the matroid.

A *basis* is a maximal independent set.

ALGORITHM:

Test all subsets of the groundset of cardinality `self.full_rank()`.

EXAMPLES:

```
sage: M = matroids.Uniform(2, 4)
sage: sorted([sorted(X) for X in M.bases_iterator()])
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2), Integer(4))
>>> sorted([sorted(X) for X in M.bases_iterator()])
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

See also

M.independent_sets_iterator()

basis()

Return an arbitrary basis of the matroid.

A *basis* is an inclusionwise maximal independent set.

Note

The output of this method can change in between calls.

OUTPUT: a set of elements

EXAMPLES:

```
sage: M = matroids.catalog.Pappus()
sage: B = M.basis()
sage: M.is_basis(B)
True
sage: len(B)
3
sage: M.rank(B)
3
sage: M.full_rank()
3
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus()
>>> B = M.basis()
>>> M.is_basis(B)
True
>>> len(B)
3
>>> M.rank(B)
3
>>> M.full_rank()
3
```

bergman_complex()

Return the Bergman complex of `self`.

Let L be the lattice of flats of a matroid M with the minimum and maximum elements removed. The *Bergman complex* of a matroid M is the order complex of L.

OUTPUT: a simplicial complex

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: B = M.bergman_complex(); B                                                #␣
↪needs sage.graphs
Simplicial complex with 14 vertices and 21 facets
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> B = M.bergman_complex(); B                                                  #␣
↪needs sage.graphs
Simplicial complex with 14 vertices and 21 facets
```

See also

M.augmented_bergman_complex()

binary_matroid(*randomized_tests=1*, *verify=True*)

Return a binary matroid representing `self`, if such a representation exists.

INPUT:

- `randomized_tests` – (default: 1) an integer; the number of times a certain necessary condition for being binary is tested, using randomization
- `verify` – boolean (default: `True`); if `True`, any output will be a binary matroid representing `self`; if `False`, any output will represent `self` if and only if the matroid is binary

OUTPUT: either a `BinaryMatroid`, or `None`

ALGORITHM:

First, compare the binary matroids local to two random bases. If these matroids are not isomorphic, return `None`. This test is performed `randomized_tests` times. Next, if `verify` is `True`, test if a binary matroid local to some basis is isomorphic to `self`.

See also

`M.local_binary_matroid()`

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.binary_matroid()
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
sage: N = matroids.catalog.NonFano()
sage: N.binary_matroid() is None
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.binary_matroid()
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
>>> N = matroids.catalog.NonFano()
>>> N.binary_matroid() is None
True
```

broken_circuit_complex(*ordering=None*)

Return the broken circuit complex of `self`.

The broken circuit complex of a matroid with a total ordering $<$ on the groundset is obtained from the *NBC sets* under subset inclusion.

INPUT:

- `ordering` – list (optional); a total ordering of the groundset

OUTPUT: a simplicial complex of the NBC sets under inclusion

EXAMPLES:

```
sage: M = Matroid(circuits=[[1,2,3], [3,4,5], [1,2,4,5]])
sage: M.broken_circuit_complex()                                              #␣
↪needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
sage: M.broken_circuit_complex([5,4,3,2,1])                                   #␣
↪needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1),Integer(2),Integer(3)], [Integer(3),
↪Integer(4),Integer(5)], [Integer(1),Integer(2),Integer(4),Integer(5)]])
>>> M.broken_circuit_complex()                                                #␣
↪needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
>>> M.broken_circuit_complex([Integer(5),Integer(4),Integer(3),Integer(2),
↪Integer(1)])                                  # needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

For a matroid with loops, the broken circuit complex is not defined, and the method yields an error:

```
sage: M = Matroid(flats={0: ['a'], 1: ['ab', 'ac'], 2: ['abc']})
sage: M.broken_circuit_complex()
Traceback (most recent call last):
...
ValueError: broken circuit complex of matroid with loops is not defined
```

```
>>> from sage.all import *
>>> M = Matroid(flats={Integer(0): ['a'], Integer(1): ['ab', 'ac'],␣
↪Integer(2): ['abc']})
>>> M.broken_circuit_complex()
Traceback (most recent call last):
...
ValueError: broken circuit complex of matroid with loops is not defined
```

broken_circuits(*ordering=None*)

Return the broken circuits of `self`.

Let M be a matroid with groundset E, and let $<$ be a total ordering on E. A *broken circuit* for M means a subset B of E such that there exists a $u \in E$ for which $B \cup \{u\}$ is a circuit of M and $u < b$ for all $b \in B$.

INPUT:

- `ordering` – list (optional); a total ordering of the groundset

EXAMPLES:

```
sage: M = Matroid(circuits=[[1,2,3], [3,4,5], [1,2,4,5]])
sage: sorted([sorted(X) for X in M.broken_circuits()])
[[2, 3], [2, 4, 5], [4, 5]]
sage: sorted([sorted(X) for X in M.broken_circuits([5,4,3,2,1])])
[[1, 2], [1, 2, 4], [3, 4]]
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1),Integer(2),Integer(3)], [Integer(3),
↪Integer(4),Integer(5)], [Integer(1),Integer(2),Integer(4),Integer(5)]])
>>> sorted([sorted(X) for X in M.broken_circuits()])
[[2, 3], [2, 4, 5], [4, 5]]
>>> sorted([sorted(X) for X in M.broken_circuits([Integer(5),Integer(4),
↪Integer(3),Integer(2),Integer(1)])])
[[1, 2], [1, 2, 4], [3, 4]]
```

```
sage: M = Matroid(circuits=[[1,2,3], [1,4,5], [2,3,4,5]])
sage: sorted([sorted(X) for X in M.broken_circuits([5,4,3,2,1])])
[[1, 2], [1, 4], [2, 3, 4]]
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1),Integer(2),Integer(3)], [Integer(1),
↪Integer(4),Integer(5)], [Integer(2),Integer(3),Integer(4),Integer(5)]])
```

(continues on next page)

(continued from previous page)

```
>>> sorted([sorted(X) for X in M.broken_circuits([Integer(5),Integer(4),
↪Integer(3),Integer(2),Integer(1)])])
[[1, 2], [1, 4], [2, 3, 4]]
```

characteristic_polynomial(*la=None*)

Return the characteristic polynomial of the matroid.

The *characteristic polynomial* of a matroid M is the polynomial

$$\chi_M(\lambda) = \sum_{S \subseteq E} (-1)^{|S|} \lambda^{r(E)-r(S)},$$

where E is the groundset and r is the matroid's rank function. The characteristic polynomial is also equal to $\sum_{i=0}^{r} w_i \lambda^{r-i}$, where $\{w_i\}_{i=0}^{r}$ are the Whitney numbers of the first kind.

INPUT:

- `la` – a variable or numerical argument (optional)

OUTPUT: the characteristic polynomial, $\chi_M(\lambda)$, where λ is substituted with any value provided as input

EXAMPLES:

```
sage: M = matroids.CompleteGraphic(5)
sage: M.characteristic_polynomial()
l^4 - 10*l^3 + 35*l^2 - 50*l + 24
sage: M.characteristic_polynomial().factor()
(l - 4) * (l - 3) * (l - 2) * (l - 1)
sage: M.characteristic_polynomial(5)
24
```

```
>>> from sage.all import *
>>> M = matroids.CompleteGraphic(Integer(5))
>>> M.characteristic_polynomial()
l^4 - 10*l^3 + 35*l^2 - 50*l + 24
>>> M.characteristic_polynomial().factor()
(l - 4) * (l - 3) * (l - 2) * (l - 1)
>>> M.characteristic_polynomial(Integer(5))
24
```

See also

whitney_numbers()

chordality()

Return the minimal k such that the matroid `M` is k-chordal.

See also

M.is_chordal()

EXAMPLES:

```
sage: M = matroids.Uniform(2,4)
sage: M.chordality()
4
sage: M = matroids.catalog.NonFano()
sage: M.chordality()
5
sage: M = matroids.catalog.Fano()
sage: M.chordality()
4
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2),Integer(4))
>>> M.chordality()
4
>>> M = matroids.catalog.NonFano()
>>> M.chordality()
5
>>> M = matroids.catalog.Fano()
>>> M.chordality()
4
```

chow_ring(*R*, *augmented=False*, *presentation=None*)

Return the (augmented) Chow ring of `self` over `R`.

See also

- *sage.matroids.chow_ring_ideal*
- *sage.matroids.chow_ring*

INPUT:

- `M` – matroid
- `R` – commutative ring
- `augmented` – boolean (default: `False`); when `True`, this is the augmented Chow ring and if `False`, this is the non-augmented Chow ring
- `presentation` – string; if `augmented=True`, then this must be one of the following (ignored if `augmented=False`):
  - `"fy"` - the Feitchner-Yuzvinsky presentation
  - `"atom-free"` - the atom-free presentation

EXAMPLES:

```
sage: M = matroids.Wheel(2)
sage: A = M.chow_ring(R=ZZ, augmented=False); A
Chow ring of Wheel(2): Regular matroid of rank 2 on 4 elements with
5 bases over Integer Ring
sage: A.defining_ideal()._gens_constructor(A.defining_ideal().ring())
[A0*A1, A0*A23, A1*A23, A0 + A0123, A1 + A0123, A23 + A0123]
sage: A23 = A.gen(0)
```

(continues on next page)

(continued from previous page)

```
sage: A23*A23
0
```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(2))
>>> A = M.chow_ring(R=ZZ, augmented=False); A
Chow ring of Wheel(2): Regular matroid of rank 2 on 4 elements with
5 bases over Integer Ring
>>> A.defining_ideal()._gens_constructor(A.defining_ideal().ring())
[A0*A1, A0*A23, A1*A23, A0 + A0123, A1 + A0123, A23 + A0123]
>>> A23 = A.gen(Integer(0))
>>> A23*A23
0
```

We construct a more interesting example using the Fano matroid:

```
sage: M = matroids.catalog.Fano()
sage: A = M.chow_ring(QQ); A
Chow ring of Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
over Rational Field
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> A = M.chow_ring(QQ); A
Chow ring of Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
over Rational Field
```

Next we get the non-trivial generators and do some computations:

```
sage: # needs sage.libs.singular sage.rings.finite_rings
sage: G = A.gens()[7:]; G
(Aabf, Aace, Aadg, Abcd, Abeg, Acfg, Adef, Aabcdefg)
sage: Aabf, Aace, Aadg, Abcd, Abeg, Acfg, Adef, Aabcdefg = G
sage: Aabf*Aabf
-Aabcdefg^2
sage: Aabf*Acfg
0
sage: matrix([[x * y for x in G] for y in G])
[-Aabcdefg^2           0           0           0           0           0 ␣
↪      0           0]
[          0 -Aabcdefg^2           0           0           0           0 ␣
↪      0           0]
[          0           0 -Aabcdefg^2           0           0           0 ␣
↪      0           0]
[          0           0           0 -Aabcdefg^2           0           0 ␣
↪      0           0]
[          0           0           0           0 -Aabcdefg^2           0 ␣
↪      0           0]
[          0           0           0           0           0 -Aabcdefg^2 ␣
↪      0           0]
[          0           0           0           0           0           0 -
↪Aabcdefg^2           0]
```

(continues on next page)

(continued from previous page)

```
[           0            0            0            0            0            0
↪      0  Aabcdefg^2]
```

```
>>> from sage.all import *
>>> # needs sage.libs.singular sage.rings.finite_rings
>>> G = A.gens()[Integer(7):]; G
(Aabf, Aace, Aadg, Abcd, Abeg, Acfg, Adef, Aabcdefg)
>>> Aabf, Aace, Aadg, Abcd, Abeg, Acfg, Adef, Aabcdefg = G
>>> Aabf*Aabf
-Aabcdefg^2
>>> Aabf*Acfg
0
>>> matrix([[x * y for x in G] for y in G])
[-Aabcdefg^2            0            0            0            0            0
↪      0            0]
[           0 -Aabcdefg^2            0            0            0            0
↪      0            0]
[           0            0 -Aabcdefg^2            0            0            0
↪      0            0]
[           0            0            0 -Aabcdefg^2            0            0
↪      0            0]
[           0            0            0            0 -Aabcdefg^2            0
↪      0            0]
[           0            0            0            0            0 -Aabcdefg^2
↪      0            0]
[           0            0            0            0            0            0 -
↪Aabcdefg^2            0]
[           0            0            0            0            0            0
↪      0  Aabcdefg^2]
```

The augmented Chow ring can also be constructed with the Feitchner-Yuzvinsky and atom-free presentation:

```
sage: M = matroids.Wheel(3)
sage: ch = M.chow_ring(QQ, augmented=True, presentation='fy'); ch
Augmented Chow ring of Wheel(3): Regular matroid of rank 3 on
6 elements with 16 bases in Feitchner-Yuzvinsky presentation over
Rational Field
sage: M = matroids.Uniform(3, 6)
sage: ch = M.chow_ring(QQ, augmented=True, presentation='atom-free'); ch
Augmented Chow ring of U(3, 6): Matroid of rank 3 on 6 elements with circuit-
↪closures
{3: {{0, 1, 2, 3, 4, 5}}} in atom-free presentation over Rational Field
```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> ch = M.chow_ring(QQ, augmented=True, presentation='fy'); ch
Augmented Chow ring of Wheel(3): Regular matroid of rank 3 on
6 elements with 16 bases in Feitchner-Yuzvinsky presentation over
Rational Field
>>> M = matroids.Uniform(Integer(3), Integer(6))
>>> ch = M.chow_ring(QQ, augmented=True, presentation='atom-free'); ch
Augmented Chow ring of U(3, 6): Matroid of rank 3 on 6 elements with circuit-
```

(continues on next page)

(continued from previous page)

```
↪closures
{3: {{0, 1, 2, 3, 4, 5}}} in atom-free presentation over Rational Field
```

circuit(*X=None*)

Return a circuit.

A *circuit* of a matroid is an inclusionwise minimal dependent subset.

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset

OUTPUT: a set of elements

- If `X` is not `None`, the output is a circuit contained in `X` if such a circuit exists. Otherwise an error is raised.
- If `X` is `None`, the output is a circuit contained in `self.groundset()` if such a circuit exists. Otherwise an error is raised.

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: sorted(M.circuit(['a', 'c', 'd', 'e', 'f']))
['c', 'd', 'e', 'f']
sage: sorted(M.circuit(['a', 'c', 'd']))
Traceback (most recent call last):
...
ValueError: no circuit in independent set
sage: M.circuit(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
sage: C = M.circuit()
sage: sorted(C)  # random
['a', 'b', 'c', 'd']
sage: M.is_circuit(C)
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> sorted(M.circuit(['a', 'c', 'd', 'e', 'f']))
['c', 'd', 'e', 'f']
>>> sorted(M.circuit(['a', 'c', 'd']))
Traceback (most recent call last):
...
ValueError: no circuit in independent set
>>> M.circuit(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
>>> C = M.circuit()
>>> sorted(C)  # random
['a', 'b', 'c', 'd']
>>> M.is_circuit(C)
True
```

circuit_closures()

Return the closures of circuits of the matroid.

A *circuit closure* is a closed set containing a circuit.

OUTPUT: a dictionary containing the circuit closures of the matroid, indexed by their ranks

See also

M.circuit(), *M.closure()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: CC = M.circuit_closures()
sage: len(CC[2])
7
sage: len(CC[3])
1
sage: len(CC[1])
Traceback (most recent call last):
...
KeyError: 1
sage: [sorted(X) for X in CC[3]]
[['a', 'b', 'c', 'd', 'e', 'f', 'g']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> CC = M.circuit_closures()
>>> len(CC[Integer(2)])
7
>>> len(CC[Integer(3)])
1
>>> len(CC[Integer(1)])
Traceback (most recent call last):
...
KeyError: 1
>>> [sorted(X) for X in CC[Integer(3)]]
[['a', 'b', 'c', 'd', 'e', 'f', 'g']]
```

circuits(*k=None*)

Return the circuits of the matroid.

INPUT:

- `k` – integer (optional); if provided, return only circuits of length k

OUTPUT: `SetSystem`

See also

M.circuit()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted([sorted(C) for C in M.circuits()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'b', 'f'],
['a', 'c', 'd', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['a', 'e', 'f', 'g'], ['b', 'c', 'd'], ['b', 'c', 'e', 'f'],
['b', 'd', 'f', 'g'], ['b', 'e', 'g'], ['c', 'd', 'e', 'g'],
['c', 'f', 'g'], ['d', 'e', 'f']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted([sorted(C) for C in M.circuits()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'b', 'f'],
['a', 'c', 'd', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['a', 'e', 'f', 'g'], ['b', 'c', 'd'], ['b', 'c', 'e', 'f'],
['b', 'd', 'f', 'g'], ['b', 'e', 'g'], ['c', 'd', 'e', 'g'],
['c', 'f', 'g'], ['d', 'e', 'f']]
```

circuits_iterator(*k=None*)

Return an iterator over the circuits of the matroid.

See also

circuit()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted([sorted(C) for C in M.circuits_iterator()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'b', 'f'],
['a', 'c', 'd', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['a', 'e', 'f', 'g'], ['b', 'c', 'd'], ['b', 'c', 'e', 'f'],
['b', 'd', 'f', 'g'], ['b', 'e', 'g'], ['c', 'd', 'e', 'g'],
['c', 'f', 'g'], ['d', 'e', 'f']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted([sorted(C) for C in M.circuits_iterator()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'b', 'f'],
['a', 'c', 'd', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['a', 'e', 'f', 'g'], ['b', 'c', 'd'], ['b', 'c', 'e', 'f'],
['b', 'd', 'f', 'g'], ['b', 'e', 'g'], ['c', 'd', 'e', 'g'],
['c', 'f', 'g'], ['d', 'e', 'f']]
```

closure(*X*)

Return the closure of a set X.

A set is *closed* if adding any extra element to it will increase the rank of the set. The *closure* of a set is the smallest closed set containing it.

INPUT:

- X – a subset (or any iterable) of the groundset

OUTPUT: superset of X

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: sorted(M.closure(set(['a', 'b', 'c'])))
['a', 'b', 'c', 'd']
sage: M.closure(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> sorted(M.closure(set(['a', 'b', 'c'])))
['a', 'b', 'c', 'd']
>>> M.closure(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

cobasis()

Return an arbitrary cobasis of the matroid.

A *cobasis* is the complement of a basis. A cobasis is a basis of the dual matroid.

Note

Output can change between calls.

OUTPUT: a set of elements

See also

M.dual(), *M.full_rank()*

EXAMPLES:

```
sage: M = matroids.catalog.Pappus()
sage: B = M.cobasis()
sage: M.is_cobasis(B)
True
sage: len(B)
6
sage: M.corank(B)
6
sage: M.full_corank()
6
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus()
>>> B = M.cobasis()
>>> M.is_cobasis(B)
```

(continues on next page)

(continued from previous page)

```
True
>>> len(B)
6
>>> M.corank(B)
6
>>> M.full_corank()
6
```

cocircuit(*X=None*)

Return a cocircuit.

A *cocircuit* is an inclusionwise minimal subset that is dependent in the dual matroid.

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset

OUTPUT: a set of elements

- If `X` is not `None`, the output is a cocircuit contained in `X` if such a cocircuit exists. Otherwise an error is raised.
- If `X` is `None`, the output is a cocircuit contained in `self.groundset()` if such a cocircuit exists. Otherwise an error is raised.

See also

M.dual(), *M.circuit()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: sorted(M.cocircuit(['a', 'c', 'd', 'e', 'f']))
['c', 'd', 'e', 'f']
sage: sorted(M.cocircuit(['a', 'c', 'd']))
Traceback (most recent call last):
...
ValueError: no cocircuit in coindependent set.
sage: M.cocircuit(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
sage: C = M.cocircuit()
sage: sorted(C)  # random
['e', 'f', 'g', 'h']
sage: M.is_cocircuit(C)
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> sorted(M.cocircuit(['a', 'c', 'd', 'e', 'f']))
['c', 'd', 'e', 'f']
>>> sorted(M.cocircuit(['a', 'c', 'd']))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: no cocircuit in coindependent set.
>>> M.cocircuit(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
>>> C = M.cocircuit()
>>> sorted(C)  # random
['e', 'f', 'g', 'h']
>>> M.is_cocircuit(C)
True
```

cocircuits()

Return the cocircuits of the matroid.

OUTPUT: SetSystem

See also

M.cocircuit()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted([sorted(C) for C in M.cocircuits()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'c', 'd', 'f'],
['a', 'e', 'f', 'g'], ['b', 'c', 'e', 'f'], ['b', 'd', 'f', 'g'],
['c', 'd', 'e', 'g']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted([sorted(C) for C in M.cocircuits()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'c', 'd', 'f'],
['a', 'e', 'f', 'g'], ['b', 'c', 'e', 'f'], ['b', 'd', 'f', 'g'],
['c', 'd', 'e', 'g']]
```

cocircuits_iterator()

Return an iterator over the cocircuits of the matroid.

See also

M.cocircuit()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted([sorted(C) for C in M.cocircuits_iterator()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'c', 'd', 'f'],
['a', 'e', 'f', 'g'], ['b', 'c', 'e', 'f'], ['b', 'd', 'f', 'g'],
['c', 'd', 'e', 'g']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted([sorted(C) for C in M.cocircuits_iterator()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'c', 'd', 'f'],
['a', 'e', 'f', 'g'], ['b', 'c', 'e', 'f'], ['b', 'd', 'f', 'g'],
['c', 'd', 'e', 'g']]
```

coclosure(*X*)

Return the coclosure of a set `X`.

A set is *coclosed* if it is closed in the dual matroid. The *coclosure* of X is the smallest coclosed set containing X.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: superset of `X`

See also

M.dual(), *M.closure()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: sorted(M.coclosure(set(['a', 'b', 'c'])))
['a', 'b', 'c', 'd']
sage: M.coclosure(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> sorted(M.coclosure(set(['a', 'b', 'c'])))
['a', 'b', 'c', 'd']
>>> M.coclosure(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

coextension(*element=None*, *subsets=None*)

Return a coextension of the matroid.

A *coextension* of M by an element e is a matroid M' such that $M'/e = M$. The element `element` is placed such that it lies in the *coclosure* of each set in `subsets`, and otherwise as freely as possible.

This is the dual method of *M.extension()*. See the documentation there for more details.

INPUT:

- `element` – (default: `None`) the label of the new element. If not specified, a new label will be generated automatically.

- `subsets` – (default: `None`) a set of subsets of the matroid. The coextension should be such that the new element is in the cospan of each of these. If not specified, the element is assumed to be in the cospan of the full groundset.

OUTPUT: matroid

See also

M.dual(), *M.coextensions()*, *M.modular_cut()*, *M.extension()*, *M.linear_subclasses()*, *sage.matroids.extension*

EXAMPLES:

Add an element in general position:

```
sage: M = matroids.Uniform(3, 6)
sage: N = M.coextension(6)
sage: N.is_isomorphic(matroids.Uniform(4, 7))
True
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(3), Integer(6))
>>> N = M.coextension(Integer(6))
>>> N.is_isomorphic(matroids.Uniform(Integer(4), Integer(7)))
True
```

Add one inside the span of a specified hyperplane:

```
sage: M = matroids.Uniform(3, 6)
sage: H = [frozenset([0, 1])]
sage: N = M.coextension(6, H)
sage: N
Matroid of rank 4 on 7 elements with 34 bases
sage: [sorted(C) for C in N.cocircuits() if len(C) == 3]
[[0, 1, 6]]
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(3), Integer(6))
>>> H = [frozenset([Integer(0), Integer(1)])]
>>> N = M.coextension(Integer(6), H)
>>> N
Matroid of rank 4 on 7 elements with 34 bases
>>> [sorted(C) for C in N.cocircuits() if len(C) == Integer(3)]
[[0, 1, 6]]
```

Put an element in series with another:

```
sage: M = matroids.catalog.Fano()
sage: N = M.coextension('z', ['c'])
sage: N.corank('cz')
1
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> N = M.coextension('z', ['c'])
>>> N.corank('cz')
1
```

coextensions(*element=None*, *coline_length=None*, *subsets=None*)

Return an iterable set of single-element coextensions of the matroid.

A *coextension* of a matroid M by element e is a matroid M' such that $M'/e = M$. By default, this method returns an iterable containing all coextensions, but it can be restricted in two ways. If `coline_length` is specified, the output is restricted to those matroids not containing a coline minor of length k greater than `coline_length`. If `subsets` is specified, then the output is restricted to those matroids for which the new element lies in the *coclosure* of each member of `subsets`.

This method is dual to *M.extensions()*.

INPUT:

- `element` – (optional) the name of the newly added element in each coextension.
- `coline_length` – (optional) a natural number. If given, restricts the output to coextensions that do not contain a $U_{k-2,k}$ minor where `k > coline_length`.
- `subsets` – (optional) a collection of subsets of the groundset. If given, restricts the output to extensions where the new element is contained in all cohyperplanes that contain an element of `subsets`.

OUTPUT: an iterable containing matroids

Note

The coextension by a coloop will always occur. The extension by a loop will never occur.

See also

M.coextension(), *M.modular_cut()*, *M.linear_subclasses()*, *sage.matroids.extension*, *M.extensions()*, *M.dual()*

EXAMPLES:

```
sage: M = matroids.catalog.P8()
sage: len(list(M.coextensions()))
1705
sage: len(list(M.coextensions(coline_length=4)))
41
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
sage: len(list(M.coextensions(subsets=[{'a', 'b'}], coline_length=4)))
5
```

```
>>> from sage.all import *
>>> M = matroids.catalog.P8()
>>> len(list(M.coextensions()))
```

(continues on next page)

(continued from previous page)

```
1705
>>> len(list(M.coextensions(coline_length=Integer(4))))
41
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> len(list(M.coextensions(subsets=[{'a', 'b'}], coline_length=Integer(4))))
5
```

coflats(k)

Return the collection of coflats of the matroid of specified corank.

A *coflat* is a coclosed set.

INPUT:

- `k` – integer

OUTPUT: `SetSystem`

See also

M.coclosure()

EXAMPLES:

```
sage: M = matroids.catalog.Q6()                                              #␣
↪needs sage.rings.finite_rings
sage: sorted([sorted(F) for F in M.coflats(2)])                              #␣
↪needs sage.rings.finite_rings
[['a', 'b'], ['a', 'c'], ['a', 'd', 'f'], ['a', 'e'], ['b', 'c'],
['b', 'd'], ['b', 'e'], ['b', 'f'], ['c', 'd'], ['c', 'e', 'f'],
['d', 'e']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Q6()                                                #␣
↪needs sage.rings.finite_rings
>>> sorted([sorted(F) for F in M.coflats(Integer(2))])                        ␣
↪     # needs sage.rings.finite_rings
[['a', 'b'], ['a', 'c'], ['a', 'd', 'f'], ['a', 'e'], ['b', 'c'],
['b', 'd'], ['b', 'e'], ['b', 'f'], ['c', 'd'], ['c', 'e', 'f'],
['d', 'e']]
```

coloops()

Return the set of coloops of the matroid.

A *coloop* is an element u of the groundset such that the one-element set $\{u\}$ is a cocircuit. In other words, a coloop is a loop of the dual of the matroid.

OUTPUT: a set of elements

See also

M.dual(), *M.loops()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano().dual()
sage: M.coloops()
frozenset()
sage: (M.delete(['a', 'b'])).coloops()
frozenset({'f'})
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano().dual()
>>> M.coloops()
frozenset()
>>> (M.delete(['a', 'b'])).coloops()
frozenset({'f'})
```

components()

Return a list of the components of the matroid.

A *component* is an inclusionwise maximal connected subset of the matroid. A subset is *connected* if the matroid resulting from deleting the complement of that subset is *connected*.

OUTPUT: list of subsets

See also

M.is_connected(), *M.delete()*

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = Matroid(ring=QQ, matrix=[[1, 0, 0, 1, 1, 0],
....:                              [0, 1, 0, 1, 2, 0],
....:                              [0, 0, 1, 0, 0, 1]])
sage: setprint(M.components())
[{0, 1, 3, 4}, {2, 5}]
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = Matroid(ring=QQ, matrix=[[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(0)],
...                              [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(0)],
...                              [Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)]])
>>> setprint(M.components())
[{0, 1, 3, 4}, {2, 5}]
```

connectivity(*S*, *T=None*)

Evaluate the connectivity function of the matroid.

If the input is a single subset S of the groundset E, then the output is $r(S) + r(E \S) - r(E)$.

If the input are disjoint subsets S, T of the groundset, then the output is

$$\min\{r(X) + r(Y) - r(E) \mid X \subseteq S, Y \subseteq T, X, Y \text{a partition of} E\}.$$

INPUT:

- `S` – a subset (or any iterable) of the groundset
- `T` – (optional) a subset (or any iterable) of the groundset disjoint from `S`

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.BetsyRoss()
sage: M.connectivity('ab')
2
sage: M.connectivity('ab', 'cd')
2
```

```
>>> from sage.all import *
>>> M = matroids.catalog.BetsyRoss()
>>> M.connectivity('ab')
2
>>> M.connectivity('ab', 'cd')
2
```

contract(X)

Contract elements.

If e is a non-loop element, then the matroid M/e is a matroid on groundset $E(M)-e$. A set X is independent in M/e if and only if $X \cup e$ is independent in M. If e is a loop then contracting e is the same as deleting e. We say that M/e is the matroid obtained from M by *contracting* e. Contracting an element in M is the same as deleting an element in the dual of M.

When contracting a set, the elements of that set are contracted one by one. It can be shown that the resulting matroid does not depend on the order of the contractions.

Sage supports the shortcut notation `M / X` for `M.contract(X)`.

INPUT:

- `X` – either a single element of the groundset, or a collection of elements

OUTPUT: the matroid obtained by contracting the element(s) in `X`

See also

M.delete() M.dual() M.minor()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
sage: M.contract(['a', 'c'])
Binary matroid of rank 1 on 5 elements, type (1, 0)
sage: M.contract(['a']) == M / ['a']
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> M.contract(['a', 'c'])
Binary matroid of rank 1 on 5 elements, type (1, 0)
>>> M.contract(['a']) == M / ['a']
True
```

One can use a single element, rather than a set:

```
sage: M = matroids.CompleteGraphic(4)                                    #␣
↪needs sage.graphs
sage: M.contract(1) == M.contract([1])                                   #␣
↪needs sage.graphs
True
sage: M / 1                                                              #␣
↪needs sage.graphs
Graphic matroid of rank 2 on 5 elements
```

```
>>> from sage.all import *
>>> M = matroids.CompleteGraphic(Integer(4))                             ␣
↪     # needs sage.graphs
>>> M.contract(Integer(1)) == M.contract([Integer(1)])                   ␣
↪              # needs sage.graphs
True
>>> M / Integer(1)                                                       ␣
↪     # needs sage.graphs
Graphic matroid of rank 2 on 5 elements
```

Note that one can iterate over strings:

```
sage: M = matroids.catalog.Fano()
sage: M / 'abc'
Binary matroid of rank 0 on 4 elements, type (0, 0)
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M / 'abc'
Binary matroid of rank 0 on 4 elements, type (0, 0)
```

The following is therefore ambiguous. Sage will contract the single element:

```
sage: M = Matroid(groundset=['a', 'b', 'c', 'abc'],
....:             bases=[['a', 'b', 'c'], ['a', 'b', 'abc']])
sage: sorted((M / 'abc').groundset())
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> M = Matroid(groundset=['a', 'b', 'c', 'abc'],
...             bases=[['a', 'b', 'c'], ['a', 'b', 'abc']])
>>> sorted((M / 'abc').groundset())
['a', 'b', 'c']
```

corank(*X=None*)

Return the corank of `X`, or the corank of the groundset if `X` is `None`.

The *corank* of a set X is the rank of X in the dual matroid.

If `X` is `None`, the corank of the groundset is returned.

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset

OUTPUT: integer

See also

M.dual(), *M.rank()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.corank()
4
sage: M.corank('cdeg')
3
sage: M.rank(['a', 'b', 'x'])
Traceback (most recent call last):
...
ValueError: ['a', 'b', 'x'] is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.corank()
4
>>> M.corank('cdeg')
3
>>> M.rank(['a', 'b', 'x'])
Traceback (most recent call last):
...
ValueError: ['a', 'b', 'x'] is not a subset of the groundset
```

cosimplify()

Return the cosimplification of the matroid.

A matroid is *cosimple* if it contains no cocircuits of length 1 or 2. The *cosimplification* of a matroid is obtained by contracting all coloops (cocircuits of length 1) and contracting all but one element from each series class (a coclosed set of rank 1, that is, each pair in it forms a cocircuit of length 2).

OUTPUT: matroid

See also

M.is_cosimple(), *M.coloops()*, *M.cocircuit()*, *M.simplify()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano().dual().delete('a')
sage: M.cosimplify().size()
3
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano().dual().delete('a')
>>> M.cosimplify().size()
3
```

delete(X)

Delete elements.

If e is an element, then the matroid $M \setminus e$ is a matroid on groundset $E(M) - e$. A set X is independent in $M \setminus e$ if and only if X is independent in M. We say that $M \setminus e$ is the matroid obtained from M by *deleting* e.

When deleting a set, the elements of that set are deleted one by one. It can be shown that the resulting matroid does not depend on the order of the deletions.

DEPRECATED: Sage supports the shortcut notation `M \ X` for `M.delete(X)`.

INPUT:

- `X` – either a single element of the groundset, or a collection of elements

OUTPUT: the matroid obtained by deleting the element(s) in `X`

See also

M.contract() M.minor()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
sage: M.delete(['a', 'c'])
Binary matroid of rank 3 on 5 elements, type (1, 6)
sage: M.delete(['a']) == M.delete(['a'])
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> M.delete(['a', 'c'])
Binary matroid of rank 3 on 5 elements, type (1, 6)
>>> M.delete(['a']) == M.delete(['a'])
True
```

One can use a single element, rather than a set:

```
sage: M = matroids.CompleteGraphic(4)                                    #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
sage: M.delete(1) == M.delete([1])                                          #␣
↪needs sage.graphs
True
sage: M.delete(1)                                                           #␣
↪needs sage.graphs
Graphic matroid of rank 3 on 5 elements
```

```
>>> from sage.all import *
>>> M = matroids.CompleteGraphic(Integer(4))                                ␣
↪     # needs sage.graphs
>>> M.delete(Integer(1)) == M.delete([Integer(1)])                          ␣
↪              # needs sage.graphs
True
>>> M.delete(Integer(1))                                                    ␣
↪     # needs sage.graphs
Graphic matroid of rank 3 on 5 elements
```

Note that one can iterate over strings:

```
sage: M = matroids.catalog.Fano()
sage: M.delete('abc')
Binary matroid of rank 3 on 4 elements, type (0, 5)
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.delete('abc')
Binary matroid of rank 3 on 4 elements, type (0, 5)
```

The following is therefore ambiguous. Sage will delete the single element:

```
sage: M = Matroid(groundset=['a', 'b', 'c', 'abc'],
....:             bases=[['a', 'b', 'c'], ['a', 'b', 'abc']])
sage: sorted((M.delete('abc')).groundset())
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> M = Matroid(groundset=['a', 'b', 'c', 'abc'],
...             bases=[['a', 'b', 'c'], ['a', 'b', 'abc']])
>>> sorted((M.delete('abc')).groundset())
['a', 'b', 'c']
```

dependent_r_sets(**args*, ***kwds*)

Deprecated: Use *dependent_sets()* instead. See Issue #38057 for details.

dependent_sets(*k*)

Return the dependent sets of fixed size.

INPUT:

- k – integer

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.dependent_sets(3)
SetSystem of 0 sets over 8 elements
sage: sorted([sorted(X) for X in
....: matroids.catalog.Vamos().dependent_sets(4)])
[['a', 'b', 'c', 'd'], ['a', 'b', 'e', 'f'], ['a', 'b', 'g', 'h'],
['c', 'd', 'e', 'f'], ['e', 'f', 'g', 'h']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.dependent_sets(Integer(3))
SetSystem of 0 sets over 8 elements
>>> sorted([sorted(X) for X in
... matroids.catalog.Vamos().dependent_sets(Integer(4))])
[['a', 'b', 'c', 'd'], ['a', 'b', 'e', 'f'], ['a', 'b', 'g', 'h'],
['c', 'd', 'e', 'f'], ['e', 'f', 'g', 'h']]
```

ALGORITHM:

Test all subsets of the groundset of cardinality k.

dependent_sets_iterator(k)

Return an iterator over the dependent sets of fixed size.

INPUT:

- `k` – integer

ALGORITHM:

Test all subsets of the groundset of cardinality k.

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: list(M.dependent_sets_iterator(3))
[]
sage: sorted([sorted(X) for X in
....: matroids.catalog.Vamos().dependent_sets_iterator(4)])
[['a', 'b', 'c', 'd'], ['a', 'b', 'e', 'f'], ['a', 'b', 'g', 'h'],
['c', 'd', 'e', 'f'], ['e', 'f', 'g', 'h']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> list(M.dependent_sets_iterator(Integer(3)))
[]
>>> sorted([sorted(X) for X in
... matroids.catalog.Vamos().dependent_sets_iterator(Integer(4))])
[['a', 'b', 'c', 'd'], ['a', 'b', 'e', 'f'], ['a', 'b', 'g', 'h'],
['c', 'd', 'e', 'f'], ['e', 'f', 'g', 'h']]
```

direct_sum(*matroids*)

Return the matroid direct sum with another matroid or list of matroids.

Let $(M_1, M_2, \ldots, M_k)$ be a list of matroids where each M_i has groundset E_i. The matroid sum of $(E_1, I_1), \ldots, (E_n, I_n)$ is a matroid (E, I) where $E = \bigsqcup_{i=1}^{n} E_i$ and $I = \bigsqcup_{i=1}^{n} I_i$.

INPUT:

- `matroids` – matroid or list of matroids

OUTPUT: an instance of `MatroidSum`

EXAMPLES:

```
sage: M = matroids.catalog.Pappus()
sage: N = matroids.catalog.Fano().direct_sum(M); N
Matroid of rank 6 on 16 elements as matroid sum of
Binary matroid of rank 3 on 7 elements, type (3, 0)
Matroid of rank 3 on 9 elements with 9 nonspanning circuits
sage: len(N.independent_sets())
6897
sage: len(N.bases())
2100
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus()
>>> N = matroids.catalog.Fano().direct_sum(M); N
Matroid of rank 6 on 16 elements as matroid sum of
Binary matroid of rank 3 on 7 elements, type (3, 0)
Matroid of rank 3 on 9 elements with 9 nonspanning circuits
>>> len(N.independent_sets())
6897
>>> len(N.bases())
2100
```

dual()

Return the dual of the matroid.

Let M be a matroid with groundset E. If B is the set of bases of M, then the set $\{E - b : b \in B\}$ is the set of bases of another matroid, the *dual* of M.

Note

This function wraps `self` in a `DualMatroid` object. For more efficiency, subclasses that can, should override this method.

EXAMPLES:

```
sage: M = matroids.catalog.Pappus()
sage: N = M.dual()
sage: N.rank()
6
sage: N
Dual of 'Pappus: Matroid of rank 3 on 9 elements with 9 nonspanning circuits'
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus()
>>> N = M.dual()
>>> N.rank()
6
```

(continues on next page)

(continued from previous page)

```
>>> N
Dual of 'Pappus: Matroid of rank 3 on 9 elements with 9 nonspanning circuits'
```

equals(*other*)

Test for matroid equality.

Two matroids M and N are *equal* if they have the same groundset and a subset X is independent in M if and only if it is independent in N.

INPUT:

- `other` – matroid

OUTPUT: boolean

Note

This method tests abstract matroid equality. The == operator takes a more restricted view: `M == N` returns `True` only if

1. the internal representations are of the same type,
2. those representations are equivalent (for an appropriate meaning of "equivalent" in that class), and
3. `M.equals(N)`.

EXAMPLES:

A *BinaryMatroid* and *BasisMatroid* use different representations of the matroid internally, so == yields `False`, even if the matroids are equal:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.catalog.Fano(); M
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
sage: M1 = BasisMatroid(M)
sage: M2 = Matroid(groundset='abcdefg', reduced_matrix=[
....:       [0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 0, 1]], field=GF(2))
sage: M.equals(M1)
True
sage: M.equals(M2)
True
sage: M == M1
False
sage: M == M2
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.Fano(); M
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
>>> M1 = BasisMatroid(M)
>>> M2 = Matroid(groundset='abcdefg', reduced_matrix=[
...       [Integer(0), Integer(1), Integer(1), Integer(1)], [Integer(1),␣
↪Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1), Integer(0),␣
↪Integer(1)]], field=GF(Integer(2)))
```

(continues on next page)

(continued from previous page)

```
>>> M.equals(M1)
True
>>> M.equals(M2)
True
>>> M == M1
False
>>> M == M2
True
```

LinearMatroid instances `M` and `N` satisfy `M == N` if the representations are equivalent up to row operations and column scaling:

```
sage: M1 = LinearMatroid(groundset='abcd', matrix=Matrix(GF(7),
....:                                [[1, 0, 1, 1], [0, 1, 1, 2]]))
sage: M2 = LinearMatroid(groundset='abcd', matrix=Matrix(GF(7),
....:                                [[1, 0, 1, 1], [0, 1, 1, 3]]))
sage: M3 = LinearMatroid(groundset='abcd', matrix=Matrix(GF(7),
....:                                [[2, 6, 1, 0], [6, 1, 0, 1]]))
sage: M1.equals(M2)
True
sage: M1.equals(M3)
True
sage: M1 == M2
False
sage: M1 == M3
True
```

```
>>> from sage.all import *
>>> M1 = LinearMatroid(groundset='abcd', matrix=Matrix(GF(Integer(7)),
...                                [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2)]]))
>>> M2 = LinearMatroid(groundset='abcd', matrix=Matrix(GF(Integer(7)),
...                                [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(3)]]))
>>> M3 = LinearMatroid(groundset='abcd', matrix=Matrix(GF(Integer(7)),
...                                [[Integer(2), Integer(6), Integer(1),␣
↪Integer(0)], [Integer(6), Integer(1), Integer(0), Integer(1)]]))
>>> M1.equals(M2)
True
>>> M1.equals(M3)
True
>>> M1 == M2
False
>>> M1 == M3
True
```

extension(*element=None*, *subsets=None*)

Return an extension of the matroid.

An *extension* of M by an element e is a matroid M' such that $M' \setminus e = M$. The element `element` is placed such that it lies in the *closure* of each set in `subsets`, and otherwise as freely as possible. More precisely, the extension is defined by the *modular cut* generated by the sets in `subsets`.

INPUT:

- `element` – (default: `None`) the label of the new element. If not specified, a new label will be generated automatically.
- `subsets` – (default: `None`) a set of subsets of the matroid. The extension should be such that the new element is in the span of each of these. If not specified, the element is assumed to be in the span of the full groundset.

OUTPUT: matroid

Note

Internally, sage uses the notion of a *linear subclass* for matroid extension. If `subsets` already consists of a linear subclass (i.e. the set of hyperplanes of a modular cut) then the faster method `M._extension()` can be used.

See also

M.extensions(), *M.modular_cut()*, *M.coextension()*, *M.linear_subclasses()*, *sage.matroids.extension*

EXAMPLES:

First we add an element in general position:

```
sage: M = matroids.Uniform(3, 6)
sage: N = M.extension(6)
sage: N.is_isomorphic(matroids.Uniform(3, 7))
True
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(3), Integer(6))
>>> N = M.extension(Integer(6))
>>> N.is_isomorphic(matroids.Uniform(Integer(3), Integer(7)))
True
```

Next we add one inside the span of a specified hyperplane:

```
sage: M = matroids.Uniform(3, 6)
sage: H = [frozenset([0, 1])]
sage: N = M.extension(6, H)
sage: N
Matroid of rank 3 on 7 elements with 34 bases
sage: [sorted(C) for C in N.circuits() if len(C) == 3]
[[0, 1, 6]]
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(3), Integer(6))
>>> H = [frozenset([Integer(0), Integer(1)])]
>>> N = M.extension(Integer(6), H)
>>> N
Matroid of rank 3 on 7 elements with 34 bases
>>> [sorted(C) for C in N.circuits() if len(C) == Integer(3)]
[[0, 1, 6]]
```

Putting an element in parallel with another:

```
sage: M = matroids.catalog.Fano()
sage: N = M.extension('z', ['c'])
sage: N.rank('cz')
1
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> N = M.extension('z', ['c'])
>>> N.rank('cz')
1
```

extensions(*element=None*, *line_length=None*, *subsets=None*)

Return an iterable set of single-element extensions of the matroid.

An *extension* of a matroid M by element e is a matroid M' such that $M' \setminus e = M$. By default, this method returns an iterable containing all extensions, but it can be restricted in two ways. If `line_length` is specified, the output is restricted to those matroids not containing a line minor of length k greater than `line_length`. If `subsets` is specified, then the output is restricted to those matroids for which the new element lies in the *closure* of each member of `subsets`.

INPUT:

- `element` – (optional) the name of the newly added element in each extension.
- `line_length` – (optional) a natural number. If given, restricts the output to extensions that do not contain a $U_{2,k}$ minor where `k > line_length`.
- `subsets` – (optional) a collection of subsets of the groundset. If given, restricts the output to extensions where the new element is contained in all hyperplanes that contain an element of `subsets`.

OUTPUT: an iterable containing matroids

Note

The extension by a loop will always occur. The extension by a coloop will never occur.

See also

M.extension(), *M.modular_cut()*, *M.linear_subclasses()*, *sage.matroids.extension*, *M.coextensions()*

EXAMPLES:

```
sage: M = matroids.catalog.P8()
sage: len(list(M.extensions()))
1705
sage: len(list(M.extensions(line_length=4)))
41
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
sage: len(list(M.extensions(subsets=[{'a', 'b'}], line_length=4)))
5
```

```
>>> from sage.all import *
>>> M = matroids.catalog.P8()
>>> len(list(M.extensions()))
1705
>>> len(list(M.extensions(line_length=Integer(4))))
41
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> len(list(M.extensions(subsets=[{'a', 'b'}], line_length=Integer(4))))
5
```

f_vector()

Return the f-vector of the matroid.

The f-*vector* is a vector $(f_0, \ldots, f_r)$, where f_i is the number of independent sets of rank i, and r is the rank of the matroid.

OUTPUT: list of integers

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.f_vector()
[1, 8, 28, 56, 65]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.f_vector()
[1, 8, 28, 56, 65]
```

flat_cover(*solver=None*, *verbose=0*, *integrality_tolerance=0.001*)

Return a minimum-size cover of the nonbases by nonspanning flats.

A *nonbasis* is a subset that has the size of a basis, yet is dependent. A *flat* is a closed set.

INPUT:

- `solver` – (default: `None`) specify a Linear Program (LP) solver to be used. If set to `None`, the default one is used. For more information on LP solvers and which default solver is used, see the method `solve()` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: 0); sets the level of verbosity of the LP solver. Set to 0 by default, which means quiet.

See also

M.nonbases(), *M.flats()*

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Fano()
sage: setprint(M.flat_cover())                                                  #␣
↪needs sage.rings.finite_rings
[{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'},
```

(continues on next page)

(continued from previous page)

```
 {'b', 'c', 'd'}, {'b', 'e', 'g'}, {'c', 'f', 'g'},
 {'d', 'e', 'f'}]
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Fano()
>>> setprint(M.flat_cover())                                                #␣
↪needs sage.rings.finite_rings
[{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'},
 {'b', 'c', 'd'}, {'b', 'e', 'g'}, {'c', 'f', 'g'},
 {'d', 'e', 'f'}]
```

flats(*k*)

Return the collection of flats of the matroid of specified rank.

A *flat* is a closed set.

INPUT:

- k – integer

OUTPUT: SetSystem

See also

M.closure()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted([sorted(F) for F in M.flats(2)])
[['a', 'b', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['b', 'c', 'd'], ['b', 'e', 'g'], ['c', 'f', 'g'],
['d', 'e', 'f']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted([sorted(F) for F in M.flats(Integer(2))])
[['a', 'b', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['b', 'c', 'd'], ['b', 'e', 'g'], ['c', 'f', 'g'],
['d', 'e', 'f']]
```

full_corank()

Return the corank of the matroid.

The *corank* of the matroid equals the rank of the dual matroid. It is given by M.size() - M.full_rank().

OUTPUT: integer

See also

M.dual(), *M.full_rank()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.full_corank()
4
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.full_corank()
4
```

full_rank()

Return the rank of the matroid.

The *rank* of the matroid is the size of the largest independent subset of the groundset.

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.full_rank()
4
sage: M.dual().full_rank()
4
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.full_rank()
4
>>> M.dual().full_rank()
4
```

fundamental_circuit(B, e)

Return the B-fundamental circuit using e.

If B is a basis, and e an element not in B, then the B-*fundamental circuit* using e is the unique matroid circuit contained in $B \cup e$.

INPUT:

- `B` – a basis of the matroid
- `e` – an element not in `B`

OUTPUT: a set of elements

See also

M.circuit(), *M.basis()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: sorted(M.fundamental_circuit('defg', 'c'))
['c', 'd', 'e', 'f']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> sorted(M.fundamental_circuit('defg', 'c'))
['c', 'd', 'e', 'f']
```

fundamental_cocircuit(*B*, *e*)

Return the B-fundamental cocircuit using e.

If B is a basis, and e an element of B, then the *B-fundamental cocircuit* using e is the unique matroid cocircuit that intersects B only in e.

This is equal to `M.dual().fundamental_circuit(M.groundset().difference(B), e)`.

INPUT:

- `B` – a basis of the matroid
- `e` – an element of `B`

OUTPUT: a set of elements

See also

M.cocircuit(), M.basis(), M.fundamental_circuit()

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: sorted(M.fundamental_cocircuit('abch', 'c'))
['c', 'd', 'e', 'f']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> sorted(M.fundamental_cocircuit('abch', 'c'))
['c', 'd', 'e', 'f']
```

girth()

Return the girth of the matroid.

The girth is the size of the smallest circuit. In case the matroid has no circuits the girth is ∞.

EXAMPLES:

```
sage: matroids.Uniform(5, 5).girth()
+Infinity
sage: matroids.catalog.K4().girth()
3
sage: matroids.catalog.Vamos().girth()
4
```

```
>>> from sage.all import *
>>> matroids.Uniform(Integer(5), Integer(5)).girth()
+Infinity
>>> matroids.catalog.K4().girth()
3
```

(continues on next page)

(continued from previous page)

```
>>> matroids.catalog.Vamos().girth()
4
```

REFERENCES:

[Oxl2011], p. 327.

groundset()

Return the groundset of the matroid.

The groundset is the set of elements that comprise the matroid.

OUTPUT: `frozenset`

Note

Subclasses should implement this method. The return type should be frozenset or any type with compatible interface.

EXAMPLES:

```
sage: M = sage.matroids.matroid.Matroid()
sage: M.groundset()
Traceback (most recent call last):
...
NotImplementedError: subclasses need to implement this
```

```
>>> from sage.all import *
>>> M = sage.matroids.matroid.Matroid()
>>> M.groundset()
Traceback (most recent call last):
...
NotImplementedError: subclasses need to implement this
```

has_line_minor(*k*, *hyperlines=None*, *certificate=False*)

Test if the matroid has a $U_{2,k}$-minor.

The matroid $U_{2,k}$ is a matroid on k elements in which every subset of at most 2 elements is independent, and every subset of more than two elements is dependent.

The optional argument `hyperlines` restricts the search space: this method returns `True` if $si(M/F)$ is isomorphic to $U_{2,l}$ with $l \geq k$ for some F in `hyperlines`, and `False` otherwise.

INPUT:

- `k` – the length of the line minor
- `hyperlines` – (default: `None`) a set of flats of codimension 2. Defaults to the set of all flats of codimension 2.
- `certificate` – boolean (default: `False`); if `True` returns `(True, F)`, where `F` is a flat and `self.minor(contractions=F)` has a $U_{2,k}$ restriction or `(False, None)`.

OUTPUT: boolean or tuple

See also

Matroid.has_minor()

EXAMPLES:

```
sage: M = matroids.catalog.N1()
sage: M.has_line_minor(4)
True
sage: M.has_line_minor(5)
False
sage: M.has_line_minor(k=4, hyperlines=[['a', 'b', 'c']])
False
sage: M.has_line_minor(k=4, hyperlines=[['a', 'b', 'c'],
....:                                  ['a', 'b', 'd' ]])
True
sage: M.has_line_minor(4, certificate=True)
(True, frozenset({'a', 'b', 'd'}))
sage: M.has_line_minor(5, certificate=True)
(False, None)
sage: M.has_line_minor(k=4, hyperlines=[['a', 'b', 'c'],
....:                                  ['a', 'b', 'd' ]], certificate=True)
(True, frozenset({'a', 'b', 'd'}))
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N1()
>>> M.has_line_minor(Integer(4))
True
>>> M.has_line_minor(Integer(5))
False
>>> M.has_line_minor(k=Integer(4), hyperlines=[['a', 'b', 'c']])
False
>>> M.has_line_minor(k=Integer(4), hyperlines=[['a', 'b', 'c'],
...                                  ['a', 'b', 'd' ]])
True
>>> M.has_line_minor(Integer(4), certificate=True)
(True, frozenset({'a', 'b', 'd'}))
>>> M.has_line_minor(Integer(5), certificate=True)
(False, None)
>>> M.has_line_minor(k=Integer(4), hyperlines=[['a', 'b', 'c'],
...                                  ['a', 'b', 'd' ]], certificate=True)
(True, frozenset({'a', 'b', 'd'}))
```

has_minor(*N*, *certificate=False*)

Check if `self` has a minor isomorphic to `N`, and optionally return frozensets `X` and `Y` so that `N` is isomorphic to `self.minor(X, Y)`.

INPUT:

- `N` – an instance of a *Matroid* object
- `certificate` – boolean (default: `False`); if `True`, returns `True, (X, Y, dic)` where `N` is isomorphic to `self.minor(X, Y)`, and `dic` is an isomorphism between `N` and `self.minor(X, Y)`

OUTPUT: boolean or tuple

See also

M.minor(), *M.is_isomorphic()*

Todo

This important method can (and should) be optimized considerably. See [Hli2006] p.1219 for hints to that end.

EXAMPLES:

```
sage: M = matroids.Whirl(3)
sage: matroids.catalog.Fano().has_minor(M)
False
sage: matroids.catalog.NonFano().has_minor(M)
True
sage: matroids.catalog.NonFano().has_minor(M, certificate=True)
(True, (frozenset(), frozenset({...}), {...}))
sage: M = matroids.catalog.Fano()
sage: M.has_minor(M, True)
(True,
 (frozenset(),
  frozenset(),
  {'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd', 'e': 'e', 'f': 'f', 'g': 'g'}))
```

```
>>> from sage.all import *
>>> M = matroids.Whirl(Integer(3))
>>> matroids.catalog.Fano().has_minor(M)
False
>>> matroids.catalog.NonFano().has_minor(M)
True
>>> matroids.catalog.NonFano().has_minor(M, certificate=True)
(True, (frozenset(), frozenset({...}), {...}))
>>> M = matroids.catalog.Fano()
>>> M.has_minor(M, True)
(True,
 (frozenset(),
  frozenset(),
  {'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd', 'e': 'e', 'f': 'f', 'g': 'g'}))
```

hyperplanes()

Return the hyperplanes of the matroid.

A *hyperplane* is a flat of rank `self.full_rank() - 1`. A *flat* is a closed set.

OUTPUT: `SetSystem`

See also

M.flats()

EXAMPLES:

```
sage: M = matroids.Uniform(2, 3)
sage: sorted([sorted(F) for F in M.hyperplanes()])
[[0], [1], [2]]
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2), Integer(3))
>>> sorted([sorted(F) for F in M.hyperplanes()])
[[0], [1], [2]]
```

independence_matroid_polytope()

Return the independence matroid polytope of `self`.

This is defined as the convex hull of the vertices

$$\sum_{i \in I} e_i$$

over all independent sets I of the matroid. Here e_i are the standard basis vectors of $\mathbf{R}^n$. An arbitrary labelling of the groundset by $\{0, \ldots, n-1\}$ is chosen.

See also

matroid_polytope()

EXAMPLES:

```
sage: M = matroids.Whirl(4)
sage: M.independence_matroid_polytope()                                 #␣
↪needs sage.geometry.polyhedron sage.rings.finite_rings
A 8-dimensional polyhedron in ZZ^8 defined as the convex hull
of 135 vertices

sage: M = matroids.catalog.NonFano()
sage: M.independence_matroid_polytope()                                 #␣
↪needs sage.geometry.polyhedron sage.rings.finite_rings
A 7-dimensional polyhedron in ZZ^7 defined as the convex hull
of 58 vertices
```

```
>>> from sage.all import *
>>> M = matroids.Whirl(Integer(4))
>>> M.independence_matroid_polytope()                                   #␣
↪needs sage.geometry.polyhedron sage.rings.finite_rings
A 8-dimensional polyhedron in ZZ^8 defined as the convex hull
of 135 vertices

>>> M = matroids.catalog.NonFano()
>>> M.independence_matroid_polytope()                                   #␣
↪needs sage.geometry.polyhedron sage.rings.finite_rings
A 7-dimensional polyhedron in ZZ^7 defined as the convex hull
of 58 vertices
```

REFERENCES:

[DLHK2007]

independent_r_sets(**args*, ***kwds*)

Deprecated: Use *independent_sets()* instead. See Issue #38057 for details.

independent_sets(*k=-1*)

Return the independent sets of the matroid.

INPUT:

- k – integer (optional); if specified, return the size-k independent sets of the matroid

OUTPUT: SetSystem

EXAMPLES:

```
sage: M = matroids.catalog.Pappus()
sage: I = M.independent_sets()
sage: len(I)
121
sage: M.independent_sets(4)
SetSystem of 0 sets over 9 elements
sage: S = M.independent_sets(3); S
SetSystem of 75 sets over 9 elements
sage: frozenset({'a', 'c', 'e'}) in S
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus()
>>> I = M.independent_sets()
>>> len(I)
121
>>> M.independent_sets(Integer(4))
SetSystem of 0 sets over 9 elements
>>> S = M.independent_sets(Integer(3)); S
SetSystem of 75 sets over 9 elements
>>> frozenset({'a', 'c', 'e'}) in S
True
```

See also

M.bases()

independent_sets_iterator(*k=None*)

Return an iterator over the independent sets of the matroid.

INPUT:

- k – integer (optional); if specified, return an iterator over the size-k independent sets of the matroid

EXAMPLES:

```
sage: M = matroids.catalog.Pappus()
sage: I = list(M.independent_sets_iterator())
sage: len(I)
121
sage: M = matroids.catalog.Pappus()
```

(continues on next page)

(continued from previous page)

```
sage: list(M.independent_sets_iterator(4))
[]
sage: S = list(M.independent_sets_iterator(3))
sage: len(S)
75
sage: frozenset({'a', 'c', 'e'}) in S
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus()
>>> I = list(M.independent_sets_iterator())
>>> len(I)
121
>>> M = matroids.catalog.Pappus()
>>> list(M.independent_sets_iterator(Integer(4)))
[]
>>> S = list(M.independent_sets_iterator(Integer(3)))
>>> len(S)
75
>>> frozenset({'a', 'c', 'e'}) in S
True
```

See also

M.bases_iterator()

intersection(*other*, *weights=None*)

Return a maximum-weight common independent set.

A *common independent set* of matroids M and N with the same groundset E is a subset of E that is independent both in M and N. The *weight* of a subset `S` is `sum(weights(e) for e in S)`.

INPUT:

- `other` – a second matroid with the same groundset as this matroid
- `weights` – (default: `None`) a dictionary which specifies a weight for each element of the common groundset; defaults to the all-1 weight function

OUTPUT: a subset of the groundset

EXAMPLES:

```
sage: M = matroids.catalog.T12()
sage: N = matroids.catalog.ExtendedTernaryGolayCode()
sage: w = {'a':30, 'b':10, 'c':11, 'd':20, 'e':70, 'f':21, 'g':90,
....:      'h':12, 'i':80, 'j':13, 'k':40, 'l':21}
sage: Y = M.intersection(N, w)
sage: sorted(Y)
['a', 'd', 'e', 'g', 'i', 'k']
sage: sum([w[y] for y in Y])
330
sage: M = matroids.catalog.Fano()
```

(continues on next page)

(continued from previous page)

```
sage: N = matroids.Uniform(4, 7)
sage: M.intersection(N)
Traceback (most recent call last):
...
ValueError: matroid intersection requires equal groundsets.
```

```
>>> from sage.all import *
>>> M = matroids.catalog.T12()
>>> N = matroids.catalog.ExtendedTernaryGolayCode()
>>> w = {'a':Integer(30), 'b':Integer(10), 'c':Integer(11), 'd':Integer(20),
↪'e':Integer(70), 'f':Integer(21), 'g':Integer(90),
...      'h':Integer(12), 'i':Integer(80), 'j':Integer(13), 'k':Integer(40),
↪'l':Integer(21)}
>>> Y = M.intersection(N, w)
>>> sorted(Y)
['a', 'd', 'e', 'g', 'i', 'k']
>>> sum([w[y] for y in Y])
330
>>> M = matroids.catalog.Fano()
>>> N = matroids.Uniform(Integer(4), Integer(7))
>>> M.intersection(N)
Traceback (most recent call last):
...
ValueError: matroid intersection requires equal groundsets.
```

intersection_unweighted(*other*)

Return a maximum-cardinality common independent set.

A *common independent set* of matroids M and N with the same groundset E is a subset of E that is independent both in M and N.

INPUT:

- `other` – a second matroid with the same groundset as this matroid

OUTPUT: subset of the groundset

EXAMPLES:

```
sage: M = matroids.catalog.T12()
sage: N = matroids.catalog.ExtendedTernaryGolayCode()
sage: len(M.intersection_unweighted(N))
6
sage: M = matroids.catalog.Fano()
sage: N = matroids.Uniform(4, 7)
sage: M.intersection_unweighted(N)
Traceback (most recent call last):
...
ValueError: matroid intersection requires equal groundsets.
```

```
>>> from sage.all import *
>>> M = matroids.catalog.T12()
>>> N = matroids.catalog.ExtendedTernaryGolayCode()
>>> len(M.intersection_unweighted(N))
```

(continues on next page)

(continued from previous page)

```
6
>>> M = matroids.catalog.Fano()
>>> N = matroids.Uniform(Integer(4), Integer(7))
>>> M.intersection_unweighted(N)
Traceback (most recent call last):
...
ValueError: matroid intersection requires equal groundsets.
```

is_3connected(*certificate=False*, *algorithm=None*)

Return `True` if the matroid is 3-connected, `False` otherwise. It can optionally return a separator as a witness.

A k-*separation* in a matroid is a partition (X, Y) of the groundset with $|X| \geq k$, $|Y| \geq k$ and $r(X)+r(Y)-r(M) < k$. A matroid is k-*connected* if it has no l-separations for $l < k$.

INPUT:

- `certificate` – boolean (default: `False`); if `True`, then return `True, None` if the matroid is 3-connected, and `False, X` otherwise, where X is a < 3-separation
- `algorithm` – (default: `None`) specify which algorithm to compute 3-connectivity:
  - `None` – the most appropriate algorithm is chosen automatically
  - `'bridges'` – Bixby and Cunningham's algorithm, based on bridges [BC1977]; note that this cannot return a separator
  - `'intersection'` – an algorithm based on matroid intersection
  - `'shifting'` – an algorithm based on the shifting algorithm [Raj1987]

OUTPUT: boolean, or a tuple `(boolean, frozenset)`

See also

M.is_connected() M.is_4connected() M.is_kconnected()

ALGORITHM:

- Bridges based: The 3-connectivity algorithm from [BC1977] which runs in $O((r(E))^2|E|)$ time.
- Matroid intersection based: Evaluates the connectivity between $O(|E|^2)$ pairs of disjoint sets S, T with $|S| = |T| = 2$.
- Shifting algorithm: The shifting algorithm from [Raj1987] which runs in $O((r(E))^2|E|)$ time.

EXAMPLES:

```
sage: matroids.Uniform(2, 3).is_3connected()
True
sage: M = Matroid(ring=QQ, matrix=[[1, 0, 0, 1, 1, 0],
....:                              [0, 1, 0, 1, 2, 0],
....:                              [0, 0, 1, 0, 0, 1]])
sage: M.is_3connected()
False
sage: M.is_3connected() == M.is_3connected(algorithm='bridges')
True
sage: M.is_3connected() == M.is_3connected(algorithm='intersection')
True
```

(continues on next page)

(continued from previous page)

```
sage: N = Matroid(circuit_closures={2: ['abc', 'cdef'],
....:                               3: ['abcdef']},
....:             groundset='abcdef')
sage: N.is_3connected()
False
sage: matroids.catalog.BetsyRoss().is_3connected()                              #␣
↪needs sage.graphs
True
sage: M = matroids.catalog.R6()
sage: M.is_3connected()                                                         #␣
↪needs sage.graphs
False
sage: B, X = M.is_3connected(True)
sage: M.connectivity(X)
1
```

```
>>> from sage.all import *
>>> matroids.Uniform(Integer(2), Integer(3)).is_3connected()
True
>>> M = Matroid(ring=QQ, matrix=[[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(0)],
...                           [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(0)],
...                           [Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)]])
>>> M.is_3connected()
False
>>> M.is_3connected() == M.is_3connected(algorithm='bridges')
True
>>> M.is_3connected() == M.is_3connected(algorithm='intersection')
True
>>> N = Matroid(circuit_closures={Integer(2): ['abc', 'cdef'],
...                               Integer(3): ['abcdef']},
...             groundset='abcdef')
>>> N.is_3connected()
False
>>> matroids.catalog.BetsyRoss().is_3connected()                              #␣
↪needs sage.graphs
True
>>> M = matroids.catalog.R6()
>>> M.is_3connected()                                                         #␣
↪needs sage.graphs
False
>>> B, X = M.is_3connected(True)
>>> M.connectivity(X)
1
```

is_4connected(*certificate=False*, *algorithm=None*)

Return `True` if the matroid is 4-connected, `False` otherwise. It can optionally return a separator as a witness.

INPUT:

- `certificate` – boolean (default: `False`); if `True`, then return `True`, `None` if the matroid is

4-connected, and `False`, X otherwise, where X is a < 4-separation

- `algorithm` – (default: `None`) specify which algorithm to compute 4-connectivity:
  - `None` – the most appropriate algorithm is chosen automatically
  - `'intersection'` – an algorithm based on matroid intersection, equivalent to calling `is_kconnected(4, certificate)`
  - `'shifting'` – an algorithm based on the shifting algorithm [Raj1987]

OUTPUT: boolean, or a tuple `(boolean, frozenset)`

See also

M.is_connected() M.is_3connected() M.is_kconnected()

EXAMPLES:

```
sage: M = matroids.Uniform(2, 6)
sage: B, X = M.is_4connected(True)
sage: (B, M.connectivity(X)<=3)
(False, True)
sage: matroids.Uniform(4, 8).is_4connected()
True
sage: M = Matroid(field=GF(2), matrix=[[1,0,0,1,0,1,1,0,0,1,1,1],
....:                                  [0,1,0,1,0,1,0,1,0,0,0,1],
....:                                  [0,0,1,1,0,0,1,1,0,1,0,1],
....:                                  [0,0,0,0,1,1,1,1,0,0,1,1],
....:                                  [0,0,0,0,0,0,0,0,1,1,1,1]])
sage: M.is_4connected() == M.is_4connected(algorithm='shifting')             #␣
↪needs sage.graphs
True
sage: M.is_4connected() == M.is_4connected(algorithm='intersection')
True
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2), Integer(6))
>>> B, X = M.is_4connected(True)
>>> (B, M.connectivity(X)<=Integer(3))
(False, True)
>>> matroids.Uniform(Integer(4), Integer(8)).is_4connected()
True
>>> M = Matroid(field=GF(Integer(2)), matrix=[[Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(1)],
...                                  [Integer(0),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(0),Integer(0),Integer(1)],
...                                  [Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(0),Integer(0),Integer(1),Integer(1),Integer(0),
↪Integer(1),Integer(0),Integer(1)],
...                                  [Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1),Integer(1),Integer(1),Integer(1),Integer(0),
↪Integer(0),Integer(1),Integer(1)],
```

(continues on next page)

(continued from previous page)

```
...                                   [Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(1),
↪Integer(1),Integer(1),Integer(1)]])
>>> M.is_4connected() == M.is_4connected(algorithm='shifting')            #␣
↪needs sage.graphs
True
>>> M.is_4connected() == M.is_4connected(algorithm='intersection')
True
```

is_basis(*X*)

Check if a subset is a basis of the matroid.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: boolean

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_basis('abc')
False
sage: M.is_basis('abce')
True
sage: M.is_basis('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_basis('abc')
False
>>> M.is_basis('abce')
True
>>> M.is_basis('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_binary(*randomized_tests=1*)

Decide if `self` is a binary matroid.

INPUT:

- `randomized_tests` – (default: 1) an integer; the number of times a certain necessary condition for being binary is tested, using randomization

OUTPUT: boolean

ALGORITHM:

First, compare the binary matroids local to two random bases. If these matroids are not isomorphic, return `False`. This test is performed `randomized_tests` times. Next, test if a binary matroid local to some basis is isomorphic to `self`.

See also

M.binary_matroid()

EXAMPLES:

```
sage: N = matroids.catalog.Fano()
sage: N.is_binary()
True
sage: N = matroids.catalog.NonFano()
sage: N.is_binary()
False
```

```
>>> from sage.all import *
>>> N = matroids.catalog.Fano()
>>> N.is_binary()
True
>>> N = matroids.catalog.NonFano()
>>> N.is_binary()
False
```

is_chordal(*k1=4*, *k2=None*, *certificate=False*)

Return if a matroid is `[k1, k2]`-chordal.

A matroid M is $[k_1, k_2]$-chordal if every circuit of length ℓ with $k_1 \leq \ell \leq k_2$ has a *chord*. We say M is k-chordal if $k_1 = k$ and $k_2 = \infty$. We call M *chordal* if it is 4-chordal.

INPUT:

- `k1` – (optional) the integer k_1
- `k2` – (optional) the integer k_2; if not specified, then this method returns if `self` is k_1-chordal
- `certificate` – boolean (default: `False`); if `True` return `True, C`, where `C` is a non `k1 k2` circuit

OUTPUT: boolean or tuple

See also

M.chordality()

EXAMPLES:

```
sage: M = matroids.Uniform(2,4)
sage: [M.is_chordal(i) for i in range(4, 8)]
[True, True, True, True]
sage: M = matroids.catalog.NonFano()
sage: [M.is_chordal(i) for i in range(4, 8)]
[False, True, True, True]
sage: M = matroids.catalog.N2()
sage: [M.is_chordal(i) for i in range(4, 10)]
[False, False, False, False, True, True]
sage: M.is_chordal(4, 5)
False
```

(continues on next page)

(continued from previous page)

```
sage: M.is_chordal(4, 5, certificate=True)
(False, frozenset({...}))
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2),Integer(4))
>>> [M.is_chordal(i) for i in range(Integer(4), Integer(8))]
[True, True, True, True]
>>> M = matroids.catalog.NonFano()
>>> [M.is_chordal(i) for i in range(Integer(4), Integer(8))]
[False, True, True, True]
>>> M = matroids.catalog.N2()
>>> [M.is_chordal(i) for i in range(Integer(4), Integer(10))]
[False, False, False, False, True, True]
>>> M.is_chordal(Integer(4), Integer(5))
False
>>> M.is_chordal(Integer(4), Integer(5), certificate=True)
(False, frozenset({...}))
```

is_circuit(*X*)

Test if a subset is a circuit of the matroid.

A *circuit* is an inclusionwise minimal dependent subset.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: boolean

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_circuit('abc')
False
sage: M.is_circuit('abcd')
True
sage: M.is_circuit('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_circuit('abc')
False
>>> M.is_circuit('abcd')
True
>>> M.is_circuit('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_circuit_chordal(*C*, *certificate=False*)

Check if the circuit `C` has a chord.

A circuit C in a matroid M has a *chord* $x \in E$ if there exists sets A, B such that $C = A \sqcup B$ and $A + x$ and $B + x$ are circuits.

INPUT:

- `C` – a circuit
- `certificate` – boolean (default: `False`); if `True` return `True, (x, Ax, Bx)`, where `x` is a chord and `Ax` and `Bx` are circuits whose union is the elements of `C` together with `x`, if `False` return `False, None`

OUTPUT: boolean or tuple

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.is_circuit_chordal(['b','c','d'])
False
sage: M.is_circuit_chordal(['b','c','d'], certificate=True)
(False, None)
sage: M.is_circuit_chordal(['a','b','d','e'])
True
sage: X = M.is_circuit_chordal(frozenset(['a','b','d','e']),
....:                          certificate=True)[1]
sage: X  # random
('c', frozenset({'b', 'c', 'd'}), frozenset({'a', 'c', 'e'}))
sage: M.is_circuit(X[1]) and M.is_circuit(X[2])
True
sage: X[1].intersection(X[2]) == frozenset([X[0]])
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.is_circuit_chordal(['b','c','d'])
False
>>> M.is_circuit_chordal(['b','c','d'], certificate=True)
(False, None)
>>> M.is_circuit_chordal(['a','b','d','e'])
True
>>> X = M.is_circuit_chordal(frozenset(['a','b','d','e']),
...                          certificate=True)[Integer(1)]
>>> X  # random
('c', frozenset({'b', 'c', 'd'}), frozenset({'a', 'c', 'e'}))
>>> M.is_circuit(X[Integer(1)]) and M.is_circuit(X[Integer(2)])
True
>>> X[Integer(1)].intersection(X[Integer(2)]) == frozenset([X[Integer(0)]])
True
```

is_closed(X)

Test if a subset is a closed set of the matroid.

A set is *closed* if adding any element to it will increase the rank of the set.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: boolean

See also

M.closure()

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_closed('abc')
False
sage: M.is_closed('abcd')
True
sage: M.is_closed('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_closed('abc')
False
>>> M.is_closed('abcd')
True
>>> M.is_closed('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_cobasis(*X*)

Check if a subset is a cobasis of the matroid.

A *cobasis* is the complement of a basis. It is a basis of the dual matroid.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: boolean

See also

M.dual(), *M.is_basis()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_cobasis('abc')
False
sage: M.is_cobasis('abce')
True
sage: M.is_cobasis('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_cobasis('abc')
False
>>> M.is_cobasis('abce')
True
>>> M.is_cobasis('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_cocircuit(*X*)

Test if a subset is a cocircuit of the matroid.

A *cocircuit* is an inclusionwise minimal subset that is dependent in the dual matroid.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: boolean

See also

M.dual(), *M.is_circuit()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_cocircuit('abc')
False
sage: M.is_cocircuit('abcd')
True
sage: M.is_cocircuit('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_cocircuit('abc')
False
>>> M.is_cocircuit('abcd')
True
>>> M.is_cocircuit('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_coclosed(*X*)

Test if a subset is a coclosed set of the matroid.

A set is *coclosed* if it is a closed set of the dual matroid.

INPUT:

- X – a subset (or any iterable) of the groundset

OUTPUT: boolean

See also

M.dual(), *M.is_closed()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_coclosed('abc')
False
sage: M.is_coclosed('abcd')
True
sage: M.is_coclosed('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_coclosed('abc')
False
>>> M.is_coclosed('abcd')
True
>>> M.is_coclosed('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_codependent(*X*)

Check if a subset is codependent in the matroid.

A set is *codependent* if it is dependent in the dual of the matroid.

INPUT:

- X – a subset (or any iterable) of the groundset

OUTPUT: boolean

See also

M.dual(), *M.is_dependent()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_codependent('abc')
False
sage: M.is_codependent('abcd')
True
sage: M.is_codependent('abcx')
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_codependent('abc')
False
>>> M.is_codependent('abcd')
True
>>> M.is_codependent('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_coindependent(X)

Check if a subset is coindependent in the matroid.

A set is *coindependent* if it is independent in the dual matroid.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: boolean

See also

M.dual(), *M.is_independent()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_coindependent('abc')
True
sage: M.is_coindependent('abcd')
False
sage: M.is_coindependent('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_coindependent('abc')
True
>>> M.is_coindependent('abcd')
False
>>> M.is_coindependent('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_connected(*certificate=False*)

Test if the matroid is connected.

A *separation* in a matroid is a partition (X, Y) of the groundset with X, Y nonempty and $r(X) + r(Y) = r(X \cup Y)$. A matroid is *connected* if it has no separations.

OUTPUT: boolean

See also

M.components(), *M.is_3connected()*

EXAMPLES:

```
sage: M = Matroid(ring=QQ, matrix=[[1, 0, 0, 1, 1, 0],
....:                              [0, 1, 0, 1, 2, 0],
....:                              [0, 0, 1, 0, 0, 1]])
sage: M.is_connected()
False
sage: matroids.catalog.Pappus().is_connected()
True
```

```
>>> from sage.all import *
>>> M = Matroid(ring=QQ, matrix=[[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(0)],
...                           [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(0)],
...                           [Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)]])
>>> M.is_connected()
False
>>> matroids.catalog.Pappus().is_connected()
True
```

is_cosimple()

Test if the matroid is cosimple.

A matroid is *cosimple* if it contains no cocircuits of length 1 or 2.

Dual method of *M.is_simple()*.

OUTPUT: boolean

See also

M.is_simple(), *M.coloops()*, *M.cocircuit()*, *M.cosimplify()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano().dual()
sage: M.is_cosimple()
True
sage: N = M.delete('a')
```

(continues on next page)

(continued from previous page)

```
sage: N.is_cosimple()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano().dual()
>>> M.is_cosimple()
True
>>> N = M.delete('a')
>>> N.is_cosimple()
False
```

is_dependent(X)

Check if a subset `X` is dependent in the matroid.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: boolean

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_dependent('abc')
False
sage: M.is_dependent('abcd')
True
sage: M.is_dependent('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_dependent('abc')
False
>>> M.is_dependent('abcd')
True
>>> M.is_dependent('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_graphic()

Return if `self` is graphic.

A matroid is graphic if and only if it has no minor isomorphic to any of the matroids $U_{2,4}$, F_7, F_7^*, $M^*(K_5)$, and $M^*(K_{3,3})$.

EXAMPLES:

```
sage: M = matroids.catalog.Wheel4()
sage: M.is_graphic()
True
```

(continues on next page)

(continued from previous page)

```
sage: M = matroids.catalog.U24()
sage: M.is_graphic()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Wheel4()
>>> M.is_graphic()
True
>>> M = matroids.catalog.U24()
>>> M.is_graphic()
False
```

REFERENCES:

[Oxl2011], p. 385.

is_independent(*X*)

Check if a subset `X` is independent in the matroid.

INPUT:

- `X` – a subset (or any iterable) of the groundset

OUTPUT: boolean

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_independent('abc')
True
sage: M.is_independent('abcd')
False
sage: M.is_independent('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_independent('abc')
True
>>> M.is_independent('abcd')
False
>>> M.is_independent('abcx')
Traceback (most recent call last):
...
ValueError: 'abcx' is not a subset of the groundset
```

is_isomorphic(*other*, *certificate=False*)

Test matroid isomorphism.

Two matroids M and N are *isomorphic* if there is a bijection f from the groundset of M to the groundset of N such that a subset X is independent in M if and only if $f(X)$ is independent in N.

INPUT:

- `other` – matroid
- `certificate` – boolean (default: `False`)

OUTPUT: boolean, and, if `certificate=True`, a dictionary or `None`

EXAMPLES:

```
sage: M1 = matroids.Wheel(3)
sage: M2 = matroids.CompleteGraphic(4)                                  #␣
↪needs sage.graphs
sage: M1.is_isomorphic(M2)                                              #␣
↪needs sage.graphs
True
sage: M1.is_isomorphic(M2, certificate=True)                            #␣
↪needs sage.graphs
(True, {0: 0, 1: 1, 2: 2, 3: 3, 4: 5, 5: 4})
sage: G3 = graphs.CompleteGraph(4)                                      #␣
↪needs sage.graphs
sage: M1.is_isomorphic(G3)                                              #␣
↪needs sage.graphs
Traceback (most recent call last):
...
TypeError: can only test for isomorphism between matroids.


sage: M1 = matroids.catalog.Fano()
sage: M2 = matroids.catalog.NonFano()
sage: M1.is_isomorphic(M2)
False
sage: M1.is_isomorphic(M2, certificate=True)
(False, None)
```

```
>>> from sage.all import *
>>> M1 = matroids.Wheel(Integer(3))
>>> M2 = matroids.CompleteGraphic(Integer(4))                                ␣
↪     # needs sage.graphs
>>> M1.is_isomorphic(M2)                                                  #␣
↪needs sage.graphs
True
>>> M1.is_isomorphic(M2, certificate=True)                                #␣
↪needs sage.graphs
(True, {0: 0, 1: 1, 2: 2, 3: 3, 4: 5, 5: 4})
>>> G3 = graphs.CompleteGraph(Integer(4))                                  ␣
↪     # needs sage.graphs
>>> M1.is_isomorphic(G3)                                                  #␣
↪needs sage.graphs
Traceback (most recent call last):
...
TypeError: can only test for isomorphism between matroids.


>>> M1 = matroids.catalog.Fano()
>>> M2 = matroids.catalog.NonFano()
>>> M1.is_isomorphic(M2)
```

(continues on next page)

(continued from previous page)

```
False
>>> M1.is_isomorphic(M2, certificate=True)
(False, None)
```

is_isomorphism(*other*, *morphism*)

Test if a provided morphism induces a matroid isomorphism.

A *morphism* is a map from the groundset of `self` to the groundset of `other`.

INPUT:

- `other` – matroid
- `morphism` – a map; can be, for instance, a dictionary, function, or permutation

OUTPUT: boolean

See also

M.is_isomorphism()

Note

If you know the input is valid, consider using the faster method `self._is_isomorphism`.

EXAMPLES:

```
sage: M = matroids.catalog.Pappus()
sage: N = matroids.catalog.NonPappus()
sage: N.is_isomorphism(M, {e:e for e in M.groundset()})
False

sage: M = matroids.catalog.Fano().delete(['g'])
sage: N = matroids.Wheel(3)
sage: morphism = {'a':0, 'b':1, 'c': 2, 'd':4, 'e':5, 'f':3}
sage: M.is_isomorphism(N, morphism)
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus()
>>> N = matroids.catalog.NonPappus()
>>> N.is_isomorphism(M, {e:e for e in M.groundset()})
False

>>> M = matroids.catalog.Fano().delete(['g'])
>>> N = matroids.Wheel(Integer(3))
>>> morphism = {'a':Integer(0), 'b':Integer(1), 'c': Integer(2), 'd
↪':Integer(4), 'e':Integer(5), 'f':Integer(3)}
>>> M.is_isomorphism(N, morphism)
True
```

A morphism can be specified as a dictionary (above), a permutation, a function, and many other types of maps:

```
sage: M = matroids.catalog.Fano()
sage: P = PermutationGroup([[('a', 'b', 'c'),                                #␣
↪needs sage.rings.finite_rings
....:                         ('d', 'e', 'f'), ('g')]]).gen()
sage: M.is_isomorphism(M, P)                                                 #␣
↪needs sage.rings.finite_rings
True

sage: M = matroids.catalog.Pappus()
sage: N = matroids.catalog.NonPappus()
sage: def f(x):
....:     return x
....:
sage: N.is_isomorphism(M, f)
False
sage: N.is_isomorphism(N, f)
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> P = PermutationGroup([[('a', 'b', 'c'),                                  #␣
↪needs sage.rings.finite_rings
...                       ('d', 'e', 'f'), ('g')]]).gen()
>>> M.is_isomorphism(M, P)                                                   #␣
↪needs sage.rings.finite_rings
True

>>> M = matroids.catalog.Pappus()
>>> N = matroids.catalog.NonPappus()
>>> def f(x):
...     return x
....:
>>> N.is_isomorphism(M, f)
False
>>> N.is_isomorphism(N, f)
True
```

There is extensive checking for inappropriate input:

```
sage: # needs sage.graphs
sage: M = matroids.CompleteGraphic(4)
sage: M.is_isomorphism(graphs.CompleteGraph(4), lambda x: x)
Traceback (most recent call last):
...
TypeError: can only test for isomorphism between matroids.

sage: # needs sage.graphs
sage: M = matroids.CompleteGraphic(4)
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
sage: M.is_isomorphism(M, {0: 1, 1: 2, 2: 3})
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: domain of morphism does not contain groundset of this
matroid.

sage: # needs sage.graphs
sage: M = matroids.CompleteGraphic(4)
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
sage: M.is_isomorphism(M, {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 1})
Traceback (most recent call last):
...
ValueError: range of morphism does not contain groundset of other
matroid.

sage: # needs sage.graphs
sage: M = matroids.CompleteGraphic(3)
sage: N = Matroid(bases=['ab', 'ac', 'bc'])
sage: f = [0, 1, 2]
sage: g = {'a': 0, 'b': 1, 'c': 2}
sage: N.is_isomorphism(M, f)
Traceback (most recent call last):
...
ValueError: the morphism argument does not seem to be an
isomorphism.

sage: # needs sage.graphs
sage: N.is_isomorphism(M, g)
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = matroids.CompleteGraphic(Integer(4))
>>> M.is_isomorphism(graphs.CompleteGraph(Integer(4)), lambda x: x)
Traceback (most recent call last):
...
TypeError: can only test for isomorphism between matroids.

>>> # needs sage.graphs
>>> M = matroids.CompleteGraphic(Integer(4))
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
>>> M.is_isomorphism(M, {Integer(0): Integer(1), Integer(1): Integer(2),␣
↪Integer(2): Integer(3)})
Traceback (most recent call last):
...
ValueError: domain of morphism does not contain groundset of this
matroid.

>>> # needs sage.graphs
>>> M = matroids.CompleteGraphic(Integer(4))
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
>>> M.is_isomorphism(M, {Integer(0): Integer(1), Integer(1): Integer(1),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(2): Integer(1), Integer(3): Integer(1), Integer(4): Integer(1),␣
↪Integer(5): Integer(1)})
Traceback (most recent call last):
...
ValueError: range of morphism does not contain groundset of other
matroid.

>>> # needs sage.graphs
>>> M = matroids.CompleteGraphic(Integer(3))
>>> N = Matroid(bases=['ab', 'ac', 'bc'])
>>> f = [Integer(0), Integer(1), Integer(2)]
>>> g = {'a': Integer(0), 'b': Integer(1), 'c': Integer(2)}
>>> N.is_isomorphism(M, f)
Traceback (most recent call last):
...
ValueError: the morphism argument does not seem to be an
isomorphism.

>>> # needs sage.graphs
>>> N.is_isomorphism(M, g)
True
```

is_k_closed(k)

Return if `self` is a `k`-closed matroid.

We say a matroid is k-closed if all k-closed subsets are closed in `M`.

EXAMPLES:

```
sage: # needs sage.combinat
sage: PR = RootSystem(['A',4]).root_lattice().positive_roots()
sage: m = matrix([x.to_vector() for x in PR]).transpose()
sage: M = Matroid(m)
sage: M.is_k_closed(3)
True
sage: M.is_k_closed(4)
True

sage: # needs sage.combinat
sage: PR = RootSystem(['D',4]).root_lattice().positive_roots()
sage: m = matrix([x.to_vector() for x in PR]).transpose()
sage: M = Matroid(m)
sage: M.is_k_closed(3)
False
sage: M.is_k_closed(4)
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> PR = RootSystem(['A',Integer(4)]).root_lattice().positive_roots()
>>> m = matrix([x.to_vector() for x in PR]).transpose()
>>> M = Matroid(m)
>>> M.is_k_closed(Integer(3))
```

(continues on next page)

(continued from previous page)

```
True
>>> M.is_k_closed(Integer(4))
True

>>> # needs sage.combinat
>>> PR = RootSystem(['D',Integer(4)]).root_lattice().positive_roots()
>>> m = matrix([x.to_vector() for x in PR]).transpose()
>>> M = Matroid(m)
>>> M.is_k_closed(Integer(3))
False
>>> M.is_k_closed(Integer(4))
True
```

is_kconnected(*k*, *certificate=False*)

Return `True` if the matroid is k-connected, `False` otherwise. It can optionally return a separator as a witness.

INPUT:

- `k` – integer greater or equal to 1
- `certificate` – boolean (default: `False`); if `True`, then return `True, None` if the matroid is k-connected, and `False, X` otherwise, where `X` is a $< k$-separation

OUTPUT: boolean or tuple `(boolean, frozenset)`

See also

M.is_connected() M.is_3connected() M.is_4connected()

ALGORITHM:

Apply linking algorithm to find small separator.

EXAMPLES:

```
sage: matroids.Uniform(2, 3).is_kconnected(3)
True
sage: M = Matroid(ring=QQ, matrix=[[1, 0, 0, 1, 1, 0],
....:                              [0, 1, 0, 1, 2, 0],
....:                              [0, 0, 1, 0, 0, 1]])
sage: M.is_kconnected(3)
False
sage: N = Matroid(circuit_closures={2: ['abc', 'cdef'],
....:                               3: ['abcdef']},
....:             groundset='abcdef')
sage: N.is_kconnected(3)
False
sage: matroids.catalog.BetsyRoss().is_kconnected(3)                    #␣
↪needs sage.graphs
True
sage: matroids.AG(5,2).is_kconnected(4)
True
sage: M = matroids.catalog.R6()
sage: M.is_kconnected(3)                                               #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
False
sage: B, X = M.is_kconnected(3,True)
sage: M.connectivity(X)<3
True
```

```
>>> from sage.all import *
>>> matroids.Uniform(Integer(2), Integer(3)).is_kconnected(Integer(3))
True
>>> M = Matroid(ring=QQ, matrix=[[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(0)],
...                           [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(0)],
...                           [Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)]])
>>> M.is_kconnected(Integer(3))
False
>>> N = Matroid(circuit_closures={Integer(2): ['abc', 'cdef'],
...                               Integer(3): ['abcdef']},
...             groundset='abcdef')
>>> N.is_kconnected(Integer(3))
False
>>> matroids.catalog.BetsyRoss().is_kconnected(Integer(3))                    ␣
↪     # needs sage.graphs
True
>>> matroids.AG(Integer(5),Integer(2)).is_kconnected(Integer(4))
True
>>> M = matroids.catalog.R6()
>>> M.is_kconnected(Integer(3))                                               ␣
↪     # needs sage.graphs
False
>>> B, X = M.is_kconnected(Integer(3),True)
>>> M.connectivity(X)<Integer(3)
True
```

is_max_weight_coindependent_generic(*X=None*, *weights=None*)

Test if only one cobasis of the subset `X` has maximal weight.

The *weight* of a subset `S` is `sum(weights(e) for e in S)`.

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset
- `weights` – dictionary or function mapping the elements of `X` to nonnegative weights

OUTPUT: boolean

ALGORITHM:

The greedy algorithm. If a weight function is given, then sort the elements of `X` by increasing weight, and otherwise use the ordering in which `X` lists its elements. Then greedily select elements if they are coindependent of all that was selected before. If an element is not coindependent of the previously selected elements, then we check if it is coindependent with the previously selected elements with higher weight.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Fano()
sage: M.is_max_weight_coindependent_generic()
False

sage: def wt(x):
....:   return x
....:
sage: M = matroids.Uniform(2, 8)
sage: M.is_max_weight_coindependent_generic(weights=wt)
True
sage: M.is_max_weight_coindependent_generic(weights={x: x for x in M.
↪groundset()})
True
sage: M.is_max_weight_coindependent_generic()
False

sage: M = matroids.Uniform(2, 5)
sage: wt = {0: 1, 1: 1, 2: 1, 3: 2, 4: 2}
sage: M.is_max_weight_independent_generic(weights=wt)
True
sage: M.dual().is_max_weight_coindependent_generic(weights=wt)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Fano()
>>> M.is_max_weight_coindependent_generic()
False

>>> def wt(x):
...   return x
....:
>>> M = matroids.Uniform(Integer(2), Integer(8))
>>> M.is_max_weight_coindependent_generic(weights=wt)
True
>>> M.is_max_weight_coindependent_generic(weights={x: x for x in M.
↪groundset()})
True
>>> M.is_max_weight_coindependent_generic()
False

>>> M = matroids.Uniform(Integer(2), Integer(5))
>>> wt = {Integer(0): Integer(1), Integer(1): Integer(1), Integer(2):␣
↪Integer(1), Integer(3): Integer(2), Integer(4): Integer(2)}
>>> M.is_max_weight_independent_generic(weights=wt)
True
>>> M.dual().is_max_weight_coindependent_generic(weights=wt)
True
```

Here is an example from [GriRei18] (Example 7.4.12 in v6):

```
sage: A = Matrix(QQ, [[ 1,  1,  0,  0],
....:                 [-1,  0,  1,  1],
....:                 [ 0, -1, -1, -1]])
sage: M = Matroid(A)
sage: M.is_max_weight_coindependent_generic()
False
sage: M.is_max_weight_coindependent_generic(weights={0: 1, 1: 3, 2: 3, 3: 2})
True
sage: M.is_max_weight_coindependent_generic(weights={0: 1, 1: 3, 2: 2, 3: 2})
False
sage: M.is_max_weight_coindependent_generic(weights={0: 2, 1: 3, 2: 1, 3: 1})
False

sage: M.is_max_weight_coindependent_generic(weights={0: 2, 1: 3, 2: -1, 3: 1})
Traceback (most recent call last):
...
ValueError: nonnegative weights were expected.
```

```
>>> from sage.all import *
>>> A = Matrix(QQ, [[ Integer(1),  Integer(1),  Integer(0),  Integer(0)],
...                 [-Integer(1),  Integer(0),  Integer(1),  Integer(1)],
...                 [ Integer(0), -Integer(1), -Integer(1), -Integer(1)]])
>>> M = Matroid(A)
>>> M.is_max_weight_coindependent_generic()
False
>>> M.is_max_weight_coindependent_generic(weights={Integer(0): Integer(1),␣
↪Integer(1): Integer(3), Integer(2): Integer(3), Integer(3): Integer(2)})
True
>>> M.is_max_weight_coindependent_generic(weights={Integer(0): Integer(1),␣
↪Integer(1): Integer(3), Integer(2): Integer(2), Integer(3): Integer(2)})
False
>>> M.is_max_weight_coindependent_generic(weights={Integer(0): Integer(2),␣
↪Integer(1): Integer(3), Integer(2): Integer(1), Integer(3): Integer(1)})
False

>>> M.is_max_weight_coindependent_generic(weights={Integer(0): Integer(2),␣
↪Integer(1): Integer(3), Integer(2): -Integer(1), Integer(3): Integer(1)})
Traceback (most recent call last):
...
ValueError: nonnegative weights were expected.
```

is_max_weight_independent_generic(*X=None*, *weights=None*)

Test if only one basis of the subset `X` has maximal weight.

The *weight* of a subset `S` is `sum(weights(e) for e in S)`.

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset
- `weights` – dictionary or function mapping the elements of `X` to nonnegative weights

OUTPUT: boolean

ALGORITHM:

The greedy algorithm. If a weight function is given, then sort the elements of X by decreasing weight, and otherwise use the ordering in which X lists its elements. Then greedily select elements if they are independent of all that was selected before. If an element is not independent of the previously selected elements, then we check if it is independent with the previously selected elements with higher weight.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Fano()
sage: M.is_max_weight_independent_generic()
False

sage: def wt(x):
....:   return x
....:
sage: M = matroids.Uniform(2, 8)
sage: M.is_max_weight_independent_generic(weights=wt)
True
sage: M.is_max_weight_independent_generic(weights={x: x for x in M.
↪groundset()})
True
sage: M.is_max_weight_independent_generic()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Fano()
>>> M.is_max_weight_independent_generic()
False

>>> def wt(x):
...   return x
....:
>>> M = matroids.Uniform(Integer(2), Integer(8))
>>> M.is_max_weight_independent_generic(weights=wt)
True
>>> M.is_max_weight_independent_generic(weights={x: x for x in M.groundset()})
True
>>> M.is_max_weight_independent_generic()
False
```

Here is an example from [GriRei18] (Example 7.4.12 in v6):

```
sage: A = Matrix(QQ, [[ 1,  1,  0,  0],
....:                 [-1,  0,  1,  1],
....:                 [ 0, -1, -1, -1]])
sage: M = Matroid(A)
sage: M.is_max_weight_independent_generic()
False
sage: M.is_max_weight_independent_generic(weights={0: 1, 1: 3, 2: 3, 3: 2})
True
sage: M.is_max_weight_independent_generic(weights={0: 1, 1: 3, 2: 2, 3: 2})
False
sage: M.is_max_weight_independent_generic(weights={0: 2, 1: 3, 2: 1, 3: 1})
```

(continues on next page)

(continued from previous page)

```
True

sage: M.is_max_weight_independent_generic(weights={0: 2, 1: 3, 2: -1, 3: 1})
Traceback (most recent call last):
...
ValueError: nonnegative weights were expected.
```

```
>>> from sage.all import *
>>> A = Matrix(QQ, [[ Integer(1),  Integer(1),  Integer(0),  Integer(0)],
...                 [-Integer(1),  Integer(0),  Integer(1),  Integer(1)],
...                 [ Integer(0), -Integer(1), -Integer(1), -Integer(1)]])
>>> M = Matroid(A)
>>> M.is_max_weight_independent_generic()
False
>>> M.is_max_weight_independent_generic(weights={Integer(0): Integer(1),␣
↪Integer(1): Integer(3), Integer(2): Integer(3), Integer(3): Integer(2)})
True
>>> M.is_max_weight_independent_generic(weights={Integer(0): Integer(1),␣
↪Integer(1): Integer(3), Integer(2): Integer(2), Integer(3): Integer(2)})
False
>>> M.is_max_weight_independent_generic(weights={Integer(0): Integer(2),␣
↪Integer(1): Integer(3), Integer(2): Integer(1), Integer(3): Integer(1)})
True

>>> M.is_max_weight_independent_generic(weights={Integer(0): Integer(2),␣
↪Integer(1): Integer(3), Integer(2): -Integer(1), Integer(3): Integer(1)})
Traceback (most recent call last):
...
ValueError: nonnegative weights were expected.
```

is_paving()

Return if `self` is paving.

A matroid is paving if each of its circuits has size r or $r + 1$.

OUTPUT: boolean

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_paving()
True
sage: M = matroids.Theta(4)
sage: M.is_paving()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_paving()
True
>>> M = matroids.Theta(Integer(4))
>>> M.is_paving()
False
```

REFERENCES:

[Oxl2011], p. 24.

is_regular()

Return if `self` is regular.

A regular matroid is one that can be represented by a totally unimodular matrix, the latter being a matrix over $\mathbb{R}$ for which every square submatrix has determinant in $\{0, 1, -1\}$. A matroid is regular if and only if it is representable over every field. Alternatively, a matroid is regular if and only if it has no minor isomorphic to $U_{2,4}$, F_7, or F_7^*.

EXAMPLES:

```
sage: M = matroids.catalog.Wheel4()
sage: M.is_regular()
True
sage: M = matroids.catalog.R9()
sage: M.is_regular()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Wheel4()
>>> M.is_regular()
True
>>> M = matroids.catalog.R9()
>>> M.is_regular()
False
```

REFERENCES:

[Oxl2011], p. 373.

is_simple()

Test if the matroid is simple.

A matroid is *simple* if it contains no circuits of length 1 or 2.

OUTPUT: boolean

See also

M.is_cosimple(), *M.loops()*, *M.circuit()*, *M.simplify()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.is_simple()
True
sage: N = M / 'a'
sage: N.is_simple()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.is_simple()
```

(continues on next page)

(continued from previous page)

```
True
>>> N = M / 'a'
>>> N.is_simple()
False
```

is_sparse_paving()

Return if `self` is sparse-paving.

A matroid is sparse-paving if it is paving and its dual is paving.

OUTPUT: boolean

ALGORITHM:

First, check that the matroid is paving. Then, verify that the symmetric difference of every pair of distinct r-circuits is greater than 2.

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_sparse_paving()
True
sage: M = matroids.catalog.N1()
sage: M.is_sparse_paving()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_sparse_paving()
True
>>> M = matroids.catalog.N1()
>>> M.is_sparse_paving()
False
```

REFERENCES:

The definition of sparse-paving matroids can be found in [MNWW2011]. The algorithm uses an alternative characterization from [Jer2006].

is_subset_k_closed(X, k)

Test if `X` is a `k`-closed set of the matroid.

A set S is *k-closed* if the closure of any k element subsets is contained in S.

INPUT:

- `X` – a subset (or any iterable) of the groundset
- `k` – positive integer

OUTPUT: boolean

See also

M.k_closure()

EXAMPLES:

```
sage: m = matrix([[1,2,5,2], [0,2,1,0]])
sage: M = Matroid(m)
sage: M.is_subset_k_closed({1,3}, 2)
False
sage: M.is_subset_k_closed({0,1}, 1)
False
sage: M.is_subset_k_closed({1,2}, 1)
True

sage: m = matrix([[1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1],
....:             [0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 2],
....:             [0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1],
....:             [0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1]])
sage: M = Matroid(m)
sage: M.is_subset_k_closed({0,2,3,11}, 3)
True
sage: M.is_subset_k_closed({0,2,3,11}, 4)
False
sage: M.is_subset_k_closed({0,1}, 4)
False
sage: M.is_subset_k_closed({0,1,4}, 4)
True
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(1),Integer(2),Integer(5),Integer(2)], [Integer(0),
↪Integer(2),Integer(1),Integer(0)]])
>>> M = Matroid(m)
>>> M.is_subset_k_closed({Integer(1),Integer(3)}, Integer(2))
False
>>> M.is_subset_k_closed({Integer(0),Integer(1)}, Integer(1))
False
>>> M.is_subset_k_closed({Integer(1),Integer(2)}, Integer(1))
True

>>> m = matrix([[Integer(1), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(1)],
...             [Integer(0), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(2)],
...             [Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(1)],
...             [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1)]])
>>> M = Matroid(m)
>>> M.is_subset_k_closed({Integer(0),Integer(2),Integer(3),Integer(11)},␣
↪Integer(3))
True
>>> M.is_subset_k_closed({Integer(0),Integer(2),Integer(3),Integer(11)},␣
↪Integer(4))
```

(continues on next page)

(continued from previous page)

```
False
>>> M.is_subset_k_closed({Integer(0),Integer(1)}, Integer(4))
False
>>> M.is_subset_k_closed({Integer(0),Integer(1),Integer(4)}, Integer(4))
True
```

is_ternary(*randomized_tests=1*)

Decide if `self` is a ternary matroid.

INPUT:

- `randomized_tests` – (default: 1) an integer; the number of times a certain necessary condition for being ternary is tested, using randomization

OUTPUT: boolean

ALGORITHM:

First, compare the ternary matroids local to two random bases. If these matroids are not isomorphic, return `False`. This test is performed `randomized_tests` times. Next, test if a ternary matroid local to some basis is isomorphic to `self`.

See also

M.ternary_matroid()

EXAMPLES:

```
sage: N = matroids.catalog.Fano()
sage: N.is_ternary()                                                    #␣
↪needs sage.graphs
False
sage: N = matroids.catalog.NonFano()
sage: N.is_ternary()                                                    #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> N = matroids.catalog.Fano()
>>> N.is_ternary()                                                      #␣
↪needs sage.graphs
False
>>> N = matroids.catalog.NonFano()
>>> N.is_ternary()                                                      #␣
↪needs sage.graphs
True
```

is_valid(*certificate=False*)

Test if the data obey the matroid axioms.

The default implementation checks the (disproportionately slow) rank axioms. If r is the rank function of a matroid, we check, for all pairs X, Y of subsets,

- $0 \leq r(X) \leq |X|$

- If $X \subseteq Y$ then $r(X) \leq r(Y)$
- $r(X \cup Y) + r(X \cap Y) \leq r(X) + r(Y)$

Certain subclasses may check other axioms instead.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: boolean, or (boolean, dictionary)

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.is_valid()
True
```

The following is the 'Escher matroid' by Brylawski and Kelly. See Example 1.5.5 in [Oxl2011]

```
sage: M = Matroid(circuit_closures={2: [[1, 2, 3], [1, 4, 5]],
....: 3: [[1, 2, 3, 4, 5], [1, 2, 3, 6, 7], [1, 4, 5, 6, 7]]})
sage: M.is_valid()
False
```

```
>>> from sage.all import *
>>> M = Matroid(circuit_closures={Integer(2): [[Integer(1), Integer(2),␣
↪Integer(3)], [Integer(1), Integer(4), Integer(5)]],
... Integer(3): [[Integer(1), Integer(2), Integer(3), Integer(4), Integer(5)],
↪ [Integer(1), Integer(2), Integer(3), Integer(6), Integer(7)], [Integer(1),␣
↪Integer(4), Integer(5), Integer(6), Integer(7)]]})
>>> M.is_valid()
False
```

isomorphism(*other*)

Return a matroid isomorphism.

Two matroids M and N are *isomorphic* if there is a bijection f from the groundset of M to the groundset of N such that a subset X is independent in M if and only if $f(X)$ is independent in N. This method returns one isomorphism f from `self` to `other`, if such an isomorphism exists.

INPUT:

- `other` – matroid

OUTPUT: dictionary or `None`

EXAMPLES:

```
sage: M1 = matroids.Wheel(3)
sage: M2 = matroids.CompleteGraphic(4)                                    #␣
↪needs sage.graphs
sage: morphism = M1.isomorphism(M2)                                       #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
sage: M1.is_isomorphism(M2, morphism)                                    #␣
↪needs sage.graphs
True
sage: G3 = graphs.CompleteGraph(4)                                       #␣
↪needs sage.graphs
sage: M1.isomorphism(G3)                                                 #␣
↪needs sage.graphs
Traceback (most recent call last):
...
TypeError: can only give isomorphism between matroids.

sage: M1 = matroids.catalog.Fano()
sage: M2 = matroids.catalog.NonFano()
sage: M1.isomorphism(M2) is not None
False
```

```
>>> from sage.all import *
>>> M1 = matroids.Wheel(Integer(3))
>>> M2 = matroids.CompleteGraphic(Integer(4))                            ␣
↪     # needs sage.graphs
>>> morphism = M1.isomorphism(M2)                                        #␣
↪needs sage.graphs
>>> M1.is_isomorphism(M2, morphism)                                      #␣
↪needs sage.graphs
True
>>> G3 = graphs.CompleteGraph(Integer(4))                                ␣
↪     # needs sage.graphs
>>> M1.isomorphism(G3)                                                   #␣
↪needs sage.graphs
Traceback (most recent call last):
...
TypeError: can only give isomorphism between matroids.

>>> M1 = matroids.catalog.Fano()
>>> M2 = matroids.catalog.NonFano()
>>> M1.isomorphism(M2) is not None
False
```

k_closure(X, k)

Return the `k`-closure of `X`.

A subset S of the groundset is *k-closed* if the closure of any subset T of S satisfying $|T| \leq k$ is contained in S. The *k-closure* of a set X is the smallest k-closed set containing X.

INPUT:

- `X` – a subset (or any iterable) of the groundset
- `k` – positive integer

EXAMPLES:

```
sage: m = matrix([[1,2,5,2], [0,2,1,0]])
sage: M = Matroid(m)
sage: sorted(M.k_closure({1,3}, 2))
[0, 1, 2, 3]
sage: sorted(M.k_closure({0,1}, 1))
[0, 1, 3]
sage: sorted(M.k_closure({1,2}, 1))
[1, 2]

sage: m = matrix([[1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1],
....:             [0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 2],
....:             [0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1],
....:             [0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1]])
sage: M = Matroid(m)
sage: sorted(M.k_closure({0,2,3,11}, 3))
[0, 2, 3, 11]
sage: sorted(M.k_closure({0,2,3,11}, 4))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
sage: sorted(M.k_closure({0,1}, 4))
[0, 1, 4]
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(1),Integer(2),Integer(5),Integer(2)], [Integer(0),
↪Integer(2),Integer(1),Integer(0)]])
>>> M = Matroid(m)
>>> sorted(M.k_closure({Integer(1),Integer(3)}, Integer(2)))
[0, 1, 2, 3]
>>> sorted(M.k_closure({Integer(0),Integer(1)}, Integer(1)))
[0, 1, 3]
>>> sorted(M.k_closure({Integer(1),Integer(2)}, Integer(1)))
[1, 2]

>>> m = matrix([[Integer(1), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(0), Integer(1), Integer(1), Integer(1),␣
↪Integer(1)],
...             [Integer(0), Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(2)],
...             [Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(1), Integer(0), Integer(1),␣
↪Integer(1)],
...             [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1)]])
>>> M = Matroid(m)
>>> sorted(M.k_closure({Integer(0),Integer(2),Integer(3),Integer(11)},␣
↪Integer(3)))
[0, 2, 3, 11]
>>> sorted(M.k_closure({Integer(0),Integer(2),Integer(3),Integer(11)},␣
↪Integer(4)))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> sorted(M.k_closure({Integer(0),Integer(1)}, Integer(4)))
[0, 1, 4]
```

lattice_of_flats()

Return the lattice of flats of the matroid.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.lattice_of_flats()                                                  #␣
↪needs sage.graphs
Finite lattice containing 16 elements
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.lattice_of_flats()                                                  #␣
↪needs sage.graphs
Finite lattice containing 16 elements
```

linear_subclasses(*line_length=None*, *subsets=None*)

Return an iterable set of linear subclasses of the matroid.

A *linear subclass* is a set of hyperplanes (i.e. closed sets of rank $r(M) - 1$) with the following property:

- If H_1 and H_2 are members, and $r(H_1 \cap H_2) = r(M) - 2$, then any hyperplane H_3 containing $H_1 \cap H_2$ is a member too.

A linear subclass is the set of hyperplanes of a *modular cut* and uniquely determines the modular cut. Hence the collection of linear subclasses is in 1-to-1 correspondence with the collection of single-element extensions of a matroid. See [Oxl2011], section 7.2.

INPUT:

- `line_length` – (default: `None`) a natural number. If given, restricts the output to modular cuts that generate an extension by e that does not contain a minor N isomorphic to $U_{2,k}$, where `k > line_length`, and such that $e \in E(N)$.
- `subsets` – (default: `None`) a collection of subsets of the groundset. If given, restricts the output to linear subclasses such that each hyperplane contains an element of `subsets`.

OUTPUT: an iterable collection of linear subclasses

Note

The `line_length` argument only checks for lines using the new element of the corresponding extension. It is still possible that a long line exists by contracting the new element!

See also

M.flats(), *M.modular_cut()*, *M.extension()*, *sage.matroids.extension*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: len(list(M.linear_subclasses()))
16
sage: len(list(M.linear_subclasses(line_length=3)))
```

(continues on next page)

(continued from previous page)

```
8
sage: len(list(M.linear_subclasses(subsets=[{'a', 'b'}])))
5
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> len(list(M.linear_subclasses()))
16
>>> len(list(M.linear_subclasses(line_length=Integer(3))))
8
>>> len(list(M.linear_subclasses(subsets=[{'a', 'b'}])))
5
```

The following matroid has an extension by element e such that contracting e creates a 6-point line, but no 6-point line minor uses e. Consequently, this method returns the modular cut, but the *M.extensions()* method doesn't return the corresponding extension:

```
sage: M = Matroid(circuit_closures={2: ['abc', 'def'],
....:                               3: ['abcdef']})
sage: len(list(M.extensions('g', line_length=5)))
43
sage: len(list(M.linear_subclasses(line_length=5)))
44
```

```
>>> from sage.all import *
>>> M = Matroid(circuit_closures={Integer(2): ['abc', 'def'],
...                               Integer(3): ['abcdef']})
>>> len(list(M.extensions('g', line_length=Integer(5))))
43
>>> len(list(M.linear_subclasses(line_length=Integer(5))))
44
```

link(S, T)

Given disjoint subsets S and T, return a connector I and a separation X, which are optimal dual solutions in Tutte's Linking Theorem:

$$\max\{r_N(S) + r_N(T) - r(N) \mid N = M/I \setminus J, E(N) = S \cup T\} =$$
$$\min\{r_M(X) + r_M(Y) - r_M(E) \mid X \subseteq S, Y \subseteq T, E = X \cup Y, X \cap Y = \emptyset\}.$$

Here M denotes this matroid.

INPUT:

- `S` – a subset (or any iterable) of the groundset
- `T` – a subset (or any iterable) of the groundset disjoint from `S`

OUTPUT: a tuple `(I, X)` containing a frozenset `I` and a frozenset `X`

ALGORITHM:

Compute a maximum-cardinality common independent set I of of $M/S \setminus T$ and $M \setminus S/T$.

EXAMPLES:

```
sage: M = matroids.catalog.BetsyRoss()
sage: S = set('ab')
sage: T = set('cd')
sage: I, X = M.link(S, T)
sage: M.connectivity(X)
2
sage: J = M.groundset()-(S|T|I)
sage: N = (M/I).delete(J)
sage: N.connectivity(S)
2
```

```
>>> from sage.all import *
>>> M = matroids.catalog.BetsyRoss()
>>> S = set('ab')
>>> T = set('cd')
>>> I, X = M.link(S, T)
>>> M.connectivity(X)
2
>>> J = M.groundset()-(S|T|I)
>>> N = (M/I).delete(J)
>>> N.connectivity(S)
2
```

loops()

Return the set of loops of the matroid.

A *loop* is an element u of the groundset such that the one-element set $\{u\}$ is dependent.

OUTPUT: a set of elements

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.loops()
frozenset()
sage: (M / ['a', 'b']).loops()
frozenset({'f'})
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.loops()
frozenset()
>>> (M / ['a', 'b']).loops()
frozenset({'f'})
```

matroid_polytope()

Return the matroid polytope of `self`.

This is defined as the convex hull of the vertices

$$e_B = \sum_{i \in B} e_i$$

over all bases B of the matroid. Here e_i are the standard basis vectors of $\mathbf{R}^n$. An arbitrary labelling of the groundset by $\{0, \ldots, n-1\}$ is chosen.

See also

independence_matroid_polytope()

EXAMPLES:

```
sage: M = matroids.Whirl(4)
sage: P = M.matroid_polytope(); P                                              #␣
↪needs sage.geometry.polyhedron sage.rings.finite_rings
A 7-dimensional polyhedron in ZZ^8 defined as the convex hull
of 46 vertices

sage: M = matroids.catalog.NonFano()
sage: M.matroid_polytope()                                                     #␣
↪needs sage.geometry.polyhedron sage.rings.finite_rings
A 6-dimensional polyhedron in ZZ^7 defined as the convex hull
of 29 vertices
```

```
>>> from sage.all import *
>>> M = matroids.Whirl(Integer(4))
>>> P = M.matroid_polytope(); P                                                #␣
↪needs sage.geometry.polyhedron sage.rings.finite_rings
A 7-dimensional polyhedron in ZZ^8 defined as the convex hull
of 46 vertices

>>> M = matroids.catalog.NonFano()
>>> M.matroid_polytope()                                                       #␣
↪needs sage.geometry.polyhedron sage.rings.finite_rings
A 6-dimensional polyhedron in ZZ^7 defined as the convex hull
of 29 vertices
```

REFERENCES:

[DLHK2007]

max_coindependent(*X*)

Compute a maximal coindependent subset of X.

A set is *coindependent* if it is independent in the dual matroid. A set is coindependent if and only if the complement is *spanning* (i.e. contains a basis of the matroid).

INPUT:

- X – a subset (or any iterable) of the groundset

OUTPUT: a subset of X

See also

M.dual(), *M.max_independent()*

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: X = M.max_coindependent(['a', 'c', 'd', 'e', 'f'])
sage: sorted(X)  # random
['a', 'c', 'd', 'f']
sage: M.is_coindependent(X)
True
sage: all(M.is_codependent(X.union([y])) for y in M.groundset() if y not in X)
True
sage: M.max_coindependent(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> X = M.max_coindependent(['a', 'c', 'd', 'e', 'f'])
>>> sorted(X)  # random
['a', 'c', 'd', 'f']
>>> M.is_coindependent(X)
True
>>> all(M.is_codependent(X.union([y])) for y in M.groundset() if y not in X)
True
>>> M.max_coindependent(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

max_independent(*X*)

Compute a maximal independent subset of X.

INPUT:

- X – a subset (or any iterable) of the groundset

OUTPUT: subset of X

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: X = M.max_independent(['a', 'c', 'd', 'e', 'f'])
sage: M.is_independent(X)
True
sage: all(M.is_dependent(X.union([y])) for y in M.groundset() if y not in X)
True
sage: M.max_independent(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> X = M.max_independent(['a', 'c', 'd', 'e', 'f'])
>>> M.is_independent(X)
True
```

(continues on next page)

(continued from previous page)

```
>>> all(M.is_dependent(X.union([y])) for y in M.groundset() if y not in X)
True
>>> M.max_independent(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

max_weight_coindependent(*X=None*, *weights=None*)

Return a maximum-weight coindependent set contained in `X`.

The *weight* of a subset `S` is `sum(weights(e) for e in S)`.

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset
- `weights` – dictionary or function mapping the elements of `X` to nonnegative weights

OUTPUT: a subset of `X`

ALGORITHM:

The greedy algorithm. If a weight function is given, then sort the elements of `X` by decreasing weight, and otherwise use the ordering in which `X` lists its elements. Then greedily select elements if they are coindependent of all that was selected before.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Fano()
sage: X = M.max_weight_coindependent()
sage: M.is_cobasis(X)
True

sage: wt = {'a': 1, 'b': 2, 'c': 2, 'd': 1/2, 'e': 1, 'f': 2, 'g': 2}
sage: setprint(M.max_weight_coindependent(weights=wt))
{'b', 'c', 'f', 'g'}
sage: wt = {'a': 1, 'b': -10, 'c': 2, 'd': 1/2, 'e': 1, 'f': 2, 'g': 2}
sage: setprint(M.max_weight_coindependent(weights=wt))
Traceback (most recent call last):
...
ValueError: nonnegative weights were expected.

sage: def wt(x):
....:   return x
....:
sage: M = matroids.Uniform(2, 8)
sage: setprint(M.max_weight_coindependent(weights=wt))
{2, 3, 4, 5, 6, 7}
sage: setprint(M.max_weight_coindependent())
{0, 1, 2, 3, 4, 5}
sage: M.max_weight_coindependent(X=[], weights={})
frozenset()
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
```

(continues on next page)

(continued from previous page)

```
>>> M = matroids.catalog.Fano()
>>> X = M.max_weight_coindependent()
>>> M.is_cobasis(X)
True

>>> wt = {'a': Integer(1), 'b': Integer(2), 'c': Integer(2), 'd': Integer(1)/
↪Integer(2), 'e': Integer(1), 'f': Integer(2), 'g': Integer(2)}
>>> setprint(M.max_weight_coindependent(weights=wt))
{'b', 'c', 'f', 'g'}
>>> wt = {'a': Integer(1), 'b': -Integer(10), 'c': Integer(2), 'd':␣
↪Integer(1)/Integer(2), 'e': Integer(1), 'f': Integer(2), 'g': Integer(2)}
>>> setprint(M.max_weight_coindependent(weights=wt))
Traceback (most recent call last):
...
ValueError: nonnegative weights were expected.

>>> def wt(x):
...   return x
....:
>>> M = matroids.Uniform(Integer(2), Integer(8))
>>> setprint(M.max_weight_coindependent(weights=wt))
{2, 3, 4, 5, 6, 7}
>>> setprint(M.max_weight_coindependent())
{0, 1, 2, 3, 4, 5}
>>> M.max_weight_coindependent(X=[], weights={})
frozenset()
```

max_weight_independent(*X=None*, *weights=None*)

Return a maximum-weight independent set contained in a subset.

The *weight* of a subset `S` is `sum(weights(e) for e in S)`.

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset
- `weights` – dictionary or function mapping the elements of `X` to nonnegative weights

OUTPUT: a subset of `X`

ALGORITHM:

The greedy algorithm. If a weight function is given, then sort the elements of `X` by decreasing weight, and otherwise use the ordering in which `X` lists its elements. Then greedily select elements if they are independent of all that was selected before.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Fano()
sage: X = M.max_weight_independent()
sage: M.is_basis(X)
True

sage: wt = {'a': 1, 'b': 2, 'c': 2, 'd': 1/2, 'e': 1,
....:       'f': 2, 'g': 2}
```

(continues on next page)

(continued from previous page)

```
sage: setprint(M.max_weight_independent(weights=wt))
{'b', 'f', 'g'}
sage: def wt(x):
....:   return x
....:
sage: M = matroids.Uniform(2, 8)
sage: setprint(M.max_weight_independent(weights=wt))
{6, 7}
sage: setprint(M.max_weight_independent())
{0, 1}
sage: M.max_weight_coindependent(X=[], weights={})
frozenset()
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Fano()
>>> X = M.max_weight_independent()
>>> M.is_basis(X)
True

>>> wt = {'a': Integer(1), 'b': Integer(2), 'c': Integer(2), 'd': Integer(1)/
↪Integer(2), 'e': Integer(1),
...       'f': Integer(2), 'g': Integer(2)}
>>> setprint(M.max_weight_independent(weights=wt))
{'b', 'f', 'g'}
>>> def wt(x):
...   return x
....:
>>> M = matroids.Uniform(Integer(2), Integer(8))
>>> setprint(M.max_weight_independent(weights=wt))
{6, 7}
>>> setprint(M.max_weight_independent())
{0, 1}
>>> M.max_weight_coindependent(X=[], weights={})
frozenset()
```

minor(*contractions=None*, *deletions=None*)

Return the minor of `self` obtained by contracting, respectively deleting, the element(s) of `contractions` and `deletions`.

A *minor* of a matroid is a matroid obtained by repeatedly removing elements in one of two ways: either *contract* or *delete* them. It can be shown that the final matroid does not depend on the order in which elements are removed.

INPUT:

- `contractions` – (default: `None`) an element or set of elements to be contracted
- `deletions` – (default: `None`) an element or set of elements to be deleted

OUTPUT: matroid

Note

The output is either of the same type as `self`, or an instance of *MinorMatroid*.

See also

M.contract(), *M.delete()*

EXAMPLES:

```
sage: M = matroids.Wheel(4)
sage: N = M.minor(contractions=[7], deletions=[0])
sage: N.is_isomorphic(matroids.Wheel(3))
True
```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(4))
>>> N = M.minor(contractions=[Integer(7)], deletions=[Integer(0)])
>>> N.is_isomorphic(matroids.Wheel(Integer(3)))
True
```

The sets of contractions and deletions need not be independent, respectively coindependent:

```
sage: M = matroids.catalog.Fano()
sage: M.rank('abf')
2
sage: M.minor(contractions='abf')
Binary matroid of rank 1 on 4 elements, type (1, 0)
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.rank('abf')
2
>>> M.minor(contractions='abf')
Binary matroid of rank 1 on 4 elements, type (1, 0)
```

However, they need to be subsets of the groundset, and disjoint:

```
sage: M = matroids.catalog.Vamos()
sage: N = M.minor('abc', 'defg')
sage: N
M / {'a', 'b', 'c'} \ {'d', 'e', 'f', 'g'}, where M is Vamos:
Matroid of rank 4 on 8 elements with circuit-closures
...
sage: N.groundset()
frozenset({'h'})

sage: N = M.minor('defgh', 'abc')
sage: N  # random
M / {'d', 'e', 'f', 'g'} \ {'a', 'b', 'c', 'h'}, where M is Vamos:
Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'},
     {'a', 'b', 'g', 'h'}, {'c', 'd', 'e', 'f'},
```

(continues on next page)

(continued from previous page)

```
     {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: N.groundset()
frozenset()

sage: M.minor([1, 2, 3], 'efg')
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not a subset of the groundset
sage: M.minor('efg', [1, 2, 3])
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not a subset of the groundset
sage: M.minor('ade', 'efg')
Traceback (most recent call last):
...
ValueError: contraction and deletion sets are not disjoint.
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> N = M.minor('abc', 'defg')
>>> N
M / {'a', 'b', 'c'} \ {'d', 'e', 'f', 'g'}, where M is Vamos:
Matroid of rank 4 on 8 elements with circuit-closures
...
>>> N.groundset()
frozenset({'h'})

>>> N = M.minor('defgh', 'abc')
>>> N  # random
M / {'d', 'e', 'f', 'g'} \ {'a', 'b', 'c', 'h'}, where M is Vamos:
Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'},
     {'a', 'b', 'g', 'h'}, {'c', 'd', 'e', 'f'},
     {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
>>> N.groundset()
frozenset()

>>> M.minor([Integer(1), Integer(2), Integer(3)], 'efg')
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not a subset of the groundset
>>> M.minor('efg', [Integer(1), Integer(2), Integer(3)])
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not a subset of the groundset
>>> M.minor('ade', 'efg')
Traceback (most recent call last):
...
ValueError: contraction and deletion sets are not disjoint.
```

Warning

There can be ambiguity if elements of the groundset are themselves iterable, and their elements are in the groundset. The main example of this is when an element is a string. See the documentation of the methods *contract()* and *delete()* for an example of this.

modular_cut(*subsets*)

Compute the modular cut generated by `subsets`.

A *modular cut* is a collection C of flats such that

- If $F \in C$ and F' is a flat containing F, then $F' \in C$
- If $F_1, F_2 \in C$ form a modular pair of flats, then $F_1 \cap F_2 \in C$.

A *flat* is a closed set, a *modular pair* is a pair F_1, F_2 of flats with $r(F_1) + r(F_2) = r(F_1 \cup F_2) + r(F_1 \cap F_2)$, where r is the rank function of the matroid.

The modular cut *generated by* `subsets` is the smallest modular cut C for which closure`(S) in C` for all S in `subsets`.

There is a one-to-one correspondence between the modular cuts of a matroid and the single-element extensions of the matroid. See [Oxl2011] Section 7.2 for more information.

Note

Sage uses linear subclasses, rather than modular cuts, internally for matroid extension. A linear subclass is the set of hyperplanes (flats of rank $r(M) - 1$) of a modular cut. It determines the modular cut uniquely (see [Oxl2011] Section 7.2).

INPUT:

- `subsets` – a collection of subsets of the groundset

OUTPUT: a collection of subsets

See also

M.flats(), *M.linear_subclasses()*, *M.extension()*

EXAMPLES:

Any extension of the Vamos matroid where the new point is placed on the lines through elements $\{a, b\}$ and through $\{c, d\}$ is an extension by a loop:

```
sage: M = matroids.catalog.Vamos()
sage: frozenset() in M.modular_cut(['ab', 'cd'])
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> frozenset() in M.modular_cut(['ab', 'cd'])
True
```

In any extension of the matroid $S_8 \setminus h$, a point on the lines through $\{c, g\}$ and $\{a, e\}$ also is on the line through $\{b, f\}$:

```
sage: M = matroids.catalog.S8()
sage: N = M.delete('h')
sage: frozenset('bf') in N.modular_cut(['cg', 'ae'])
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.S8()
>>> N = M.delete('h')
>>> frozenset('bf') in N.modular_cut(['cg', 'ae'])
True
```

The modular cut of the full groundset is equal to just the groundset:

```
sage: M = matroids.catalog.Fano()
sage: M.modular_cut([M.groundset()]).difference(
....:                                 [frozenset(M.groundset())])
set()
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.modular_cut([M.groundset()]).difference(
...                                [frozenset(M.groundset())])
set()
```

no_broken_circuits_sets(*ordering=None*)

Return the no broken circuits (NBC) sets of `self`.

An NBC set is a subset A of the groundset under some total ordering $<$ such that A contains no broken circuit.

INPUT:

- `ordering` – list (optional); a total ordering of the groundset

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: from sage.matroids.basis_matroid import BasisMatroid
sage: M = BasisMatroid(Matroid(circuits=[[1,2,3], [3,4,5], [1,2,4,5]]))
sage: SimplicialComplex(M.no_broken_circuits_sets())                    #␣
↪needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
sage: SimplicialComplex(M.no_broken_circuits_sets([5,4,3,2,1]))        #␣
↪needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

```
>>> from sage.all import *
>>> from sage.matroids.basis_matroid import BasisMatroid
>>> M = BasisMatroid(Matroid(circuits=[[Integer(1),Integer(2),Integer(3)],␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(3),Integer(4),Integer(5)], [Integer(1),Integer(2),Integer(4),
↪Integer(5)]]))
>>> SimplicialComplex(M.no_broken_circuits_sets())                          #␣
↪needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
>>> SimplicialComplex(M.no_broken_circuits_sets([Integer(5),Integer(4),
↪Integer(3),Integer(2),Integer(1)]))             # needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

```
sage: M = Matroid(circuits=[[1,2,3], [1,4,5], [2,3,4,5]])
sage: SimplicialComplex(M.no_broken_circuits_sets([5,4,3,2,1]))             #␣
↪needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (2, 3, 5), (2, 4, 5), (3, 4, 5)}
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1),Integer(2),Integer(3)], [Integer(1),
↪Integer(4),Integer(5)], [Integer(2),Integer(3),Integer(4),Integer(5)]])
>>> SimplicialComplex(M.no_broken_circuits_sets([Integer(5),Integer(4),
↪Integer(3),Integer(2),Integer(1)]))             # needs sage.graphs
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (2, 3, 5), (2, 4, 5), (3, 4, 5)}
```

ALGORITHM:

The following algorithm is adapted from page 7 of [BDPR2011].

Note

Sage uses the convention that a broken circuit is found by removing a minimal element from a circuit, while [BDPR2011]. use the convention that removal of the *maximal* element of circuit yields a broken circuit. This implementation reverses the provided order so that it returns n.b.c. sets under the minimal-removal convention, while the implementation is not modified from the published algorithm.

no_broken_circuits_sets_iterator(*ordering=None*)

Return an iterator over no broken circuits (NBC) sets of `self`.

An NBC set is a subset A of the groundset under some total ordering $<$ such that A contains no broken circuit.

INPUT:

- `ordering` – list (optional); a total ordering of the groundset

EXAMPLES:

```
sage: M = Matroid(circuits=[[1,2,3], [3,4,5], [1,2,4,5]])
sage: SimplicialComplex(list(M.no_broken_circuits_sets_iterator()))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
sage: SimplicialComplex(list(M.no_broken_circuits_sets_iterator([5,4,3,2,1])))
```

(continues on next page)

(continued from previous page)

```
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1),Integer(2),Integer(3)], [Integer(3),
↪Integer(4),Integer(5)], [Integer(1),Integer(2),Integer(4),Integer(5)]])
>>> SimplicialComplex(list(M.no_broken_circuits_sets_iterator()))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
>>> SimplicialComplex(list(M.no_broken_circuits_sets_iterator([Integer(5),
↪Integer(4),Integer(3),Integer(2),Integer(1)])))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

```
sage: M = Matroid(circuits=[[1,2,3], [1,4,5], [2,3,4,5]])
sage: SimplicialComplex(list(M.no_broken_circuits_sets_iterator([5,4,3,2,1])))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (2, 3, 5), (2, 4, 5), (3, 4, 5)}
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1),Integer(2),Integer(3)], [Integer(1),
↪Integer(4),Integer(5)], [Integer(2),Integer(3),Integer(4),Integer(5)]])
>>> SimplicialComplex(list(M.no_broken_circuits_sets_iterator([Integer(5),
↪Integer(4),Integer(3),Integer(2),Integer(1)])))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (2, 3, 5), (2, 4, 5), (3, 4, 5)}
```

For a matroid with loops all sets contain the broken circuit $\emptyset$, and thus we shouldn't get any set as output:

```
sage: M = Matroid(groundset=[1,2,3], circuits=[[3]])
sage: list(M.no_broken_circuits_sets_iterator())
[]
```

```
>>> from sage.all import *
>>> M = Matroid(groundset=[Integer(1),Integer(2),Integer(3)],␣
↪circuits=[[Integer(3)]])
>>> list(M.no_broken_circuits_sets_iterator())
[]
```

nonbases()

Return the nonbases of the matroid.

A *nonbasis* is a set with cardinality `self.full_rank()` that is not a basis.

OUTPUT: `SetSystem`

See also

M.basis()

EXAMPLES:

```
sage: M = matroids.Uniform(2, 4)
sage: list(M.nonbases())
[]
sage: [sorted(X) for X in matroids.catalog.P6().nonbases()]
[['a', 'b', 'c']]
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2), Integer(4))
>>> list(M.nonbases())
[]
>>> [sorted(X) for X in matroids.catalog.P6().nonbases()]
[['a', 'b', 'c']]
```

ALGORITHM:

Test all subsets of the groundset of cardinality `self.full_rank()`

nonbases_iterator()

Return an iterator over the nonbases of the matroid.

A *nonbasis* is a set with cardinality `self.full_rank()` that is not a basis.

See also

M.basis()

ALGORITHM:

Test all subsets of the groundset of cardinality `self.full_rank()`.

EXAMPLES:

```
sage: M = matroids.Uniform(2, 4)
sage: list(M.nonbases_iterator())
[]
sage: [sorted(X) for X in matroids.catalog.P6().nonbases_iterator()]
[['a', 'b', 'c']]
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2), Integer(4))
>>> list(M.nonbases_iterator())
[]
>>> [sorted(X) for X in matroids.catalog.P6().nonbases_iterator()]
[['a', 'b', 'c']]
```

noncospanning_cocircuits()

Return the noncospanning cocircuits of the matroid.

A *noncospanning cocircuit* is a cocircuit whose corank is strictly smaller than the corank of the matroid.

OUTPUT: `SetSystem`

See also

M.cocircuit(), *M.corank()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano().dual()
sage: sorted([sorted(C) for C in M.noncospanning_cocircuits()])
[['a', 'b', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['b', 'c', 'd'], ['b', 'e', 'g'], ['c', 'f', 'g'],
['d', 'e', 'f']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano().dual()
>>> sorted([sorted(C) for C in M.noncospanning_cocircuits()])
[['a', 'b', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['b', 'c', 'd'], ['b', 'e', 'g'], ['c', 'f', 'g'],
['d', 'e', 'f']]
```

nonspanning_circuit_closures()

Return the closures of nonspanning circuits of the matroid.

A *nonspanning circuit closure* is a closed set containing a nonspanning circuit.

OUTPUT: a dictionary containing the nonspanning circuit closures of the matroid, indexed by their ranks

See also

M.nonspanning_circuits(), *M.closure()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: CC = M.nonspanning_circuit_closures()
sage: len(CC[2])
7
sage: len(CC[3])
Traceback (most recent call last):
...
KeyError: 3
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> CC = M.nonspanning_circuit_closures()
>>> len(CC[Integer(2)])
7
>>> len(CC[Integer(3)])
Traceback (most recent call last):
...
KeyError: 3
```

nonspanning_circuits()

Return the nonspanning circuits of the matroid.

A *nonspanning circuit* is a circuit whose rank is strictly smaller than the rank of the matroid.

OUTPUT: `SetSystem`

See also

M.circuit(), *M.rank()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted([sorted(C) for C in M.nonspanning_circuits()])
[['a', 'b', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['b', 'c', 'd'], ['b', 'e', 'g'], ['c', 'f', 'g'],
['d', 'e', 'f']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted([sorted(C) for C in M.nonspanning_circuits()])
[['a', 'b', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['b', 'c', 'd'], ['b', 'e', 'g'], ['c', 'f', 'g'],
['d', 'e', 'f']]
```

nonspanning_circuits_iterator()

Return an iterator over the nonspanning circuits of the matroid.

A *nonspanning circuit* is a circuit whose rank is strictly smaller than the rank of the matroid.

See also

M.circuit(), *M.rank()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted([sorted(C) for C in M.nonspanning_circuits_iterator()])
[['a', 'b', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['b', 'c', 'd'], ['b', 'e', 'g'], ['c', 'f', 'g'],
['d', 'e', 'f']]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted([sorted(C) for C in M.nonspanning_circuits_iterator()])
[['a', 'b', 'f'], ['a', 'c', 'e'], ['a', 'd', 'g'],
['b', 'c', 'd'], ['b', 'e', 'g'], ['c', 'f', 'g'],
['d', 'e', 'f']]
```

orlik_solomon_algebra(*R*, *ordering=None*, ***kwargs*)

Return the Orlik-Solomon algebra of `self`.

INPUT:

- `R` – the base ring
- `ordering` – (optional) an ordering of the groundset
- `invariant` – (optional) either a semigroup `G` whose `__call__` acts on the groundset, or pair `(G, action)` where `G` is a semigroup and `action` is a function `action(g,e)` which takes a pair of a

group element and a groundset element and returns the groundset element which is the result of `e` acted upon by `g`

See also

`OrlikSolomonAlgebra`

EXAMPLES:

```
sage: M = matroids.Uniform(3, 4)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS
Orlik-Solomon algebra of U(3, 4): Matroid of rank 3 on 4 elements
 with circuit-closures
 {3: {{0, 1, 2, 3}}}

sage: G = SymmetricGroup(3);                                                #␣
↪needs sage.groups
sage: OSG = M.orlik_solomon_algebra(QQ, invariant=G)                        #␣
↪needs sage.groups

sage: # needs sage.groups
sage: G = SymmetricGroup(4)
sage: action = lambda g,x: g(x+1)-1
sage: OSG1 = M.orlik_solomon_algebra(QQ, invariant=(G,action))
sage: OSG2 = M.orlik_solomon_algebra(QQ, invariant=(action,G))
sage: OSG1 is OSG2
True
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(3), Integer(4))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> OS
Orlik-Solomon algebra of U(3, 4): Matroid of rank 3 on 4 elements
 with circuit-closures
 {3: {{0, 1, 2, 3}}}

>>> G = SymmetricGroup(Integer(3));                                          ␣
↪     # needs sage.groups
>>> OSG = M.orlik_solomon_algebra(QQ, invariant=G)                          #␣
↪needs sage.groups

>>> # needs sage.groups
>>> G = SymmetricGroup(Integer(4))
>>> action = lambda g,x: g(x+Integer(1))-Integer(1)
>>> OSG1 = M.orlik_solomon_algebra(QQ, invariant=(G,action))
>>> OSG2 = M.orlik_solomon_algebra(QQ, invariant=(action,G))
>>> OSG1 is OSG2
True
```

partition()

Return a minimum number of disjoint independent sets that covers the groundset.

OUTPUT: list of disjoint independent sets that covers the groundset

EXAMPLES:

```
sage: M = matroids.catalog.Block_9_4()
sage: P = M.partition()
sage: all(map(M.is_independent,P))
True
sage: set.union(*P)==M.groundset()
True
sage: sum(map(len,P))==len(M.groundset())
True
sage: Matroid(matrix([])).partition()
[]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Block_9_4()
>>> P = M.partition()
>>> all(map(M.is_independent,P))
True
>>> set.union(*P)==M.groundset()
True
>>> sum(map(len,P))==len(M.groundset())
True
>>> Matroid(matrix([])).partition()
[]
```

ALGORITHM:

Reduce partition to a matroid intersection between a matroid sum and a partition matroid. It's known the direct method doesn't gain much advantage over matroid intersection. [Cun1986]

plot(*B=None*, *lineorders=None*, *pos_method=None*, *pos_dict=None*, *save_pos=False*)

Return geometric representation as a sage graphics object.

INPUT:

- `B` – (optional) list containing a basis; if internal point placement is used, these elements will be placed as vertices of a triangle
- `lineorders` – (optional) list of lists where each of the inner lists specify groundset elements in a certain order which will be used to draw the corresponding line in geometric representation (if it exists)
- **`pos_method` – integer specifying positioning method**
  - `0`: default positioning
  - `1`: use pos_dict if it is not `None`
  - `2`: force directed (Not yet implemented)
- `pos_dict` – dictionary mapping groundset elements to their (x,y) positions
- `save_pos` – boolean indicating that point placements (either internal or user provided) and line orders (if provided) will be cached in the matroid (`M._cached_info`) and can be used for reproducing the geometric representation during the same session

OUTPUT:

A sage graphics object of type <class 'sage.plot.graphics.Graphics'> that corresponds to the geometric representation of the matroid.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: G = M.plot()                                                          #␣
↪needs sage.plot sage.rings.finite_rings
sage: type(G)                                                               #␣
↪needs sage.plot sage.rings.finite_rings
<class 'sage.plot.graphics.Graphics'>
sage: G.show()                                                              #␣
↪needs sage.plot sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> G = M.plot()                                                            #␣
↪needs sage.plot sage.rings.finite_rings
>>> type(G)                                                                 #␣
↪needs sage.plot sage.rings.finite_rings
<class 'sage.plot.graphics.Graphics'>
>>> G.show()                                                                #␣
↪needs sage.plot sage.rings.finite_rings
```

rank(*X=None*)

Return the rank of `X`.

The *rank* of a subset X is the size of the largest independent set contained in X.

If `X` is `None`, the rank of the groundset is returned.

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.rank()
3
sage: M.rank(['a', 'b', 'f'])
2
sage: M.rank(['a', 'b', 'x'])
Traceback (most recent call last):
...
ValueError: ['a', 'b', 'x'] is not a subset of the groundset
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.rank()
3
>>> M.rank(['a', 'b', 'f'])
2
>>> M.rank(['a', 'b', 'x'])
Traceback (most recent call last):
...
ValueError: ['a', 'b', 'x'] is not a subset of the groundset
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: from sage.matroids.rank_matroid import RankMatroid
sage: N = matroids.catalog.Sp8pp()
sage: M = RankMatroid(groundset=N.groundset(), rank_function=N.rank)
sage: sorted(M.groundset())
[1, 2, 3, 4, 5, 6, 7, 8]
sage: N = M.relabel({8: 0})
sage: sorted(N.groundset())
[0, 1, 2, 3, 4, 5, 6, 7]
sage: M.is_isomorphic(N)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.rank_matroid import RankMatroid
>>> N = matroids.catalog.Sp8pp()
>>> M = RankMatroid(groundset=N.groundset(), rank_function=N.rank)
>>> sorted(M.groundset())
[1, 2, 3, 4, 5, 6, 7, 8]
>>> N = M.relabel({Integer(8): Integer(0)})
>>> sorted(N.groundset())
[0, 1, 2, 3, 4, 5, 6, 7]
>>> M.is_isomorphic(N)
True
```

show(*B=None*, *lineorders=None*, *pos_method=None*, *pos_dict=None*, *save_pos=False*, *lims=None*)

Show the geometric representation of the matroid.

INPUT:

- `B` – (optional) a list containing elements of the groundset not in any particular order. If internal point placement is used, these elements will be placed as vertices of a triangle.
- `lineorders` – (optional) a list of lists where each of the inner lists specify groundset elements in a certain order which will be used to draw the corresponding line in geometric representation (if it exists)
- **`pos_method`** – **integer specifying the positioning method**
  - `0`: default positioning
  - `1`: use pos_dict if it is not `None`
  - `2`: Force directed (Not yet implemented).
- `pos_dict` – dictionary mapping groundset elements to their (x, y) positions
- `save_pos` – boolean indicating that point placements (either internal or user provided) and line orders (if provided) will be cached in the matroid (`M._cached_info`) and can be used for reproducing the geometric representation during the same session

- `lims` – list of 4 elements `[xmin,xmax,ymin,ymax]`

EXAMPLES:

```
sage: M = matroids.catalog.TernaryDowling3()
sage: M.show(B=['a','b','c'])                                                #␣
↪needs sage.plot sage.rings.finite_rings
sage: M.show(B=['a','b','c'], lineorders=[['f','e','i']])                    #␣
↪needs sage.plot sage.rings.finite_rings
sage: pos = {'a':(0,0), 'b': (0,1), 'c':(1,0), 'd':(1,1),                    #␣
↪needs sage.plot
....:        'e':(1,-1), 'f':(-1,1), 'g':(-1,-1),'h':(2,0), 'i':(0,2)}
sage: M.show(pos_method=1, pos_dict=pos, lims=[-3,3,-3,3])                   #␣
↪needs sage.plot sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> M = matroids.catalog.TernaryDowling3()
>>> M.show(B=['a','b','c'])                                                  #␣
↪needs sage.plot sage.rings.finite_rings
>>> M.show(B=['a','b','c'], lineorders=[['f','e','i']])                      #␣
↪needs sage.plot sage.rings.finite_rings
>>> pos = {'a':(Integer(0),Integer(0)), 'b': (Integer(0),Integer(1)), 'c
↪':(Integer(1),Integer(0)), 'd':(Integer(1),Integer(1)),                     #␣
↪needs sage.plot
...        'e':(Integer(1),-Integer(1)), 'f':(-Integer(1),Integer(1)), 'g':(-
↪Integer(1),-Integer(1)),'h':(Integer(2),Integer(0)), 'i':(Integer(0),
↪Integer(2))}
>>> M.show(pos_method=Integer(1), pos_dict=pos, lims=[-Integer(3),Integer(3),-
↪Integer(3),Integer(3)])                  # needs sage.plot sage.rings.
↪finite_rings
```

simplify()

Return the simplification of the matroid.

A matroid is *simple* if it contains no circuits of length 1 or 2. The *simplification* of a matroid is obtained by deleting all loops (circuits of length 1) and deleting all but one element from each parallel class (a closed set of rank 1, that is, each pair in it forms a circuit of length 2).

OUTPUT: matroid

See also

M.is_simple(), *M.loops()*, *M.circuit()*, *M.cosimplify()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano().contract('a')
sage: M.size() - M.simplify().size()
3
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano().contract('a')
>>> M.size() - M.simplify().size()
3
```

size()

Return the size of the groundset.

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.size()
8
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.size()
8
```

ternary_matroid(*randomized_tests=1*, *verify=True*)

Return a ternary matroid representing `self`, if such a representation exists.

INPUT:

- `randomized_tests` – (default: 1) an integer; the number of times a certain necessary condition for being ternary is tested, using randomization
- `verify` – boolean (default: `True`); if `True`, any output will be a ternary matroid representing `self`; if `False`, any output will represent `self` if and only if the matroid is ternary

OUTPUT: either a *TernaryMatroid*, or `None`

ALGORITHM:

First, compare the ternary matroids local to two random bases. If these matroids are not isomorphic, return `None`. This test is performed `randomized_tests` times. Next, if `verify` is `True`, test if a ternary matroid local to some basis is isomorphic to `self`.

See also

`M._local_ternary_matroid()`

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.ternary_matroid() is None                                               #␣
↪needs sage.graphs
True
sage: N = matroids.catalog.NonFano()
sage: N.ternary_matroid()                                                       #␣
↪needs sage.graphs
NonFano: Ternary matroid of rank 3 on 7 elements, type 0-
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.ternary_matroid() is None                                                 #␣
↪needs sage.graphs
True
>>> N = matroids.catalog.NonFano()
```

(continues on next page)

(continued from previous page)

```
>>> N.ternary_matroid()                                                    #␣
↪needs sage.graphs
NonFano: Ternary matroid of rank 3 on 7 elements, type 0-
```

truncation()

Return a rank-1 truncation of the matroid.

Let M be a matroid of rank r. The *truncation* of M is the matroid obtained by declaring all subsets of size r dependent. It can be obtained by adding an element freely to the span of the matroid and then contracting that element.

OUTPUT: matroid

See also

M.extension(), *M.contract()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: N = M.truncation()
sage: N.is_isomorphic(matroids.Uniform(2, 7))
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> N = M.truncation()
>>> N.is_isomorphic(matroids.Uniform(Integer(2), Integer(7)))
True
```

tutte_polynomial(*x=None*, *y=None*)

Return the Tutte polynomial of the matroid.

The *Tutte polynomial* of a matroid is the polynomial

$$T(x,y) = \sum_{A \subseteq E} (x-1)^{r(E)-r(A)}(y-1)^{r^*(E)-r^*(E \setminus A)},$$

where E is the groundset of the matroid, r is the rank function, and r^* is the corank function. Tutte defined his polynomial differently:

$$T(x,y) = \sum_{B} x^i(B)y^e(B),$$

where the sum ranges over all bases of the matroid, $i(B)$ is the number of internally active elements of B, and $e(B)$ is the number of externally active elements of B.

INPUT:

- `x` – (optional) a variable or numerical argument
- `y` – (optional) a variable or numerical argument

OUTPUT:

The Tutte-polynomial $T(x,y)$, where x and y are substituted with any values provided as input.

Todo

Make implementation more efficient, e.g. generalizing the approach from Issue #1314 from graphs to matroids.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.tutte_polynomial()
y^4 + x^3 + 3*y^3 + 4*x^2 + 7*x*y + 6*y^2 + 3*x + 3*y
sage: M.tutte_polynomial(1, 1) == M.bases_count()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.tutte_polynomial()
y^4 + x^3 + 3*y^3 + 4*x^2 + 7*x*y + 6*y^2 + 3*x + 3*y
>>> M.tutte_polynomial(Integer(1), Integer(1)) == M.bases_count()
True
```

ALGORITHM:

Enumerate the bases and compute the internal and external activities for each B.

union(*matroids*)

Return the matroid union with another matroid or a list of matroids.

Let $(M_1, M_2, \ldots, M_k)$ be a list of matroids where each M_i has groundset E_i. The *matroid union* M of $(M_1, M_2, \ldots, M_k)$ has groundset $E = \cup E_i$. Moreover, a set $I \subseteq E$ is independent in M if and only if the restriction of I to E_i is independent in M_i for every i.

INPUT:

- `matroids` – matroid or a list of matroids

OUTPUT: an instance of `MatroidUnion`

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: N = M.union(matroids.catalog.NonFano()); N
Matroid of rank 6 on 7 elements as matroid union of
Binary matroid of rank 3 on 7 elements, type (3, 0)
Ternary matroid of rank 3 on 7 elements, type 0-
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> N = M.union(matroids.catalog.NonFano()); N
Matroid of rank 6 on 7 elements as matroid union of
Binary matroid of rank 3 on 7 elements, type (3, 0)
Ternary matroid of rank 3 on 7 elements, type 0-
```

whitney_numbers()

Return the Whitney numbers of the first kind of the matroid.

The Whitney numbers of the first kind – here encoded as a vector $(w_0 = 1, \ldots, w_r)$ – are numbers of alternating sign, where w_i is the value of the coefficient of the $(r-i)$-th degree term of the matroid's characteristic

polynomial. Moreover, $|w_i|$ is the number of $(i-1)$-dimensional faces of the broken circuit complex of the matroid.

OUTPUT: list of integers

EXAMPLES:

```
sage: M = matroids.catalog.BetsyRoss()
sage: M.whitney_numbers()
[1, -11, 35, -25]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.BetsyRoss()
>>> M.whitney_numbers()
[1, -11, 35, -25]
```

whitney_numbers2()

Return the Whitney numbers of the second kind of the matroid.

The Whitney numbers of the second kind are here encoded as a vector $(W_0, \ldots, W_r)$, where W_i is the number of flats of rank i, and r is the rank of the matroid.

OUTPUT: list of integers

EXAMPLES:

```
sage: M = matroids.catalog.BetsyRoss()
sage: M.whitney_numbers2()
[1, 11, 20, 1]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.BetsyRoss()
>>> M.whitney_numbers2()
[1, 11, 20, 1]
```

CHAPTER

TWO

DATABASE OF MATROIDS

2.1 Catalog of matroids

A list of parametrized and individual matroids. The individual matroids are grouped into collections.

All listed matroids are available via tab completion. Simply type `matroids.` + `Tab` to see the various constructions available. For individual matroids type `matroids.catalog.` + `Tab`.

Note

To create a custom matroid using a variety of inputs, see the function *Matroid()*.

- **Parametrized matroids (`matroids.` + `Tab`)**
  - *matroids.AG*
  - *matroids.CompleteGraphic*
  - *matroids.PG*
  - *matroids.Psi*
  - *matroids.Spike*
  - *matroids.Theta*
  - *matroids.Uniform*
  - *matroids.Wheel*
  - *matroids.Whirl*
  - *matroids.Z*
- **Oxley's matroid collection (`matroids.catalog.` + `Tab`)**
  - *matroids.catalog.U24*
  - *matroids.catalog.U25*
  - *matroids.catalog.U35*
  - *matroids.catalog.K4*
  - *matroids.catalog.Whirl3*
  - *matroids.catalog.Q6*
  - *matroids.catalog.P6*
  - *matroids.catalog.U36*

- *matroids.catalog.R6*
- *matroids.catalog.Fano*
- *matroids.catalog.FanoDual*
- *matroids.catalog.NonFano*
- *matroids.catalog.NonFanoDual*
- *matroids.catalog.O7*
- *matroids.catalog.P7*
- *matroids.catalog.AG32*
- *matroids.catalog.AG32prime*
- *matroids.catalog.R8*
- *matroids.catalog.F8*
- *matroids.catalog.Q8*
- *matroids.catalog.L8*
- *matroids.catalog.S8*
- *matroids.catalog.Vamos*
- *matroids.catalog.T8*
- *matroids.catalog.J*
- *matroids.catalog.P8*
- *matroids.catalog.P8pp*
- *matroids.catalog.Wheel4*
- *matroids.catalog.Whirl4*
- *matroids.catalog.K33dual*
- *matroids.catalog.K33*
- *matroids.catalog.AG23*
- *matroids.catalog.TernaryDowling3*
- *matroids.catalog.R9*
- *matroids.catalog.Pappus*
- *matroids.catalog.NonPappus*
- *matroids.catalog.K5*
- *matroids.catalog.K5dual*
- *matroids.catalog.R10*
- *matroids.catalog.NonDesargues*
- *matroids.catalog.R12*
- *matroids.catalog.ExtendedTernaryGolayCode*
- *matroids.catalog.T12*
- *matroids.catalog.PG23*

- **Brettell's matroid collection (`matroids.catalog.` + `Tab`)**
  - *matroids.catalog.RelaxedNonFano*
  - *matroids.catalog.AG23minusDY*
  - *matroids.catalog.TQ8*
  - *matroids.catalog.P8p*
  - *matroids.catalog.KP8*
  - *matroids.catalog.Sp8*
  - *matroids.catalog.Sp8pp*
  - *matroids.catalog.LP8*
  - *matroids.catalog.WQ8*
  - *matroids.catalog.BB9*
  - *matroids.catalog.TQ9*
  - *matroids.catalog.TQ9p*
  - *matroids.catalog.M8591*
  - *matroids.catalog.PP9*
  - *matroids.catalog.BB9gDY*
  - *matroids.catalog.A9*
  - *matroids.catalog.FN9*
  - *matroids.catalog.FX9*
  - *matroids.catalog.KR9*
  - *matroids.catalog.KQ9*
  - *matroids.catalog.UG10*
  - *matroids.catalog.FF10*
  - *matroids.catalog.GP10*
  - *matroids.catalog.FZ10*
  - *matroids.catalog.UQ10*
  - *matroids.catalog.FP10*
  - *matroids.catalog.TQ10*
  - *matroids.catalog.FY10*
  - *matroids.catalog.PP10*
  - *matroids.catalog.FU10*
  - *matroids.catalog.D10*
  - *matroids.catalog.UK10*
  - *matroids.catalog.PK10*
  - *matroids.catalog.GK10*
  - *matroids.catalog.FT10*

- *matroids.catalog.TK10*
- *matroids.catalog.KT10*
- *matroids.catalog.TU10*
- *matroids.catalog.UT10*
- *matroids.catalog.FK10*
- *matroids.catalog.KF10*
- *matroids.catalog.FA11*
- *matroids.catalog.FR12*
- *matroids.catalog.GP12*
- *matroids.catalog.FQ12*
- *matroids.catalog.FF12*
- *matroids.catalog.FZ12*
- *matroids.catalog.UQ12*
- *matroids.catalog.FP12*
- *matroids.catalog.FS12*
- *matroids.catalog.UK12*
- *matroids.catalog.UA12*
- *matroids.catalog.AK12*
- *matroids.catalog.FK12*
- *matroids.catalog.KB12*
- *matroids.catalog.AF12*
- *matroids.catalog.NestOfTwistedCubes*
- *matroids.catalog.XY13*
- *matroids.catalog.N3*
- *matroids.catalog.N3pp*
- *matroids.catalog.UP14*
- *matroids.catalog.VP14*
- *matroids.catalog.FV14*
- *matroids.catalog.OW14*
- *matroids.catalog.FM14*
- *matroids.catalog.FA15*
- *matroids.catalog.N4*

- **Collection of various matroids (`matroids.catalog.` + `Tab`)**
  - *matroids.catalog.NonVamos*
  - *matroids.catalog.TicTacToe*
  - *matroids.catalog.Q10*

- *matroids.catalog.N1*
- *matroids.catalog.N2*
- *matroids.catalog.BetsyRoss*
- *matroids.catalog.Block_9_4*
- *matroids.catalog.Block_10_5*
- *matroids.catalog.ExtendedBinaryGolayCode*
- *matroids.catalog.AG23minus*
- *matroids.catalog.NotP8*
- *matroids.catalog.D16*
- *matroids.catalog.Terrahawk*
- *matroids.catalog.R9A*
- *matroids.catalog.R9B*
- *matroids.catalog.P9*

2.2 Collections of matroids

This module contains functions that access the collections of matroids in the database. Each of these functions returns an iterator over the nonparametrized matroids from the corresponding collection. These functions can be viewed by typing `matroids.` + `Tab`.

AUTHORS:

- Giorgos Mousa (2023-12-08): initial version

`sage.matroids.database_collections.`**`AllMatroids`**(*n*, *r=None*, *type='all'*)

Iterate over all matroids of certain number of elements (and, optionally, of specific rank and type).

INPUT:

- `n` – integer; the number of elements of the matroids
- `r` – integer (optional); the rank of the matroids ($0 \leq r \leq n$)
- `type` – string (default: `'all'`); the type of the matroids. Must be one of the following:
  - `'all'` – all matroids; available: (n=0-9), (n=0-12, r=0-2), (n=0-11, r=3)
  - `'unorientable'` – all unorientable matroids; the rank r must be specified; available: (n=7-11, r=3), (n=7-9, r=4)
  - any other type for which there exists an `is_type` method; availability same as for `'all'`

EXAMPLES:

```
sage: for M in matroids.AllMatroids(2):                                    #␣
↪optional - matroid_database
....:     M
all_n02_r00_#0: Matroid of rank 0 on 2 elements with 1 bases
all_n02_r01_#0: Matroid of rank 1 on 2 elements with 2 bases
all_n02_r01_#1: Matroid of rank 1 on 2 elements with 1 bases
all_n02_r02_#0: Matroid of rank 2 on 2 elements with 1 bases
```

```
>>> from sage.all import *
>>> for M in matroids.AllMatroids(Integer(2)):
↪     # optional - matroid_database
...     M
all_n02_r00_#0: Matroid of rank 0 on 2 elements with 1 bases
all_n02_r01_#0: Matroid of rank 1 on 2 elements with 2 bases
all_n02_r01_#1: Matroid of rank 1 on 2 elements with 1 bases
all_n02_r02_#0: Matroid of rank 2 on 2 elements with 1 bases
```

```
sage: for M in matroids.AllMatroids(5, 3, 'simple'):
↪optional - matroid_database
....:     M
simple_n05_r03_#0: Matroid of rank 3 on 5 elements with 10 bases
simple_n05_r03_#1: Matroid of rank 3 on 5 elements with 9 bases
simple_n05_r03_#2: Matroid of rank 3 on 5 elements with 8 bases
simple_n05_r03_#3: Matroid of rank 3 on 5 elements with 6 bases
```

```
>>> from sage.all import *
>>> for M in matroids.AllMatroids(Integer(5), Integer(3), 'simple'):
↪             # optional - matroid_database
...     M
simple_n05_r03_#0: Matroid of rank 3 on 5 elements with 10 bases
simple_n05_r03_#1: Matroid of rank 3 on 5 elements with 9 bases
simple_n05_r03_#2: Matroid of rank 3 on 5 elements with 8 bases
simple_n05_r03_#3: Matroid of rank 3 on 5 elements with 6 bases
```

```
sage: # optional - matroid_database
sage: for M in matroids.AllMatroids(4, type='paving'):
....:     M
paving_n04_r00_#0: Matroid of rank 0 on 4 elements with 1 bases
paving_n04_r01_#0: Matroid of rank 1 on 4 elements with 4 bases
paving_n04_r01_#1: Matroid of rank 1 on 4 elements with 3 bases
paving_n04_r01_#2: Matroid of rank 1 on 4 elements with 2 bases
paving_n04_r01_#3: Matroid of rank 1 on 4 elements with 1 bases
paving_n04_r02_#0: Matroid of rank 2 on 4 elements with 6 bases
paving_n04_r02_#1: Matroid of rank 2 on 4 elements with 5 bases
paving_n04_r02_#2: Matroid of rank 2 on 4 elements with 4 bases
paving_n04_r02_#3: Matroid of rank 2 on 4 elements with 3 bases
paving_n04_r03_#0: Matroid of rank 3 on 4 elements with 4 bases
paving_n04_r03_#1: Matroid of rank 3 on 4 elements with 3 bases
paving_n04_r04_#0: Matroid of rank 4 on 4 elements with 1 bases
```

```
>>> from sage.all import *
>>> # optional - matroid_database
>>> for M in matroids.AllMatroids(Integer(4), type='paving'):
...     M
paving_n04_r00_#0: Matroid of rank 0 on 4 elements with 1 bases
paving_n04_r01_#0: Matroid of rank 1 on 4 elements with 4 bases
paving_n04_r01_#1: Matroid of rank 1 on 4 elements with 3 bases
paving_n04_r01_#2: Matroid of rank 1 on 4 elements with 2 bases
paving_n04_r01_#3: Matroid of rank 1 on 4 elements with 1 bases
paving_n04_r02_#0: Matroid of rank 2 on 4 elements with 6 bases
```

(continues on next page)

(continued from previous page)

```
paving_n04_r02_#1: Matroid of rank 2 on 4 elements with 5 bases
paving_n04_r02_#2: Matroid of rank 2 on 4 elements with 4 bases
paving_n04_r02_#3: Matroid of rank 2 on 4 elements with 3 bases
paving_n04_r03_#0: Matroid of rank 3 on 4 elements with 4 bases
paving_n04_r03_#1: Matroid of rank 3 on 4 elements with 3 bases
paving_n04_r04_#0: Matroid of rank 4 on 4 elements with 1 bases
```

```
sage: # optional - matroid_database
sage: for M in matroids.AllMatroids(10, 4):
....:     M
Traceback (most recent call last):
...
ValueError: (n=10, r=4, type='all') is not available in the database
sage: for M in matroids.AllMatroids(12, 3, 'unorientable'):
....:     M
Traceback (most recent call last):
...
ValueError: (n=12, r=3, type='unorientable') is not available in the database
sage: for M in matroids.AllMatroids(8, type='unorientable'):
....:     M
Traceback (most recent call last):
...
ValueError: The rank needs to be specified for type 'unorientable'
sage: for M in matroids.AllMatroids(6, type='nice'):
....:     M
Traceback (most recent call last):
...
AttributeError: The type 'nice' is not available. There needs to be an 'is_nice()'
attribute for the type to be supported.
```

```
>>> from sage.all import *
>>> # optional - matroid_database
>>> for M in matroids.AllMatroids(Integer(10), Integer(4)):
...     M
Traceback (most recent call last):
...
ValueError: (n=10, r=4, type='all') is not available in the database
>>> for M in matroids.AllMatroids(Integer(12), Integer(3), 'unorientable'):
...     M
Traceback (most recent call last):
...
ValueError: (n=12, r=3, type='unorientable') is not available in the database
>>> for M in matroids.AllMatroids(Integer(8), type='unorientable'):
...     M
Traceback (most recent call last):
...
ValueError: The rank needs to be specified for type 'unorientable'
>>> for M in matroids.AllMatroids(Integer(6), type='nice'):
...     M
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
AttributeError: The type 'nice' is not available. There needs to be an 'is_nice()'
attribute for the type to be supported.
```

REFERENCES:

The underlying database was retrieved from Yoshitake Matsumoto's Database of Matroids; see [Mat2012].

sage.matroids.database_collections.**BrettellMatroids**()

Iterate over Brettell's matroid collection.

EXAMPLES:

```
sage: BM = list(matroids.BrettellMatroids()); len(BM)
68
sage: import random
sage: M = random.choice(BM)
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> BM = list(matroids.BrettellMatroids()); len(BM)
68
>>> import random
>>> M = random.choice(BM)
>>> M.is_valid()
True
```

See also

Matroid catalog, under `Brettell's matroid collection`.

sage.matroids.database_collections.**OxleyMatroids**()

Iterate over Oxley's matroid collection.

EXAMPLES:

```
sage: OM = list(matroids.OxleyMatroids()); len(OM)
44
sage: import random
sage: M = random.choice(OM)
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> OM = list(matroids.OxleyMatroids()); len(OM)
44
>>> import random
>>> M = random.choice(OM)
>>> M.is_valid()
True
```

See also

Matroid catalog, under Oxley's matroid collection.

REFERENCES:

These matroids are the nonparametrized matroids that appear in the Appendix Some Interesting Matroids in [Oxl2011] (p. 639-64).

sage.matroids.database_collections.**VariousMatroids**()

Iterate over various other named matroids.

EXAMPLES:

```
sage: VM = list(matroids.VariousMatroids()); len(VM)
16
sage: import random
sage: M = random.choice(VM)
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> VM = list(matroids.VariousMatroids()); len(VM)
16
>>> import random
>>> M = random.choice(VM)
>>> M.is_valid()
True
```

See also

Matroid catalog, under Collection of various matroids.

2.3 Database of matroids

This module contains the implementation and documentation for all matroids in the database, accessible through matroids. and matroids.catalog. (type those lines followed by Tab for a list).

AUTHORS:

- Michael Welsh, Stefan van Zwam (2013-04-01): initial version
- Giorgos Mousa, Andreas Triantafyllos (2023-12-08): more matroids

REFERENCES:

For more information on the matroids that belong to Oxley's matroid collection, as well as for most parametrized matroids, see [Oxl2011].

For more information on the matroids that belong to Brettell's matroid collection, see the associated publications, [Bre2023] and [BP2023].

Note

The grouping of the matroids into collections can be viewed in the *catalog of matroids*.

sage.matroids.database_matroids.**A9**(*groundset=None*)

Return the matroid $A9$.

An excluded minor for K_2-representable matroids. The UPF is P_4. Uniquely $GF(5)$-representable. (An excluded minor for H_2-representable matroids.)

EXAMPLES:

```
sage: M = matroids.catalog.A9(); M
A9: Quaternary matroid of rank 3 on 9 elements
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.A9(); M
A9: Quaternary matroid of rank 3 on 9 elements
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**AF12**(*groundset=None*)

Return the matroid $AF12$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.AF12(); M
AF12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.AF12(); M
AF12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**AG**(*n*, *q*, *x=None*, *groundset=None*)

Return the affine geometry of dimension `n` over the finite field of order `q`.

The affine geometry can be obtained from the projective geometry by removing a hyperplane.

INPUT:

- `n` – positive integer; the dimension of the projective space. This is one less than the rank of the resulting matroid.
- `q` – positive integer that is a prime power; the order of the finite field
- `x` – string (default: `None`); the name of the generator of a non-prime field, used for non-prime fields. If not supplied, `'x'` is used.
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: a linear matroid whose elements are the points of $AG(n, q)$

EXAMPLES:

```
sage: M = matroids.AG(2, 3).delete(8)
sage: M.is_isomorphic(matroids.catalog.AG23minus())
True
sage: matroids.AG(5, 4, 'z').size() == ((4 ^ 6 - 1) / (4 - 1) - (4 ^ 5 - 1)/(4 -␣
↪1))
True
sage: M = matroids.AG(4, 2); M
AG(4, 2): Binary matroid of rank 5 on 16 elements, type (5, 0)
```

```
>>> from sage.all import *
>>> M = matroids.AG(Integer(2), Integer(3)).delete(Integer(8))
>>> M.is_isomorphic(matroids.catalog.AG23minus())
True
>>> matroids.AG(Integer(5), Integer(4), 'z').size() == ((Integer(4) ** Integer(6)␣
↪- Integer(1)) / (Integer(4) - Integer(1)) - (Integer(4) ** Integer(5) -␣
↪Integer(1))/(Integer(4) - Integer(1)))
True
>>> M = matroids.AG(Integer(4), Integer(2)); M
AG(4, 2): Binary matroid of rank 5 on 16 elements, type (5, 0)
```

REFERENCES:

[Oxl2011], p. 661.

sage.matroids.database_matroids.**AG23**(*groundset='abcdefghi'*)

Return the matroid $AG(2, 3)$.

The ternary affine plane. A matroid which is not graphic, not cographic, and not regular.

EXAMPLES:

```
sage: M = matroids.catalog.AG23(); M
AG(2, 3): Ternary matroid of rank 3 on 9 elements, type 3+
sage: M.is_valid() and M.is_3connected() and M.is_ternary()
True
sage: M.has_minor(matroids.catalog.K4())
False
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: M.delete(e).is_isomorphic(matroids.catalog.AG23minus())
True
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.AG23(); M
AG(2, 3): Ternary matroid of rank 3 on 9 elements, type 3+
>>> M.is_valid() and M.is_3connected() and M.is_ternary()
True
>>> M.has_minor(matroids.catalog.K4())
False
```

(continues on next page)

(continued from previous page)

```
>>> import random
>>> e = random.choice(list(M.groundset()))
>>> M.delete(e).is_isomorphic(matroids.catalog.AG23minus())
True
>>> M.automorphism_group().is_transitive()
True
```

REFERENCES:

[Oxl2011], p. 653.

sage.matroids.database_matroids.**AG23minus**(*groundset=None*)

Return the ternary affine plane minus a point.

This is a sixth-roots-of-unity matroid, and an excluded minor for the class of near-regular matroids.

EXAMPLES:

```
sage: M = matroids.catalog.AG23minus(); M
AG23minus: Matroid of rank 3 on 8 elements with circuit-closures
{2: {{'a', 'b', 'c'}, {'a', 'd', 'f'}, {'a', 'e', 'g'},
{'b', 'd', 'h'}, {'b', 'e', 'f'}, {'c', 'd', 'g'},
{'c', 'e', 'h'}, {'f', 'g', 'h'}},
3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.AG23minus(); M
AG23minus: Matroid of rank 3 on 8 elements with circuit-closures
{2: {{'a', 'b', 'c'}, {'a', 'd', 'f'}, {'a', 'e', 'g'},
{'b', 'd', 'h'}, {'b', 'e', 'f'}, {'c', 'd', 'g'},
{'c', 'e', 'h'}, {'f', 'g', 'h'}},
3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
>>> M.is_valid()
True
```

REFERENCES:

[Oxl2011], p. 653.

sage.matroids.database_matroids.**AG23minusDY**(*groundset=None*)

Return the matroid $AG23minusDY$.

The matroid obtained from a $AG(2,3) \setminus e$ by a single $\delta - Y$ exchange on a triangle. An excluded minor for near-regular matroids. UPF is S.

EXAMPLES:

```
sage: M = matroids.catalog.AG23minusDY(); M
Delta-Y of AG(2,3)\e: Ternary matroid of rank 4 on 8 elements, type 0-
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.AG23minusDY(); M
Delta-Y of AG(2,3)\e: Ternary matroid of rank 4 on 8 elements, type 0-
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**AG32**(*groundset='abcdefgh'*)

Return the matroid $AG(3, 2)$.

The binary affine cube.

EXAMPLES:

```
sage: M = matroids.catalog.AG32(); M
AG(3, 2): Binary matroid of rank 4 on 8 elements, type (4, 0)
sage: M.is_valid() and M.is_3connected()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.AG32(); M
AG(3, 2): Binary matroid of rank 4 on 8 elements, type (4, 0)
>>> M.is_valid() and M.is_3connected()
True
```

$AG(3, 2)$ is isomorphic to the unique tipless binary 4-spike:

```
sage: M.is_isomorphic(matroids.Z(4, False))
True
```

```
>>> from sage.all import *
>>> M.is_isomorphic(matroids.Z(Integer(4), False))
True
```

and it is identically self-dual:

```
sage: M.equals(M.dual())
True
```

```
>>> from sage.all import *
>>> M.equals(M.dual())
True
```

Every single-element deletion is isomorphic to F_7^* and every single-element contraction is isomorphic to F_7:

```
sage: F7 = matroids.catalog.Fano()
sage: F7D = matroids.catalog.FanoDual()
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: M.delete(e).is_isomorphic(F7D)
True
sage: M.contract(e).is_isomorphic(F7)
True
```

```
>>> from sage.all import *
>>> F7 = matroids.catalog.Fano()
>>> F7D = matroids.catalog.FanoDual()
>>> import random
>>> e = random.choice(list(M.groundset()))
>>> M.delete(e).is_isomorphic(F7D)
True
>>> M.contract(e).is_isomorphic(F7)
True
```

REFERENCES:

[Oxl2011], p. 645.

sage.matroids.database_matroids.**AG32prime**(*groundset=None*)

Return the matroid $AG(3,2)'$, represented as circuit closures.

The matroid $AG(3,2)'$ is an 8-element matroid of rank-4. It is a smallest non-representable matroid. It is the unique relaxation of $AG(3,2)$.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.AG32prime(); M
AG(3, 2)': Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
     {'a', 'c', 'd', 'f'}, {'a', 'c', 'e', 'g'}, {'a', 'd', 'g', 'h'},
     {'a', 'e', 'f', 'h'}, {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'},
     {'b', 'e', 'g', 'h'}, {'c', 'd', 'e', 'h'}, {'c', 'f', 'g', 'h'},
     {'d', 'e', 'f', 'g'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: setprint(M.noncospanning_cocircuits())
[{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
 {'a', 'c', 'd', 'f'}, {'a', 'd', 'g', 'h'}, {'a', 'e', 'f', 'h'},
 {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'}, {'b', 'd', 'f', 'h'},
 {'b', 'e', 'g', 'h'}, {'c', 'd', 'e', 'h'}, {'c', 'f', 'g', 'h'},
 {'d', 'e', 'f', 'g'}]
sage: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.AG32prime(); M
AG(3, 2)': Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
     {'a', 'c', 'd', 'f'}, {'a', 'c', 'e', 'g'}, {'a', 'd', 'g', 'h'},
     {'a', 'e', 'f', 'h'}, {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'},
     {'b', 'e', 'g', 'h'}, {'c', 'd', 'e', 'h'}, {'c', 'f', 'g', 'h'},
     {'d', 'e', 'f', 'g'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
>>> setprint(M.noncospanning_cocircuits())
[{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
```

(continues on next page)

(continued from previous page)

```
 {'a', 'c', 'd', 'f'}, {'a', 'd', 'g', 'h'}, {'a', 'e', 'f', 'h'},
 {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'}, {'b', 'd', 'f', 'h'},
 {'b', 'e', 'g', 'h'}, {'c', 'd', 'e', 'h'}, {'c', 'f', 'g', 'h'},
 {'d', 'e', 'f', 'g'}]
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
False
```

Self-dual but not identically self-dual:

```
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
```

```
>>> from sage.all import *
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
```

Every single-element deletion is isomorphic to F_7^* or $(F_7^-)^*$ and every single-element contraction is isomorphic to F_7 or F_7^-:

```
sage: F7 = matroids.catalog.Fano()
sage: F7D = matroids.catalog.FanoDual()
sage: F7m = matroids.catalog.NonFano()
sage: F7mD = matroids.catalog.NonFanoDual()
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: M.delete(e).is_isomorphic(F7D) or M.delete(e).is_isomorphic(F7mD)
True
sage: Me = M.contract(e)
sage: Me.is_isomorphic(F7) or Me.is_isomorphic(F7m)
True
```

```
>>> from sage.all import *
>>> F7 = matroids.catalog.Fano()
>>> F7D = matroids.catalog.FanoDual()
>>> F7m = matroids.catalog.NonFano()
>>> F7mD = matroids.catalog.NonFanoDual()
>>> import random
>>> e = random.choice(list(M.groundset()))
>>> M.delete(e).is_isomorphic(F7D) or M.delete(e).is_isomorphic(F7mD)
True
>>> Me = M.contract(e)
>>> Me.is_isomorphic(F7) or Me.is_isomorphic(F7m)
True
```

REFERENCES:

[Oxl2011], p. 646.

sage.matroids.database_matroids.**AK12**(*groundset=None*)

Return the matroid $AK12$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Not self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.AK12(); M
AK12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.AK12(); M
AK12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**BB9**(*groundset=None*)

Return the matroid $BB9$.

An excluded minor for K_2-representable matroids, and a restriction of the Betsy Ross matroid. The UPF is G. Uniquely $GF(5)$-representable. (An excluded minor for H_2-representable matroids.)

EXAMPLES:

```
sage: BB = matroids.catalog.BB9(); BB
BB9: Quaternary matroid of rank 3 on 9 elements
sage: BR = matroids.catalog.BetsyRoss()
sage: from itertools import combinations
sage: pairs = combinations(sorted(BR.groundset()), 2)
sage: for pair in pairs:
....:     if BR.delete(pair).is_isomorphic(BB):
....:         print(pair)
('a', 'h')
('b', 'i')
('c', 'j')
('d', 'f')
('e', 'g')
```

```
>>> from sage.all import *
>>> BB = matroids.catalog.BB9(); BB
BB9: Quaternary matroid of rank 3 on 9 elements
>>> BR = matroids.catalog.BetsyRoss()
>>> from itertools import combinations
>>> pairs = combinations(sorted(BR.groundset()), Integer(2))
>>> for pair in pairs:
...     if BR.delete(pair).is_isomorphic(BB):
...         print(pair)
('a', 'h')
('b', 'i')
('c', 'j')
('d', 'f')
('e', 'g')
```

sage.matroids.database_matroids.**BB9gDY**(*groundset=None*)

Return the matroid $BB9gDY$.

An excluded minor for K_2-representable matroids. The UPF is G. In a DY^*-equivalence class of 4 matroids, one of which can be obtained from *BB9* by a segment-cosegment exchange on $\{a, d, i, j\}$. Uniquely $GF(5)$-representable. (An excluded minor for H_2-representable matroids.)

EXAMPLES:

```
sage: M = matroids.catalog.BB9gDY(); M
Segment cosegment exchange on BB9: Quaternary matroid of rank 5 on 9 elements
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.BB9gDY(); M
Segment cosegment exchange on BB9: Quaternary matroid of rank 5 on 9 elements
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**BetsyRoss**(*groundset=None*)

Return the Betsy Ross matroid, represented by circuit closures.

An extremal golden-mean matroid. That is, if M is simple, rank 3, has the Betsy Ross matroid as a restriction and is a Golden Mean matroid, then M is the Betsy Ross matroid.

EXAMPLES:

```
sage: M = matroids.catalog.BetsyRoss(); M
BetsyRoss: Matroid of rank 3 on 11 elements with 25 nonspanning
circuits
sage: len(M.circuit_closures()[2])
10
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.BetsyRoss(); M
BetsyRoss: Matroid of rank 3 on 11 elements with 25 nonspanning
circuits
>>> len(M.circuit_closures()[Integer(2)])
10
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**Block_10_5**(*groundset=None*)

Return the paving matroid whose nonspanning circuits form the blocks of a $3-(10, 5, 3)$ design.

EXAMPLES:

```
sage: M = matroids.catalog.Block_10_5(); M
Block(10, 5): Matroid of rank 5 on 10 elements with 36 nonspanning
circuits
sage: M.is_valid() and M.is_paving()
True
sage: BD = BlockDesign(M.groundset(), list(M.nonspanning_circuits()))
sage: BD.is_t_design(return_parameters=True)
(True, (3, 10, 5, 3))
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Block_10_5(); M
Block(10, 5): Matroid of rank 5 on 10 elements with 36 nonspanning
circuits
>>> M.is_valid() and M.is_paving()
True
>>> BD = BlockDesign(M.groundset(), list(M.nonspanning_circuits()))
>>> BD.is_t_design(return_parameters=True)
(True, (3, 10, 5, 3))
```

sage.matroids.database_matroids.**Block_9_4**(*groundset=None*)

Return the paving matroid whose nonspanning circuits form the blocks of a $2-(9,4,3)$ design.

EXAMPLES:

```
sage: M = matroids.catalog.Block_9_4(); M
Block(9, 4): Matroid of rank 4 on 9 elements with 18 nonspanning
circuits
sage: M.is_valid() and M.is_paving()
True
sage: BD = BlockDesign(M.groundset(), list(M.nonspanning_circuits()))
sage: BD.is_t_design(return_parameters=True)
(True, (2, 9, 4, 3))
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Block_9_4(); M
Block(9, 4): Matroid of rank 4 on 9 elements with 18 nonspanning
circuits
>>> M.is_valid() and M.is_paving()
True
>>> BD = BlockDesign(M.groundset(), list(M.nonspanning_circuits()))
>>> BD.is_t_design(return_parameters=True)
(True, (2, 9, 4, 3))
```

sage.matroids.database_matroids.**CompleteGraphic**(*n*, *groundset=None*)

Return the cycle matroid of the complete graph on n vertices.

INPUT:

- `n` – integer; the number of vertices of the underlying complete graph

OUTPUT: the graphic matroid associated with the n-vertex complete graph. This matroid has rank $n-1$.

EXAMPLES:

```
sage: # needs sage.graphs
sage: from sage.matroids.advanced import setprint
sage: M = matroids.CompleteGraphic(5); M
M(K5): Graphic matroid of rank 4 on 10 elements
sage: M.has_minor(matroids.Uniform(2, 4))
False
sage: simplify(M.contract(randrange(0,
....:                 10))).is_isomorphic(matroids.CompleteGraphic(4))
True
sage: setprint(M.closure([0, 2, 4, 5]))
```

(continues on next page)

(continued from previous page)

```
{0, 1, 2, 4, 5, 7}
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> from sage.matroids.advanced import setprint
>>> M = matroids.CompleteGraphic(Integer(5)); M
M(K5): Graphic matroid of rank 4 on 10 elements
>>> M.has_minor(matroids.Uniform(Integer(2), Integer(4)))
False
>>> simplify(M.contract(randrange(Integer(0),
...                 Integer(10)))).is_isomorphic(matroids.
↪CompleteGraphic(Integer(4)))
True
>>> setprint(M.closure([Integer(0), Integer(2), Integer(4), Integer(5)]))
{0, 1, 2, 4, 5, 7}
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**D10**(*groundset=None*)

Return the matroid $D10$.

An excluded minor for P_4-representable matroids. UPF is G. Has a *TQ8*-minor. In a DY^*-equivalence class of 13 matroids.

EXAMPLES:

```
sage: M = matroids.catalog.D10(); M
D10: Quaternary matroid of rank 4 on 10 elements
sage: M.has_minor(matroids.catalog.TQ8())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.D10(); M
D10: Quaternary matroid of rank 4 on 10 elements
>>> M.has_minor(matroids.catalog.TQ8())
True
```

sage.matroids.database_matroids.**D16**(*groundset='abcdefghijklmnop'*)

Return the matroid D_{16}.

Let M be a 4-connected binary matroid and N an internally 4-connected proper minor of M with at least 7 elements. Then some element of M can be deleted or contracted preserving an N-minor, unless M is D_{16}.

EXAMPLES:

```
sage: M = matroids.catalog.D16(); M
D16: Binary matroid of rank 8 on 16 elements, type (0, 0)
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.D16(); M
D16: Binary matroid of rank 8 on 16 elements, type (0, 0)
>>> M.is_valid()
True
```

REFERENCES:

[CMO2012]

sage.matroids.database_matroids.**ExtendedBinaryGolayCode**(*groundset='abcdefghijklmnopqrstuvwx'*)

Return the matroid of the extended binary Golay code.

EXAMPLES:

```
sage: M = matroids.catalog.ExtendedBinaryGolayCode(); M
Extended Binary Golay Code: Binary matroid of rank 12 on 24 elements, type (12, 0)
sage: C = LinearCode(M.representation())
sage: C.is_permutation_equivalent(codes.GolayCode(GF(2)))
True
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.ExtendedBinaryGolayCode(); M
Extended Binary Golay Code: Binary matroid of rank 12 on 24 elements, type (12, 0)
>>> C = LinearCode(M.representation())
>>> C.is_permutation_equivalent(codes.GolayCode(GF(Integer(2))))
True
>>> M.is_valid()
True
```

See also

GolayCode

sage.matroids.database_matroids.**ExtendedTernaryGolayCode**(*groundset='abcdefghijkl'*)

Return the matroid of the extended ternary Golay code.

This is the unique Steiner system $S(5, 6, 12)$.

EXAMPLES:

```
sage: M = matroids.catalog.ExtendedTernaryGolayCode(); M
Extended Ternary Golay Code: Ternary matroid of rank 6 on 12 elements,
type 6+
sage: C = LinearCode(M.representation())
sage: C.is_permutation_equivalent(codes.GolayCode(GF(3)))
True
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.ExtendedTernaryGolayCode(); M
Extended Ternary Golay Code: Ternary matroid of rank 6 on 12 elements,
type 6+
>>> C = LinearCode(M.representation())
>>> C.is_permutation_equivalent(codes.GolayCode(GF(Integer(3))))
True
>>> M.is_valid()
True
```

The automorphism group is the 5-transitive Mathieu group $M12$:

sage: # long time sage: G = M.automorphism_group() sage: G.is_transitive() True sage: G.structure_description() 'M12'

$S(5, 6, 12)$ is an identically self-dual matroid:

```
sage: M.equals(M.dual())
True
```

```
>>> from sage.all import *
>>> M.equals(M.dual())
True
```

Every contraction of three elements is isomorphic to $AG(2, 3)$; every contraction of two elements and deletion of two elements is isomorphic to $P8$:

```
sage: import random, itertools
sage: C = list(itertools.combinations(M.groundset(), 3))
sage: elements = random.choice(C)
sage: N = M.contract(elements)
sage: N.is_isomorphic(matroids.catalog.AG23())
True
sage: C = list(itertools.combinations(M.groundset(), 4))
sage: elements = list(random.choice(C))
sage: random.shuffle(elements)
sage: N = M.contract(elements[:2]).delete(elements[2:4])
sage: N.is_isomorphic(matroids.catalog.P8())
True
```

```
>>> from sage.all import *
>>> import random, itertools
>>> C = list(itertools.combinations(M.groundset(), Integer(3)))
>>> elements = random.choice(C)
>>> N = M.contract(elements)
>>> N.is_isomorphic(matroids.catalog.AG23())
True
>>> C = list(itertools.combinations(M.groundset(), Integer(4)))
>>> elements = list(random.choice(C))
>>> random.shuffle(elements)
>>> N = M.contract(elements[:Integer(2)]).delete(elements[Integer(2):Integer(4)])
>>> N.is_isomorphic(matroids.catalog.P8())
True
```

See also

GolayCode

REFERENCES:

[Oxl2011], p. 658.

sage.matroids.database_matroids.**F8**(*groundset=None*)

Return the matroid F_8, represented as circuit closures.

The matroid F_8 is an 8-element matroid of rank-4. It is a smallest non-representable matroid.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.catalog.F8(); M
F8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
     {'a', 'c', 'd', 'f'}, {'a', 'c', 'e', 'g'}, {'a', 'd', 'g', 'h'},
     {'a', 'e', 'f', 'h'}, {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'},
     {'c', 'd', 'e', 'h'}, {'c', 'f', 'g', 'h'}, {'d', 'e', 'f', 'g'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: D = get_nonisomorphic_matroids([M.contract(i) for i in M.groundset()])
sage: len(D)
3
sage: [N.is_isomorphic(matroids.catalog.Fano()) for N in D]
[...True...]
sage: [N.is_isomorphic(matroids.catalog.NonFano()) for N in D]
[...True...]
sage: M.is_valid() and M.is_paving()
True
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.F8(); M
F8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
     {'a', 'c', 'd', 'f'}, {'a', 'c', 'e', 'g'}, {'a', 'd', 'g', 'h'},
     {'a', 'e', 'f', 'h'}, {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'},
     {'c', 'd', 'e', 'h'}, {'c', 'f', 'g', 'h'}, {'d', 'e', 'f', 'g'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
>>> D = get_nonisomorphic_matroids([M.contract(i) for i in M.groundset()])
>>> len(D)
3
>>> [N.is_isomorphic(matroids.catalog.Fano()) for N in D]
[...True...]
>>> [N.is_isomorphic(matroids.catalog.NonFano()) for N in D]
[...True...]
>>> M.is_valid() and M.is_paving()
```

(continues on next page)

(continued from previous page)

```
True
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 647.

sage.matroids.database_matroids.**FA11**(*groundset=None*)

Return the matroid $FA11$.

An excluded minor for P_4-representable matroids. UPF is PT. In a DY^*-equivalence class of 6 matroids. Has an *FF10*-minor (delete 10).

EXAMPLES:

```
sage: FA11 = matroids.catalog.FA11(); FA11
FA11: Quaternary matroid of rank 5 on 11 elements
sage: FF10 = matroids.catalog.FF10()
sage: FF10.is_isomorphic(FA11.delete(10))
True
```

```
>>> from sage.all import *
>>> FA11 = matroids.catalog.FA11(); FA11
FA11: Quaternary matroid of rank 5 on 11 elements
>>> FF10 = matroids.catalog.FF10()
>>> FF10.is_isomorphic(FA11.delete(Integer(10)))
True
```

sage.matroids.database_matroids.**FA15**(*groundset=None*)

Return the matroid $FA15$.

An excluded minor for O-representable matroids. UPF is PT. In a DY^*-equivalence class of 6 matroids. Has an *SQ14*-minor.

EXAMPLES:

```
sage: M = matroids.catalog.FA15(); M
FA15: Quaternary matroid of rank 7 on 15 elements
sage: M.has_minor(matroids.catalog.N3pp())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FA15(); M
FA15: Quaternary matroid of rank 7 on 15 elements
>>> M.has_minor(matroids.catalog.N3pp())
True
```

sage.matroids.database_matroids.**FF10**(*groundset=None*)

Return the matroid $FF10$.

An excluded minor for K_2-representable matroids. UPF is P_4. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.FF10(); M
FF10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FF10(); M
FF10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FF12**(*groundset=None*)

Return the matroid $FF12$.

An excluded minor for P_4-representable matroids. Self-dual. UPF is $(Q(a, b), < -1, a, b, a-2, a-1, a+1, b-1, ab-a+b, ab-a-b, ab-a-2b >)$. Has an *FF10*-minor (contract 'c' and delete 'd').

EXAMPLES:

```
sage: M = matroids.catalog.FF12(); M
FF12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
sage: FF10 = matroids.catalog.FF10()
sage: FF10.is_isomorphic(M.contract('c').delete('d'))
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FF12(); M
FF12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
True
>>> FF10 = matroids.catalog.FF10()
>>> FF10.is_isomorphic(M.contract('c').delete('d'))
True
```

sage.matroids.database_matroids.**FK10**(*groundset=None*)

Return the matroid $FK10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.FK10(); M
FK10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FK10(); M
FK10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FK12**(*groundset=None*)

Return the matroid $FK12$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.FK12(); M
FK12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FK12(); M
FK12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FM14**(*groundset=None*)

Return the matroid $FM14$.

An excluded minor for P_4-representable matroids. Self-dual. UPF is PT.

EXAMPLES:

```
sage: M = matroids.catalog.FM14(); M
FM14: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FM14(); M
FM14: Quaternary matroid of rank 7 on 14 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FN9**(*groundset=None*)

Return the matroid $FN9$.

An excluded minor for G- and K_2-representable matroids. In a DY^*-equivalence class of 10 matroids. UPF is $U_1^{(2)}$. (An excluded minor for H_2- and $GF(5)$-representable matroids.)

EXAMPLES:

```
sage: M = matroids.catalog.FN9(); M
FN9: Quaternary matroid of rank 3 on 9 elements
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FN9(); M
FN9: Quaternary matroid of rank 3 on 9 elements
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**FP10**(*groundset=None*)

Return the matroid $FP10$.

An excluded minor for K_2- and G-representable matroids (and H_2- and $GF(5)$-representable matroids). UPF is $U_1^{(2)}$. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.FP10(); M
FP10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FP10(); M
FP10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FP12**(*groundset=None*)

Return the matroid $FP12$.

An excluded minor for K_2- and G-representable matroids (and H_2- and $GF(5)$-representable matroids). UPF is W. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.FP12(); M
FP12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FP12(); M
FP12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FQ12**(*groundset=None*)

Return the matroid $FQ12$.

An excluded minor for P_4-representable matroids. UPF is PT. Has` a *PP9*-minor (contract 4 and 7, delete 6) and *FF10*-minor (contract 'c' and delete 'd').

EXAMPLES:

```
sage: FQ12 = matroids.catalog.FQ12(); FQ12
FQ12: Quaternary matroid of rank 6 on 12 elements
sage: PP9 = matroids.catalog.PP9()
sage: PP9.is_isomorphic(FQ12.contract([4,7]).delete(6))
True
sage: FF10 = matroids.catalog.FF10()
sage: FF10.is_isomorphic(FQ12.contract('c').delete('d'))
True
```

```
>>> from sage.all import *
>>> FQ12 = matroids.catalog.FQ12(); FQ12
FQ12: Quaternary matroid of rank 6 on 12 elements
>>> PP9 = matroids.catalog.PP9()
>>> PP9.is_isomorphic(FQ12.contract([Integer(4),Integer(7)]).delete(Integer(6)))
True
>>> FF10 = matroids.catalog.FF10()
>>> FF10.is_isomorphic(FQ12.contract('c').delete('d'))
True
```

sage.matroids.database_matroids.**FR12**(*groundset=None*)

Return the matroid $FR12$.

An excluded minor for K_2-representable matroids. UPF is P_4. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.FR12(); M
FR12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FR12(); M
FR12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FS12**(*groundset=None*)

Return the matroid $FS12$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Rank 5. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.FS12(); M
FS12: Quaternary matroid of rank 5 on 12 elements
sage: M.rank()
5
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FS12(); M
FS12: Quaternary matroid of rank 5 on 12 elements
>>> M.rank()
5
```

sage.matroids.database_matroids.**FT10**(*groundset=None*)

Return the matroid $FT10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.FT10(); M
FT10: Quaternary matroid of rank 5 on 10 elements
```

(continues on next page)

(continued from previous page)

```
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FT10(); M
FT10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FU10**(*groundset=None*)

Return the matroid $FU10$.

An excluded minor for P_4-representable matroids. UPF is G. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.FU10(); M
FU10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FU10(); M
FU10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**FV14**(*groundset=None*)

Return the matroid $FV14$.

An excluded minor for P_4-representable matroids. Not self-dual. UPF is PT.

EXAMPLES:

```
sage: M = matroids.catalog.FV14(); M
FV14: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FV14(); M
FV14: Quaternary matroid of rank 7 on 14 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**FX9**(*groundset=None*)

Return the matroid $FX9$.

An excluded minor for G- and K_2-representable matroids. UPF is $(Q(a, b), < -1, a, b, a - 1, b - 1, a - b, a + b, a + b - 2, a + b - 2ab >)$. (An excluded minor for H_2- and $GF(5)$-representable matroids.)

EXAMPLES:

```
sage: M = matroids.catalog.FX9(); M
FX9: Quaternary matroid of rank 4 on 9 elements
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FX9(); M
FX9: Quaternary matroid of rank 4 on 9 elements
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**FY10**(*groundset=None*)

Return the matroid $FY10$.

An excluded minor for P_4-representable matroids. UPF is G. Not self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.FY10(); M
FY10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FY10(); M
FY10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**FZ10**(*groundset=None*)

Return the matroid $FZ10$.

An excluded minor for K_2- and G-representable matroids (and H_2- and $GF(5)$-representable matroids). UPF is W. Not self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.FZ10(); M
FZ10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FZ10(); M
FZ10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**FZ12**(*groundset=None*)

Return the matroid $FZ12$.

An excluded minor for K_2- and G-representable matroids (and H_2- and $GF(5)$-representable matroids). UPF is W. Not self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.FZ12(); M
FZ12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.FZ12(); M
FZ12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**Fano**(*groundset='abcdefg'*)

Return the Fano matroid, represented over $GF(2)$.

The Fano matroid, or Fano plane, or F_7, is a 7-element matroid of rank-3. It is representable over a field if and only if that field has characteristic two.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Fano(); M
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
sage: M.automorphism_group().is_transitive()
True
sage: M.automorphism_group().structure_description()
'PSL(3,2)'
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Fano(); M
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
>>> M.automorphism_group().is_transitive()
True
>>> M.automorphism_group().structure_description()
'PSL(3,2)'
```

Every single-element deletion of F_7 is isomorphic to $M(K_4)$:

```
sage: setprint(sorted(M.nonspanning_circuits()))
[{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'}, {'b', 'c', 'd'},
 {'b', 'e', 'g'}, {'c', 'f', 'g'}, {'d', 'e', 'f'}]
sage: M.delete(M.groundset_list()[randrange(0,
....:                  7)]).is_isomorphic(matroids.CompleteGraphic(4))
True
```

```
>>> from sage.all import *
>>> setprint(sorted(M.nonspanning_circuits()))
[{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'}, {'b', 'c', 'd'},
 {'b', 'e', 'g'}, {'c', 'f', 'g'}, {'d', 'e', 'f'}]
>>> M.delete(M.groundset_list()[randrange(Integer(0),
...                   Integer(7))]).is_isomorphic(matroids.
↪CompleteGraphic(Integer(4)))
True
```

It is also the projective plane of order two, i.e. $PG(2, 2)$:

```
sage: M.is_isomorphic(matroids.PG(2, 2))
True
```

```
>>> from sage.all import *
>>> M.is_isomorphic(matroids.PG(Integer(2), Integer(2)))
True
```

F_7 is isomorphic to the unique binary 3-spike:

```
sage: M.is_isomorphic(matroids.Z(3))
True
```

```
>>> from sage.all import *
>>> M.is_isomorphic(matroids.Z(Integer(3)))
True
```

REFERENCES:

[Oxl2011], p. 643.

sage.matroids.database_matroids.**FanoDual**(*groundset='abcdefg'*)

Return the dual of the Fano matroid.

F_7^* is a 7-element matroid of rank-3.

EXAMPLES:

```
sage: F7 = matroids.catalog.Fano()
sage: F7D = matroids.catalog.FanoDual(); F7D
F7*: Binary matroid of rank 4 on 7 elements, type (3, 7)
sage: F7.is_isomorphic(F7D.dual())
True
sage: F7D.automorphism_group().is_transitive()
True
sage: F7D.automorphism_group().structure_description()
'PSL(3,2)'
```

```
>>> from sage.all import *
>>> F7 = matroids.catalog.Fano()
>>> F7D = matroids.catalog.FanoDual(); F7D
F7*: Binary matroid of rank 4 on 7 elements, type (3, 7)
>>> F7.is_isomorphic(F7D.dual())
True
>>> F7D.automorphism_group().is_transitive()
True
>>> F7D.automorphism_group().structure_description()
'PSL(3,2)'
```

Every single-element deletion of F_7^* is isomorphic to $M(K_{2,3})$:

```
sage: K2_3 = Matroid(graphs.CompleteBipartiteGraph(2, 3))
sage: import random
sage: e = random.choice(list(F7D.groundset()))
```

(continues on next page)

(continued from previous page)

```
sage: F7D.delete(e).is_isomorphic(K2_3)
True
```

```
>>> from sage.all import *
>>> K2_3 = Matroid(graphs.CompleteBipartiteGraph(Integer(2), Integer(3)))
>>> import random
>>> e = random.choice(list(F7D.groundset()))
>>> F7D.delete(e).is_isomorphic(K2_3)
True
```

REFERENCES:

[Oxl2011], p. 643.

sage.matroids.database_matroids.**GK10**(*groundset=None*)

Return the matroid $GK10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Not self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.GK10(); M
GK10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.GK10(); M
GK10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**GP10**(*groundset=None*)

Return the matroid $GP10$.

An excluded minor for K_2-representable matroids. UPF is G. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.GP10(); M
GP10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.GP10(); M
GP10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**GP12**(*groundset=None*)

Return the matroid $GP12$.

An excluded minor for K_2-representable matroids. UPF is G. Not self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.GP12(); M
GP12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.GP12(); M
GP12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**J**(*groundset='abcdefgh'*)

Return the matroid J, represented over $GF(3)$.

The matroid J is an 8-element matroid of rank-4. It is representable over a field if and only if that field has at least three elements.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.J(); M
J: Ternary matroid of rank 4 on 8 elements, type 0-
sage: setprint(M.truncation().nonbases())
[{'a', 'b', 'f'}, {'a', 'c', 'g'}, {'a', 'd', 'h'}]
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.has_minor(matroids.CompleteGraphic(4))
False
sage: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.J(); M
J: Ternary matroid of rank 4 on 8 elements, type 0-
>>> setprint(M.truncation().nonbases())
[{'a', 'b', 'f'}, {'a', 'c', 'g'}, {'a', 'd', 'h'}]
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.has_minor(matroids.CompleteGraphic(Integer(4)))
False
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 650.

sage.matroids.database_matroids.**K33**(*groundset='abcdefghi'*)

Return the graphic matroid $M(K_{3,3})$.

$M(K_{3,3})$ is an excluded minor for the class of cographic matroids.

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = matroids.catalog.K33(); M
M(K3, 3): Regular matroid of rank 5 on 9 elements with 81 bases
sage: M.is_valid()
True
sage: G1 = M.automorphism_group()
sage: G2 = matroids.catalog.K33dual().automorphism_group()
sage: G1.is_isomorphic(G2)
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = matroids.catalog.K33(); M
M(K3, 3): Regular matroid of rank 5 on 9 elements with 81 bases
>>> M.is_valid()
True
>>> G1 = M.automorphism_group()
>>> G2 = matroids.catalog.K33dual().automorphism_group()
>>> G1.is_isomorphic(G2)
True
```

REFERENCES:

[Oxl2011], p. 652-3.

sage.matroids.database_matroids.**K33dual**(*groundset='abcdefghi'*)

Return the matroid $M * (K_{3,3})$, represented over the regular partial field.

The matroid $M*(K_{3,3})$ is a 9-element matroid of rank-4. It is an excluded minor for the class of graphic matroids. It is the graft matroid of the 4-wheel with every vertex except the hub being coloured.

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = matroids.catalog.K33dual(); M
M*(K3, 3): Regular matroid of rank 4 on 9 elements with 81 bases
sage: any(N.is_3connected() for N in M.linear_extensions(simple=True))
False
sage: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = matroids.catalog.K33dual(); M
M*(K3, 3): Regular matroid of rank 4 on 9 elements with 81 bases
>>> any(N.is_3connected() for N in M.linear_extensions(simple=True))
False
>>> M.is_valid()
True
```

(continues on next page)

(continued from previous page)

```
>>> M.automorphism_group().is_transitive()
True
```

REFERENCES:

[Oxl2011], p. 652-3.

sage.matroids.database_matroids.**K4**(*groundset='abcdef'*)

The graphic matroid of the complete graph K_4.

EXAMPLES:

```
sage: M = matroids.catalog.K4(); M
M(K4): Graphic matroid of rank 3 on 6 elements
sage: M.is_graphic()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.K4(); M
M(K4): Graphic matroid of rank 3 on 6 elements
>>> M.is_graphic()
True
```

$M(K_4)$ is isomorphic to $M(\mathcal{W}_3)$, the rank-3 wheel:

```
sage: W3 = matroids.Wheel(3)
sage: M.is_isomorphic(W3)
True
```

```
>>> from sage.all import *
>>> W3 = matroids.Wheel(Integer(3))
>>> M.is_isomorphic(W3)
True
```

and to the tipless binary 3-spike:

```
sage: Z = matroids.Z(3, False)
sage: M.is_isomorphic(Z)
True
```

```
>>> from sage.all import *
>>> Z = matroids.Z(Integer(3), False)
>>> M.is_isomorphic(Z)
True
```

It has a transitive automorphism group:

```
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> M.automorphism_group().is_transitive()
True
```

REFERENCES:

[Oxl2011], p. 640.

sage.matroids.database_matroids.**K5**(*groundset='abcdefghij'*)

Return the graphic matroid $M(K_5)$.

$M(K_5)$ is an excluded minor for the class of cographic matroids. It is the 3-dimensional Desargues configuration.

EXAMPLES:

```
sage: M = matroids.catalog.K5(); M
M(K5): Graphic matroid of rank 4 on 10 elements
sage: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.K5(); M
M(K5): Graphic matroid of rank 4 on 10 elements
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
True
```

REFERENCES:

[Oxl2011], p. 656.

sage.matroids.database_matroids.**K5dual**(*groundset='abcdefghij'*)

Return the matroid $M^*(K_5)$.

$M^*(K_5)$ is an excluded minor for the class of graphic matroids.

EXAMPLES:

```
sage: M = matroids.catalog.K5dual(); M
M*(K5): Dual of 'Graphic matroid of rank 4 on 10 elements'
sage: M.is_3connected()
True
sage: G1 = M.automorphism_group()
sage: G2 = matroids.catalog.K5().automorphism_group()
sage: G1.is_isomorphic(G2)
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.K5dual(); M
M*(K5): Dual of 'Graphic matroid of rank 4 on 10 elements'
>>> M.is_3connected()
True
>>> G1 = M.automorphism_group()
>>> G2 = matroids.catalog.K5().automorphism_group()
>>> G1.is_isomorphic(G2)
True
```

REFERENCES:

[Oxl2011], p. 656.

sage.matroids.database_matroids.**KB12**(*groundset=None*)

Return the matroid $KB12$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.KB12(); M
KB12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.KB12(); M
KB12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**KF10**(*groundset=None*)

Return the matroid $KF10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.KF10(); M
KF10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.KF10(); M
KF10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**KP8**(*groundset=None*)

Return the matroid $KP8$.

An excluded minor for K_2-representable matroids. UPF is G. Self-dual. Uniquely $GF(5)$-representable. (An excluded minor for H_2-representable matroids.)

EXAMPLES:

```
sage: M = matroids.catalog.KP8(); M
KP8: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.KP8(); M
KP8: Quaternary matroid of rank 4 on 8 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**KQ9**(*groundset=None*)

Return the matroid $KQ9$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Has a *TQ8*-minor` (delete 6) and a *KP8*-minor (delete 8). UPF is $GF(4)$.

EXAMPLES:

```
sage: KQ9 = matroids.catalog.KQ9(); KQ9
KQ9: Quaternary matroid of rank 4 on 9 elements
sage: TQ8 = matroids.catalog.TQ8()
sage: TQ8.is_isomorphic(KQ9.delete(6))
True
sage: KP8 = matroids.catalog.KP8()
sage: KP8.is_isomorphic(KQ9.delete(8))
True
```

```
>>> from sage.all import *
>>> KQ9 = matroids.catalog.KQ9(); KQ9
KQ9: Quaternary matroid of rank 4 on 9 elements
>>> TQ8 = matroids.catalog.TQ8()
>>> TQ8.is_isomorphic(KQ9.delete(Integer(6)))
True
>>> KP8 = matroids.catalog.KP8()
>>> KP8.is_isomorphic(KQ9.delete(Integer(8)))
True
```

sage.matroids.database_matroids.**KR9**(*groundset=None*)

Return the matroid $KR9$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). In a DY-equivalence class of 4 matroids. Has a *KP8*-minor (delete 8). UPF is $GF(4)$.

EXAMPLES:

```
sage: KR9 = matroids.catalog.KR9(); KR9
KR9: Quaternary matroid of rank 4 on 9 elements
sage: KP8 = matroids.catalog.KP8()
sage: KP8.is_isomorphic(KR9.delete(8))
True
```

```
>>> from sage.all import *
>>> KR9 = matroids.catalog.KR9(); KR9
KR9: Quaternary matroid of rank 4 on 9 elements
>>> KP8 = matroids.catalog.KP8()
>>> KP8.is_isomorphic(KR9.delete(Integer(8)))
True
```

sage.matroids.database_matroids.**KT10**(*groundset=None*)

Return the matroid $KT10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.KT10(); M
KT10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.KT10(); M
KT10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**L8**(*groundset=None*)

Return the matroid L_8, represented as circuit closures.

The matroid L_8 is an 8-element matroid of rank-4. It is representable over all fields with at least five elements. It is a cube, yet it is not a tipless spike.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.L8(); M
L8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'h'}, {'a', 'b', 'f', 'g'}, {'a', 'c', 'e', 'g'},
     {'a', 'e', 'f', 'h'}, {'b', 'c', 'd', 'g'}, {'b', 'd', 'f', 'h'},
     {'c', 'd', 'e', 'h'}, {'d', 'e', 'f', 'g'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: M.equals(M.dual())
True
sage: M.is_valid()  # long time
True
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.L8(); M
L8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'h'}, {'a', 'b', 'f', 'g'}, {'a', 'c', 'e', 'g'},
     {'a', 'e', 'f', 'h'}, {'b', 'c', 'd', 'g'}, {'b', 'd', 'f', 'h'},
     {'c', 'd', 'e', 'h'}, {'d', 'e', 'f', 'g'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
>>> M.equals(M.dual())
True
>>> M.is_valid()  # long time
True
>>> M.automorphism_group().is_transitive()
True
```

Every single-element contraction is isomorphic to the free extension of $M(K_4)$:

```
sage: K4 = matroids.catalog.K4(range(6))
sage: Bext = [list(b) for b in K4.bases()] + [list(I)+[6] for I in
....:                                         K4.independent_sets(2)]
```

(continues on next page)

(continued from previous page)

```
sage: K4ext = Matroid(bases=Bext)
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: M.contract(e).is_isomorphic(K4ext)
True
```

```
>>> from sage.all import *
>>> K4 = matroids.catalog.K4(range(Integer(6)))
>>> Bext = [list(b) for b in K4.bases()] + [list(I)+[Integer(6)] for I in
...                                         K4.independent_sets(Integer(2))]
>>> K4ext = Matroid(bases=Bext)
>>> import random
>>> e = random.choice(list(M.groundset()))
>>> M.contract(e).is_isomorphic(K4ext)
True
```

REFERENCES:

[Oxl2011], p. 648.

sage.matroids.database_matroids.**LP8**(*groundset=None*)

Return the matroid $LP8$.

An excluded minor for G- and K_2-representable matroids. Self-dual. UPF is W. (Also an excluded minor for H_2- and $GF(5)$-representable matroids.)

EXAMPLES:

```
sage: M = matroids.catalog.LP8(); M
LP8: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.LP8(); M
LP8: Quaternary matroid of rank 4 on 8 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**M8591**(*groundset=None*)

Return the matroid $M8591$.

An excluded minor for K_2-representable matroids. A $Y - \delta$ exchange on the unique triad gives $A9$. The UPF is P_4.

EXAMPLES:

```
sage: M = matroids.catalog.M8591(); M
M8591: Quaternary matroid of rank 4 on 9 elements
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.M8591(); M
```

(continues on next page)

(continued from previous page)

```
M8591: Quaternary matroid of rank 4 on 9 elements
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**N1**(*groundset='abcdefghij'*)

Return the matroid N_1, represented over $\mathbf{F}_3$.

N_1 is an excluded minor for the dyadic matroids.

EXAMPLES:

```
sage: M = matroids.catalog.N1(); M
N1: Ternary matroid of rank 5 on 10 elements, type 0+
sage: M.is_field_isomorphic(M.dual())
True
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N1(); M
N1: Ternary matroid of rank 5 on 10 elements, type 0+
>>> M.is_field_isomorphic(M.dual())
True
>>> M.is_valid()
True
```

REFERENCES:

[Oxl2011], p. 554.

sage.matroids.database_matroids.**N2**(*groundset='abcdefghijkl'*)

Return the matroid N_2, represented over $\mathbf{F}_3$.

N_2 is an excluded minor for the dyadic matroids.

EXAMPLES:

```
sage: M = matroids.catalog.N2(); M
N2: Ternary matroid of rank 6 on 12 elements, type 0+
sage: M.is_field_isomorphic(M.dual())
True
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N2(); M
N2: Ternary matroid of rank 6 on 12 elements, type 0+
>>> M.is_field_isomorphic(M.dual())
True
>>> M.is_valid()
True
```

REFERENCES:

[Oxl2011], p. 554.

sage.matroids.database_matroids.**N3**(*groundset=None*)

Return the matroid $N3$.

An excluded minor for dyadic matroids (and $GF(5)$-representable matroids). UPF is $GF(3)$. 4- (but not 5-) connected. Self-dual.

EXAMPLES:

```
sage: N3 = matroids.catalog.N3(); N3
N3: Ternary matroid of rank 7 on 14 elements, type 0+
sage: N3.is_isomorphic(N3.dual())
True
sage: N3.is_kconnected(4)
True
sage: N3.is_kconnected(5)
False
```

```
>>> from sage.all import *
>>> N3 = matroids.catalog.N3(); N3
N3: Ternary matroid of rank 7 on 14 elements, type 0+
>>> N3.is_isomorphic(N3.dual())
True
>>> N3.is_kconnected(Integer(4))
True
>>> N3.is_kconnected(Integer(5))
False
```

sage.matroids.database_matroids.**N3pp**(*groundset=None*)

Return the matroid $N3pp$.

An excluded minor for K_2-representable matroids. Self-dual. Obtained by relaxing the two complementary circuit-hyperplanes of *N4*. Not P_4-representable, but O-representable, and hence representable over all fields of size at least four.

EXAMPLES:

```
sage: M = matroids.catalog.N3pp(); M
N3=: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N3pp(); M
N3=: Quaternary matroid of rank 7 on 14 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**N4**(*groundset=None*)

Return the matroid $N4$.

An excluded minor for dyadic matroids (and $GF(5)$-representable matroids). UPF is $GF(3)$. 4- (but not 5-) connected. Self-dual.

EXAMPLES:

```
sage: N4 = matroids.catalog.N4(); N4
N4: Ternary matroid of rank 8 on 16 elements, type 0+
sage: N4.is_isomorphic(N4.dual())
True
sage: N4.is_kconnected(4)
True
sage: N4.is_kconnected(5)
False
```

```
>>> from sage.all import *
>>> N4 = matroids.catalog.N4(); N4
N4: Ternary matroid of rank 8 on 16 elements, type 0+
>>> N4.is_isomorphic(N4.dual())
True
>>> N4.is_kconnected(Integer(4))
True
>>> N4.is_kconnected(Integer(5))
False
```

sage.matroids.database_matroids.**NestOfTwistedCubes**(*groundset=None*)

Return the NestOfTwistedCubes matroid.

A matroid with no $U(2,4)$-detachable pairs (only $\{e_i, f_i\}$ pairs are detachable).

EXAMPLES:

```
sage: M = matroids.catalog.NestOfTwistedCubes(); M
NestOfTwistedCubes: Matroid of rank 6 on 12 elements with 57 circuits
sage: M.is_3connected()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NestOfTwistedCubes(); M
NestOfTwistedCubes: Matroid of rank 6 on 12 elements with 57 circuits
>>> M.is_3connected()
True
```

sage.matroids.database_matroids.**NonDesargues**(*groundset=None*)

Return the NonDesargues matroid.

The NonDesargues matroid is a 10-element matroid of rank-3. It is not representable over any division ring. It is not graphic, not cographic, and not regular.

EXAMPLES:

```
sage: M = matroids.catalog.NonDesargues(); M
NonDesargues: Matroid of rank 3 on 10 elements with 9 nonspanning circuits
sage: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonDesargues(); M
```

(continues on next page)

(continued from previous page)

```
NonDesargues: Matroid of rank 3 on 10 elements with 9 nonspanning circuits
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 657.

sage.matroids.database_matroids.**NonFano**(*groundset='abcdefg'*)

Return the non-Fano matroid, represented over $GF(3)$.

The non-Fano matroid, or F_7^-, is a 7-element matroid of rank-3. It is representable over a field if and only if that field has characteristic other than two. It is the unique relaxation of F_7.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.NonFano(); M
NonFano: Ternary matroid of rank 3 on 7 elements, type 0-
sage: setprint(M.nonbases())
[{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'},
 {'b', 'c', 'd'}, {'b', 'e', 'g'}, {'c', 'f', 'g'}]
sage: M.delete('f').is_isomorphic(matroids.CompleteGraphic(4))
True
sage: M.delete('g').is_isomorphic(matroids.CompleteGraphic(4))
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.NonFano(); M
NonFano: Ternary matroid of rank 3 on 7 elements, type 0-
>>> setprint(M.nonbases())
[{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'},
 {'b', 'c', 'd'}, {'b', 'e', 'g'}, {'c', 'f', 'g'}]
>>> M.delete('f').is_isomorphic(matroids.CompleteGraphic(Integer(4)))
True
>>> M.delete('g').is_isomorphic(matroids.CompleteGraphic(Integer(4)))
False
```

REFERENCES:

[Oxl2011], p. 643-4.

sage.matroids.database_matroids.**NonFanoDual**(*groundset='abcdefg'*)

Return the dual of the non-Fano matroid.

$(F_7^-)^*$ is a 7-element matroid of rank-3. Every single-element contraction of $(F_7)^*$ is isomorphic to $M(K_4)$ or $\mathcal{W}^3$.

EXAMPLES:

```
sage: M = matroids.catalog.NonFanoDual(); M
NonFano*: Ternary matroid of rank 4 on 7 elements, type 0-
```

(continues on next page)

(continued from previous page)

```
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonFanoDual(); M
NonFano*: Ternary matroid of rank 4 on 7 elements, type 0-
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Every single-element contraction of $(F_7^-)^*$ is isomorphic to $M(K_4)$ or $\mathcal{W}^3$:

```
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: N = M.contract(e)
sage: K4 = matroids.catalog.K4()
sage: W3 = matroids.catalog.Whirl3()
sage: N.is_isomorphic(K4) or N.is_isomorphic(W3)
True
```

```
>>> from sage.all import *
>>> import random
>>> e = random.choice(list(M.groundset()))
>>> N = M.contract(e)
>>> K4 = matroids.catalog.K4()
>>> W3 = matroids.catalog.Whirl3()
>>> N.is_isomorphic(K4) or N.is_isomorphic(W3)
True
```

REFERENCES:

[Oxl2011], p. 643-4.

sage.matroids.database_matroids.**NonPappus**(*groundset=None*)

Return the non-Pappus matroid.

The non-Pappus matroid is a 9-element matroid of rank-3. It is not representable over any commutative field. It is the unique relaxation of the Pappus matroid.

EXAMPLES:

```
sage: M = matroids.catalog.NonPappus(); M
NonPappus: Matroid of rank 3 on 9 elements with 8 nonspanning circuits
sage: NSC = set([('a', 'b', 'c'), ('a', 'e', 'i'), ('a', 'f', 'h'),
....:            ('b', 'd', 'i'), ('b', 'f', 'g'), ('c', 'd', 'h'),
....:            ('c', 'e', 'g'), ('g', 'h', 'i')])
sage: NSC == set(tuple(sorted(C)) for C in M.nonspanning_circuits())
True
sage: M.is_dependent(['d', 'e', 'f'])
False
sage: M.is_valid() and M.is_paving()
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonPappus(); M
NonPappus: Matroid of rank 3 on 9 elements with 8 nonspanning circuits
>>> NSC = set([('a', 'b', 'c'), ('a', 'e', 'i'), ('a', 'f', 'h'),
...            ('b', 'd', 'i'), ('b', 'f', 'g'), ('c', 'd', 'h'),
...            ('c', 'e', 'g'), ('g', 'h', 'i')])
>>> NSC == set(tuple(sorted(C)) for C in M.nonspanning_circuits())
True
>>> M.is_dependent(['d', 'e', 'f'])
False
>>> M.is_valid() and M.is_paving()
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 655.

sage.matroids.database_matroids.**NonVamos**(*groundset=None*)

Return the non-$V\acute{a}mos$ matroid.

The non-$V\acute{a}mos$ matroid, or V_8^+ is an 8-element matroid of rank 4. It is a tightening of the $V\acute{a}mos$ matroid. It is representable over some field.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.NonVamos(); M
NonVamos: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
     {'c', 'd', 'e', 'f'}, {'c', 'd', 'g', 'h'}, {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: setprint(M.nonbases())
[{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
 {'c', 'd', 'e', 'f'}, {'c', 'd', 'g', 'h'}, {'e', 'f', 'g', 'h'}]
sage: M.is_dependent(['c', 'd', 'g', 'h'])
True
sage: M.is_valid()  # long time
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.NonVamos(); M
NonVamos: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
     {'c', 'd', 'e', 'f'}, {'c', 'd', 'g', 'h'}, {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
>>> setprint(M.nonbases())
[{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
 {'c', 'd', 'e', 'f'}, {'c', 'd', 'g', 'h'}, {'e', 'f', 'g', 'h'}]
>>> M.is_dependent(['c', 'd', 'g', 'h'])
True
>>> M.is_valid()  # long time
True
```

REFERENCES:

[Oxl2011], p. 72, 84.

sage.matroids.database_matroids.**NotP8**(*groundset='abcdefgh'*)

Return the matroid `NotP8`.

EXAMPLES:

```
sage: M = matroids.catalog.P8()
sage: N = matroids.catalog.NotP8(); N
NotP8: Ternary matroid of rank 4 on 8 elements, type 0-
sage: M.is_isomorphic(N)
False
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.P8()
>>> N = matroids.catalog.NotP8(); N
NotP8: Ternary matroid of rank 4 on 8 elements, type 0-
>>> M.is_isomorphic(N)
False
>>> M.is_valid()
True
```

REFERENCES:

[Oxl1992], p.512 (the first edition).

sage.matroids.database_matroids.**O7**(*groundset='abcdefg'*)

Return the matroid O_7, represented over $GF(3)$.

The matroid O_7 is a 7-element matroid of rank-3. It is representable over a field if and only if that field has at least three elements. It is obtained by freely adding a point to any line of $M(K_4)$.

EXAMPLES:

```
sage: M = matroids.catalog.O7(); M
O7: Ternary matroid of rank 3 on 7 elements, type 0+
sage: M.delete('e').is_isomorphic(matroids.CompleteGraphic(4))
True
sage: M.tutte_polynomial()
y^4 + x^3 + x*y^2 + 3*y^3 + 4*x^2 + 5*x*y + 5*y^2 + 4*x + 4*y
```

```
>>> from sage.all import *
>>> M = matroids.catalog.O7(); M
O7: Ternary matroid of rank 3 on 7 elements, type 0+
>>> M.delete('e').is_isomorphic(matroids.CompleteGraphic(Integer(4)))
True
>>> M.tutte_polynomial()
y^4 + x^3 + x*y^2 + 3*y^3 + 4*x^2 + 5*x*y + 5*y^2 + 4*x + 4*y
```

REFERENCES:

[Oxl2011], p. 644.

sage.matroids.database_matroids.**OW14**(*groundset=None*)

Return the matroid $OW14$.

An excluded minor for P_4-representable matroids. Self-dual. UPF is $Orthrus$.

EXAMPLES:

```
sage: M = matroids.catalog.OW14(); M
OW14: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.OW14(); M
OW14: Quaternary matroid of rank 7 on 14 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**P6**(*groundset=None*)

Return the matroid P_6, represented as circuit closures.

The matroid P_6 is a 6-element matroid of rank-3. It is representable over a field if and only if that field has at least five elements. It is the unique relaxation of Q_6. It is an excluded minor for the class of quaternary matroids.

EXAMPLES:

```
sage: M = matroids.catalog.P6(); M
P6: Matroid of rank 3 on 6 elements with circuit-closures
{2: {{'a', 'b', 'c'}}, 3: {{'a', 'b', 'c', 'd', 'e', 'f'}}}
sage: len(set(M.nonspanning_circuits()).difference(M.nonbases())) == 0
True
sage: Matroid(matrix=random_matrix(GF(4, 'a'), ncols=5, nrows=5)).has_minor(M)
False
sage: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.P6(); M
P6: Matroid of rank 3 on 6 elements with circuit-closures
{2: {{'a', 'b', 'c'}}, 3: {{'a', 'b', 'c', 'd', 'e', 'f'}}}
>>> len(set(M.nonspanning_circuits()).difference(M.nonbases())) == Integer(0)
True
>>> Matroid(matrix=random_matrix(GF(Integer(4), 'a'), ncols=Integer(5),␣
↪nrows=Integer(5))).has_minor(M)
False
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 641-2.

sage.matroids.database_matroids.**P7**(*groundset='abcdefg'*)

Return the matroid P_7, represented over $GF(3)$.

The matroid P_7 is a 7-element matroid of rank-3. It is representable over a field if and only if that field has at least three elements. It is one of two ternary 3-spikes, with the other being F_7^-.

EXAMPLES:

```
sage: M = matroids.catalog.P7(); M
P7: Ternary matroid of rank 3 on 7 elements, type 1+
sage: M.whitney_numbers2()
[1, 7, 11, 1]
sage: M.has_minor(matroids.CompleteGraphic(4))
False
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.P7(); M
P7: Ternary matroid of rank 3 on 7 elements, type 1+
>>> M.whitney_numbers2()
[1, 7, 11, 1]
>>> M.has_minor(matroids.CompleteGraphic(Integer(4)))
False
>>> M.is_valid()
True
```

REFERENCES:

[Oxl2011], p. 644-5.

sage.matroids.database_matroids.**P8**(*groundset='abcdefgh'*)

Return the matroid P_8, represented over $GF(3)$.

The matroid P_8 is an 8-element matroid of rank-4. It is uniquely representable over all fields of characteristic other than two. It is an excluded minor for all fields of characteristic two with four or more elements.

EXAMPLES:

```
sage: M = matroids.catalog.P8(); M
P8: Ternary matroid of rank 4 on 8 elements, type 2+
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: Matroid(matrix=random_matrix(GF(4, 'a'), ncols=5, nrows=5)).has_minor(M)
False
sage: M.bicycle_dimension()
2
```

```
>>> from sage.all import *
>>> M = matroids.catalog.P8(); M
P8: Ternary matroid of rank 4 on 8 elements, type 2+
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> Matroid(matrix=random_matrix(GF(Integer(4), 'a'), ncols=Integer(5),␣
↪nrows=Integer(5))).has_minor(M)
False
```

(continues on next page)

(continued from previous page)

```
>>> M.bicycle_dimension()
2
```

REFERENCES:

[Oxl2011], p. 650-1.

sage.matroids.database_matroids.**P8p**(*groundset=None*)

Return the matroid $P8^-$.

$P8^-$ is obtained by relaxing one of the disjoint circuit-hyperplanes of *P8*. An excluded minor for 2-regular matroids. UPF is K_2. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.P8p(); M
P8-: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.P8p(); M
P8-: Quaternary matroid of rank 4 on 8 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**P8pp**(*groundset=None*)

Return the matroid $P_8^=$, represented as circuit closures.

The matroid $P_8^=$ is an 8-element matroid of rank-4. It can be obtained from P_8 by relaxing the unique pair of disjoint circuit-hyperplanes. It is an excluded minor for $GF(4)$-representability. It is representable over all fields with at least five elements.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.catalog.P8pp(); M
P8'': Matroid of rank 4 on 8 elements with 8 nonspanning circuits
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: len(get_nonisomorphic_matroids([M.contract(i) for i in M.groundset()]))
1
sage: M.is_valid() and M.is_paving()
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.P8pp(); M
P8'': Matroid of rank 4 on 8 elements with 8 nonspanning circuits
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> len(get_nonisomorphic_matroids([M.contract(i) for i in M.groundset()]))
1
>>> M.is_valid() and M.is_paving()
True
```

REFERENCES:

[Oxl2011], p. 651.

sage.matroids.database_matroids.**P9**(*groundset='abcdefghi'*)

Return the matroid P_9.

EXAMPLES:

```
sage: M = matroids.catalog.P9(); M
P9: Binary matroid of rank 4 on 9 elements, type (1, 1)
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.P9(); M
P9: Binary matroid of rank 4 on 9 elements, type (1, 1)
>>> M.is_valid()
True
```

REFERENCES:

This is the matroid referred to as P_9 by Oxley in his paper "The binary matroids with no 4-wheel minor", [Oxl1987].

sage.matroids.database_matroids.**PG**(*n*, *q*, *x=None*, *groundset=None*)

Return the projective geometry of dimension n over the finite field of order q.

INPUT:

- `n` – positive integer; the dimension of the projective space. This is one less than the rank of the resulting matroid.
- `q` – positive integer that is a prime power; the order of the finite field
- `x` – string (default: `None`); the name of the generator of a non-prime field, used for non-prime fields. If not supplied, `'x'` is used.
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: a linear matroid whose elements are the points of $PG(n, q)$

EXAMPLES:

```
sage: M = matroids.PG(2, 2)
sage: M.is_isomorphic(matroids.catalog.Fano())
True
sage: matroids.PG(5, 4, 'z').size() == (4^6 - 1) / (4 - 1)
True
sage: M = matroids.PG(4, 7); M
PG(4, 7): Linear matroid of rank 5 on 2801 elements represented over the Finite␣
↪Field
 of size 7
```

```
>>> from sage.all import *
>>> M = matroids.PG(Integer(2), Integer(2))
>>> M.is_isomorphic(matroids.catalog.Fano())
True
>>> matroids.PG(Integer(5), Integer(4), 'z').size() == (Integer(4)**Integer(6) -␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)) / (Integer(4) - Integer(1))
True
>>> M = matroids.PG(Integer(4), Integer(7)); M
PG(4, 7): Linear matroid of rank 5 on 2801 elements represented over the Finite␣
↪Field
 of size 7
```

REFERENCES:

[Oxl2011], p. 660.

sage.matroids.database_matroids.**PG23**(*groundset=None*)

Return the matroid $PG23$.

The second smallest projective plane. Not graphic, not cographic, not regular, not near-regular.

EXAMPLES:

```
sage: M = matroids.catalog.PG23(); M
PG(2, 3): Ternary matroid of rank 3 on 13 elements, type 3+
sage: M.is_3connected()
True
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.PG23(); M
PG(2, 3): Ternary matroid of rank 3 on 13 elements, type 3+
>>> M.is_3connected()
True
>>> M.automorphism_group().is_transitive()
True
```

REFERENCES:

[Oxl2011], p. 659.

sage.matroids.database_matroids.**PK10**(*groundset=None*)

Return the matroid $PK10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Not self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.PK10(); M
PK10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.PK10(); M
PK10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**PP10**(*groundset=None*)

Return the matroid $PP10$.

An excluded minor for P_4-representable matroids. UPF is $U_1^{(2)}$. Has a *TQ8*-minor (e.g. delete 'a' and contract 'e') and a *PP9* (and hence *P8p*) minor (contract 'x').

EXAMPLES:

```
sage: PP10 = matroids.catalog.PP10(); PP10
PP10: Quaternary matroid of rank 5 on 10 elements
sage: M = PP10.delete('a').contract('e')
sage: M.is_isomorphic(matroids.catalog.TQ8())
True
sage: M = PP10.contract('x')
sage: M.is_isomorphic(matroids.catalog.PP9())
True
```

```
>>> from sage.all import *
>>> PP10 = matroids.catalog.PP10(); PP10
PP10: Quaternary matroid of rank 5 on 10 elements
>>> M = PP10.delete('a').contract('e')
>>> M.is_isomorphic(matroids.catalog.TQ8())
True
>>> M = PP10.contract('x')
>>> M.is_isomorphic(matroids.catalog.PP9())
True
```

sage.matroids.database_matroids.**PP9**(*groundset=None*)

Return the matroid $PP9$.

An excluded minor for K_2-representable matroids. A single-element extension of $P8^-$. The UPF is P_4. Has a *P8p*-minor (delete z). Uniquely $GF(5)$-representable. (An excluded minor for H_2-representable matroids.)

EXAMPLES:

```
sage: P8p = matroids.catalog.P8p()
sage: PP9 = matroids.catalog.PP9(); PP9
PP9: Quaternary matroid of rank 4 on 9 elements
sage: for M in P8p.extensions():
....:     if M.is_isomorphic(PP9):
....:         print(True)
....:         break
True
sage: M = PP9.delete('z')
sage: M.is_isomorphic(P8p)
True
```

```
>>> from sage.all import *
>>> P8p = matroids.catalog.P8p()
>>> PP9 = matroids.catalog.PP9(); PP9
PP9: Quaternary matroid of rank 4 on 9 elements
>>> for M in P8p.extensions():
...     if M.is_isomorphic(PP9):
...         print(True)
...         break
```

(continues on next page)

(continued from previous page)

```
True
>>> M = PP9.delete('z')
>>> M.is_isomorphic(P8p)
True
```

sage.matroids.database_matroids.**Pappus**(*groundset=None*)

Return the Pappus matroid.

The Pappus matroid is a 9-element matroid of rank-3. It is representable over a field if and only if that field either has 4 elements or more than 7 elements. It is an excluded minor for the class of GF(5)-representable matroids.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Pappus(); M
Pappus: Matroid of rank 3 on 9 elements with 9 nonspanning circuits
sage: setprint(M.nonspanning_circuits())
[{'a', 'b', 'c'}, {'a', 'e', 'i'}, {'a', 'f', 'h'}, {'b', 'd', 'i'},
 {'b', 'f', 'g'}, {'c', 'd', 'h'}, {'c', 'e', 'g'}, {'d', 'e', 'f'},
 {'g', 'h', 'i'}]
sage: M.is_dependent(['d', 'e', 'f'])
True
sage: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Pappus(); M
Pappus: Matroid of rank 3 on 9 elements with 9 nonspanning circuits
>>> setprint(M.nonspanning_circuits())
[{'a', 'b', 'c'}, {'a', 'e', 'i'}, {'a', 'f', 'h'}, {'b', 'd', 'i'},
 {'b', 'f', 'g'}, {'c', 'd', 'h'}, {'c', 'e', 'g'}, {'d', 'e', 'f'},
 {'g', 'h', 'i'}]
>>> M.is_dependent(['d', 'e', 'f'])
True
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
True
```

REFERENCES:

[Oxl2011], p. 655.

sage.matroids.database_matroids.**Psi**(*r*, *groundset=None*)

Return the matroid Ψ_r.

The rank-r free swirl; defined for all $r \geq 3$.

INPUT:

- `r` – integer ($r \geq 3$); the rank of the matroid
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: matroid (Ψ_r)

EXAMPLES:

```
sage: matroids.Psi(7)
Psi_7: Matroid of rank 7 on 14 elements with 105 nonspanning circuits
```

```
>>> from sage.all import *
>>> matroids.Psi(Integer(7))
Psi_7: Matroid of rank 7 on 14 elements with 105 nonspanning circuits
```

The matroid Ψ_r is 3-connected but, for all $r \geq 4$, not 4-connected:

```
sage: M = matroids.Psi(3)
sage: M.is_4connected()
True
sage: import random
sage: r = random.choice(range(4, 8))
sage: M = matroids.Psi(r)
sage: M.is_4connected()
False
```

```
>>> from sage.all import *
>>> M = matroids.Psi(Integer(3))
>>> M.is_4connected()
True
>>> import random
>>> r = random.choice(range(Integer(4), Integer(8)))
>>> M = matroids.Psi(r)
>>> M.is_4connected()
False
```

$\Psi_3 \cong U_{3,6}$:

```
sage: M = matroids.Psi(3)
sage: M.is_isomorphic(matroids.catalog.U36())
True
```

```
>>> from sage.all import *
>>> M = matroids.Psi(Integer(3))
>>> M.is_isomorphic(matroids.catalog.U36())
True
```

It is identically self-dual with a transitive automorphism group:

```
sage: M = matroids.Psi(r)
sage: M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()  # long time
True
```

```
>>> from sage.all import *
>>> M = matroids.Psi(r)
```

(continues on next page)

(continued from previous page)

```
>>> M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()  # long time
True
```

REFERENCES:

[Oxl2011], p. 664.

sage.matroids.database_matroids.**Q10**(*groundset='abcdefghij'*)

Return the matroid Q_{10}, represented over $\mathbf{F}_4$.

Q_{10} is a 10-element, rank-5, self-dual matroid. It is representable over $\mathbf{F}_3$ and $\mathbf{F}_4$, and hence is a sixth-roots-of-unity matroid. Q_{10} is a splitter for the class of sixth-root-of-unity matroids.

EXAMPLES:

```
sage: M = matroids.catalog.Q10(); M
Q10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Q10(); M
Q10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
>>> M.is_valid()
True
```

Check the splitter property. By Seymour's Theorem, and using self-duality, we only need to check that all 3-connected single-element extensions have an excluded minor for sixth-roots-of-unity. The only excluded minors that are quaternary are $U_{2,5}, U_{3,5}, F_7, F_7^*$. As it happens, it suffices to check for $U_{2,5}$:

```
sage: S = matroids.catalog.Q10().linear_extensions(simple=True)
sage: [M for M in S if not M.has_line_minor(5)]
[]
```

```
>>> from sage.all import *
>>> S = matroids.catalog.Q10().linear_extensions(simple=True)
>>> [M for M in S if not M.has_line_minor(Integer(5))]
[]
```

sage.matroids.database_matroids.**Q6**(*groundset='abcdef'*)

Return the matroid Q_6, represented over $GF(4)$.

The matroid Q_6 is a 6-element matroid of rank-3. It is representable over a field if and only if that field has at least four elements. It is the unique relaxation of the rank-3 whirl.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Q6(); M
Q6: Quaternary matroid of rank 3 on 6 elements
sage: setprint(M.hyperplanes())
[{'a', 'b', 'd'}, {'a', 'c'}, {'a', 'e'}, {'a', 'f'}, {'b', 'c', 'e'},
 {'b', 'f'}, {'c', 'd'}, {'c', 'f'}, {'d', 'e'}, {'d', 'f'},
 {'e', 'f'}]
sage: M.nonspanning_circuits() == M.noncospanning_cocircuits()
False
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Q6(); M
Q6: Quaternary matroid of rank 3 on 6 elements
>>> setprint(M.hyperplanes())
[{'a', 'b', 'd'}, {'a', 'c'}, {'a', 'e'}, {'a', 'f'}, {'b', 'c', 'e'},
 {'b', 'f'}, {'c', 'd'}, {'c', 'f'}, {'d', 'e'}, {'d', 'f'},
 {'e', 'f'}]
>>> M.nonspanning_circuits() == M.noncospanning_cocircuits()
False
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 641.

sage.matroids.database_matroids.**Q8**(*groundset=None*)

Return the matroid Q_8, represented as circuit closures.

The matroid Q_8 is an 8-element matroid of rank-4. It is a smallest non-representable matroid.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Q8(); M
Q8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
     {'a', 'c', 'd', 'f'}, {'a', 'd', 'g', 'h'}, {'a', 'e', 'f', 'h'},
     {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'}, {'c', 'd', 'e', 'h'},
     {'c', 'f', 'g', 'h'}, {'d', 'e', 'f', 'g'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: setprint(M.flats(3))
[{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
 {'a', 'c', 'd', 'f'}, {'a', 'c', 'e'}, {'a', 'c', 'g'},
 {'a', 'd', 'g', 'h'}, {'a', 'e', 'f', 'h'}, {'a', 'e', 'g'},
 {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'}, {'b', 'd', 'f'},
 {'b', 'd', 'h'}, {'b', 'e', 'g'}, {'b', 'e', 'h'}, {'b', 'f', 'h'},
 {'b', 'g', 'h'}, {'c', 'd', 'e', 'h'}, {'c', 'e', 'g'},
 {'c', 'f', 'g', 'h'}, {'d', 'e', 'f', 'g'}, {'d', 'f', 'h'},
 {'e', 'g', 'h'}]
sage: M.is_valid()  # long time
```

(continues on next page)

(continued from previous page)

```
True
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Q8(); M
Q8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
     {'a', 'c', 'd', 'f'}, {'a', 'd', 'g', 'h'}, {'a', 'e', 'f', 'h'},
     {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'}, {'c', 'd', 'e', 'h'},
     {'c', 'f', 'g', 'h'}, {'d', 'e', 'f', 'g'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
>>> setprint(M.flats(Integer(3)))
[{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
 {'a', 'c', 'd', 'f'}, {'a', 'c', 'e'}, {'a', 'c', 'g'},
 {'a', 'd', 'g', 'h'}, {'a', 'e', 'f', 'h'}, {'a', 'e', 'g'},
 {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'}, {'b', 'd', 'f'},
 {'b', 'd', 'h'}, {'b', 'e', 'g'}, {'b', 'e', 'h'}, {'b', 'f', 'h'},
 {'b', 'g', 'h'}, {'c', 'd', 'e', 'h'}, {'c', 'e', 'g'},
 {'c', 'f', 'g', 'h'}, {'d', 'e', 'f', 'g'}, {'d', 'f', 'h'},
 {'e', 'g', 'h'}]
>>> M.is_valid()  # long time
True
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 647.

sage.matroids.database_matroids.**R10**(*groundset='abcdefghij'*)

Return the matroid R_{10}, represented over the regular partial field.

The NonDesargues matroid is a 10-element matroid of rank-5. It is the unique splitter for the class of regular matroids. It is the graft matroid of $K_{3,3}$ in which every vertex is coloured.

EXAMPLES:

```
sage: M = matroids.catalog.R10(); M
R10: Regular matroid of rank 5 on 10 elements with 162 bases
sage: cct = []
sage: for i in M.circuits():
....:     cct.append(len(i))
sage: Set(cct)
{4, 6}
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.is_valid()
True
```

(continues on next page)

(continued from previous page)

```
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R10(); M
R10: Regular matroid of rank 5 on 10 elements with 162 bases
>>> cct = []
>>> for i in M.circuits():
...     cct.append(len(i))
>>> Set(cct)
{4, 6}
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
True
```

Every single-element deletion is isomorphic to $M(K_{3,3})$, and every single-element contraction is isomorphic to $M^*(K_{3,3})$:

```
sage: K33 = matroids.catalog.K33()
sage: K33D = matroids.catalog.K33dual()
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: M.delete(e).is_isomorphic(K33)
True
sage: M.contract(e).is_isomorphic(K33D)
True
```

```
>>> from sage.all import *
>>> K33 = matroids.catalog.K33()
>>> K33D = matroids.catalog.K33dual()
>>> import random
>>> e = random.choice(list(M.groundset()))
>>> M.delete(e).is_isomorphic(K33)
True
>>> M.contract(e).is_isomorphic(K33D)
True
```

Check the splitter property:

```
sage: matroids.catalog.R10().linear_extensions(simple=True)
[]
```

```
>>> from sage.all import *
>>> matroids.catalog.R10().linear_extensions(simple=True)
[]
```

REFERENCES:

[Oxl2011], p. 656-7.

sage.matroids.database_matroids.**R12**(*groundset='abcdefghijkl'*)

Return the matroid R_{12}, represented over the regular partial field.

The matroid R_{12} is a 12-element regular matroid of rank-6. It induces a 3-separation in its 3-connected majors within the class of regular matroids. An excluded minor for the class of graphic or cographic matroids.

EXAMPLES:

```
sage: M = matroids.catalog.R12(); M
R12: Regular matroid of rank 6 on 12 elements with 441 bases
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R12(); M
R12: Regular matroid of rank 6 on 12 elements with 441 bases
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 657.

sage.matroids.database_matroids.**R6**(*groundset='abcdef'*)

Return the matroid R_6, represented over $GF(3)$.

The matroid R_6 is a 6-element matroid of rank-3. It is representable over a field if and only if that field has at least three elements. It is isomorphic to the 2-sum of two copies of $U_{2,4}$.

EXAMPLES:

```
sage: M = matroids.catalog.R6(); M
R6: Ternary matroid of rank 3 on 6 elements, type 2+
sage: M.equals(M.dual())
True
sage: M.is_connected()
True
sage: M.is_3connected()
False
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R6(); M
R6: Ternary matroid of rank 3 on 6 elements, type 2+
>>> M.equals(M.dual())
True
>>> M.is_connected()
```

(continues on next page)

(continued from previous page)

```
True
>>> M.is_3connected()
False
>>> M.automorphism_group().is_transitive()
True
```

REFERENCES:

[Oxl2011], p. 642.

sage.matroids.database_matroids.**R8**(*groundset='abcdefgh'*)

Return the matroid R_8, represented over $GF(3)$.

The matroid R_8 is an 8-element matroid of rank-4. It is representable over a field if and only if the characteristic of that field is not two. It is the real affine cube.

EXAMPLES:

```
sage: M = matroids.catalog.R8(); M
R8: Ternary matroid of rank 4 on 8 elements, type 0+
sage: M.contract(M.groundset_list()[randrange(0,
....:             8)]).is_isomorphic(matroids.catalog.NonFano())
True
sage: M.equals(M.dual())
True
sage: M.has_minor(matroids.catalog.Fano())
False
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R8(); M
R8: Ternary matroid of rank 4 on 8 elements, type 0+
>>> M.contract(M.groundset_list()[randrange(Integer(0),
...             Integer(8))]).is_isomorphic(matroids.catalog.NonFano())
True
>>> M.equals(M.dual())
True
>>> M.has_minor(matroids.catalog.Fano())
False
>>> M.automorphism_group().is_transitive()
True
```

Every single-element deletion is isomorphic to (F_7^-)^* and every single-element contraction is isomorphic to F_7^-:

```
sage: F7m = matroids.catalog.NonFano()
sage: F7mD = matroids.catalog.NonFanoDual()
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: M.delete(e).is_isomorphic(F7mD)
True
sage: M.contract(e).is_isomorphic(F7m)
True
```

```
>>> from sage.all import *
>>> F7m = matroids.catalog.NonFano()
>>> F7mD = matroids.catalog.NonFanoDual()
>>> import random
>>> e = random.choice(list(M.groundset()))
>>> M.delete(e).is_isomorphic(F7mD)
True
>>> M.contract(e).is_isomorphic(F7m)
True
```

REFERENCES:

[Oxl2011], p. 646.

sage.matroids.database_matroids.**R9**(*groundset=None*)

Return the matroid R_9.

The ternary Reid geometry. The only 9-element rank-3 simple ternary matroids are R_9, $Q_3(GF(3)^imes)$, and $AG(2,3)$. It is not graphic, not cographic, and not regular.

EXAMPLES:

```
sage: M = matroids.catalog.R9(); M
R9: Matroid of rank 3 on 9 elements with 15 nonspanning circuits
sage: M.is_valid()
True
sage: len(M.nonspanning_circuits())
15
sage: M.is_simple() and M.is_ternary()
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R9(); M
R9: Matroid of rank 3 on 9 elements with 15 nonspanning circuits
>>> M.is_valid()
True
>>> len(M.nonspanning_circuits())
15
>>> M.is_simple() and M.is_ternary()
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 654.

sage.matroids.database_matroids.**R9A**(*groundset=None*)

Return the matroid R_9^A.

The matroid R_9^A is not representable over any field, yet none of the cross-ratios in its Tuttegroup equal 1. It is one of the 4 matroids on at most 9 elements with this property, the others being $R_9^{A^*}$, R_9^B and $R_9^{B^*}$.

EXAMPLES:

```
sage: M = matroids.catalog.R9A(); M
R9A: Matroid of rank 4 on 9 elements with 13 nonspanning circuits
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R9A(); M
R9A: Matroid of rank 4 on 9 elements with 13 nonspanning circuits
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**R9B**(*groundset=None*)

Return the matroid R_9^B.

The matroid R_9^B is not representable over any field, yet none of the cross-ratios in its Tuttegroup equal 1. It is one of the 4 matroids on at most 9 elements with this property, the others being R_9^{B*}, R_9^A and R_9^{A*}.

EXAMPLES:

```
sage: M = matroids.catalog.R9B(); M
R9B: Matroid of rank 4 on 9 elements with 13 nonspanning circuits
sage: M.is_valid() and M.is_paving()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R9B(); M
R9B: Matroid of rank 4 on 9 elements with 13 nonspanning circuits
>>> M.is_valid() and M.is_paving()
True
```

sage.matroids.database_matroids.**RelaxedNonFano**(*groundset=None*)

Return the relaxed NonFano matroid.

An excluded minor for 2-regular matroids. UPF is K_2.

EXAMPLES:

```
sage: M = matroids.catalog.RelaxedNonFano(); M
F7=: Quaternary matroid of rank 3 on 7 elements
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.RelaxedNonFano(); M
F7=: Quaternary matroid of rank 3 on 7 elements
>>> M.is_valid()
True
```

sage.matroids.database_matroids.**S8**(*groundset='abcdefgh'*)

Return the matroid S_8, represented over $GF(2)$.

The matroid S_8 is an 8-element matroid of rank-4. It is representable over a field if and only if that field has characteristic two. It is the unique deletion of a non-tip element from the binary 4-spike.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.catalog.S8(); M
S8: Binary matroid of rank 4 on 8 elements, type (2, 0)
sage: M.contract('d').is_isomorphic(matroids.catalog.Fano())
True
sage: M.delete('d').is_isomorphic(matroids.catalog.FanoDual())
False
sage: M.delete('h').is_isomorphic(matroids.catalog.FanoDual())
True
sage: M.is_graphic()
False
sage: D = get_nonisomorphic_matroids(
....:       list(matroids.catalog.Fano().linear_coextensions(cosimple=True)))
sage: len(D)
2
sage: [N.is_isomorphic(M) for N in D]
[...True...]
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.S8(); M
S8: Binary matroid of rank 4 on 8 elements, type (2, 0)
>>> M.contract('d').is_isomorphic(matroids.catalog.Fano())
True
>>> M.delete('d').is_isomorphic(matroids.catalog.FanoDual())
False
>>> M.delete('h').is_isomorphic(matroids.catalog.FanoDual())
True
>>> M.is_graphic()
False
>>> D = get_nonisomorphic_matroids(
...       list(matroids.catalog.Fano().linear_coextensions(cosimple=True)))
>>> len(D)
2
>>> [N.is_isomorphic(M) for N in D]
[...True...]
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 648.

sage.matroids.database_matroids.**Sp8**(*groundset=None*)

Return the matroid $Sp8$.

An excluded minor for G- and K_2-representable matroids. UPF is $U_1^{(2)}$. Self-dual. (An excluded minor for H_2- and $GF(5)$-representable matroids.)

EXAMPLES:

```
sage: M = matroids.catalog.Sp8(); M
Sp8: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Sp8(); M
Sp8: Quaternary matroid of rank 4 on 8 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**Sp8pp**(*groundset=None*)

Return the matroid $Sp8 =$.

An excluded minor for G- and K_2-representable matroids. UPF is $(GF(2)(a, b), < a, b, a+1, b+1, ab+a+b >)$. Self-dual. (An excluded minor for H_2- and $GF(5)$-representable matroids.)

EXAMPLES:

```
sage: M = matroids.catalog.Sp8pp(); M
Sp8=: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Sp8pp(); M
Sp8=: Quaternary matroid of rank 4 on 8 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**Spike**(*r*, *t=True*, *C3=[]*, *groundset=None*)

Return a rank-r spike.

Defined for all $r \geq 3$; a rank-r spike with tip t and legs $L_1, L_2, \ldots, L_r$, where $L_i = \{t, x_i, y_i\}$. Deleting t gives a tipless rank-r spike.

The groundset is $E = \{t, x_1, x_2, \ldots, x_r, y_1, y_2, \ldots, y_r\}$ with $r(E) = r$.

The nonspanning circuits are $\{L_1, L_2, \ldots, L_r\}$, all sets of the form $(L_i \cup L_j) \setminus t$ for $1 \leq i < j \leq r$, and some (possibly empty) collection C_3 of sets of the form $\{z_1, z_2, \ldots, z_r\}$ where $z_i \in \{x_i, y_i\}$ for all i, and no two members of C_3 have more than $r - 2$ common elements.

INPUT:

- `r` – integer ($r \geq 3$); the rank of the spike
- `t` – boolean (default: `True`); whether the spike is tipped
- `C3` – list (default: `[]`); a list of extra nonspanning circuits. The default (i.e. the empty list) results in a free r-spike.
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: matroid; a rank-r spike (tipped or tipless)

EXAMPLES:

```
sage: M = matroids.Spike(3, False); M
Free 3-spike\t: M \ {'t'}, where M is Matroid of rank 3 on 7 elements with 3
 nonspanning circuits
sage: M.is_isomorphic(matroids.Uniform(3, 6))
True
sage: len(matroids.Spike(8).bases())
4864
sage: import random
sage: r = random.choice(range(3, 20))
sage: M = matroids.Spike(r)
sage: M.is_3connected()
True
```

```
>>> from sage.all import *
>>> M = matroids.Spike(Integer(3), False); M
Free 3-spike\t: M \ {'t'}, where M is Matroid of rank 3 on 7 elements with 3
 nonspanning circuits
>>> M.is_isomorphic(matroids.Uniform(Integer(3), Integer(6)))
True
>>> len(matroids.Spike(Integer(8)).bases())
4864
>>> import random
>>> r = random.choice(range(Integer(3), Integer(20)))
>>> M = matroids.Spike(r)
>>> M.is_3connected()
True
```

Each of F_7, F_7^-, and P_7, is a 3-spike. After inspection of the nonspanning circuits of these matroids, it becomes clear that they indeed constitute tipped 3-spikes. This can be verified by using an appropriate choice of extra circuits in C_3:

```
sage: M = matroids.Spike(3, C3=[['x1', 'x2', 'y3'],
....:                           ['x1', 'x3', 'y2'],
....:                           ['x2', 'x3', 'y1'],
....:                           ['y1', 'y2', 'y3']])
sage: M.is_isomorphic(matroids.catalog.Fano())
True
sage: M = matroids.Spike(3, C3=[['x1', 'x2', 'x3'],
....:                           ['x1', 'y2', 'y3'],
....:                           ['x2', 'y1', 'y3']])
sage: M.is_isomorphic(matroids.catalog.NonFano())
True
sage: M = matroids.Spike(3, C3=[['x1', 'x2', 'y3'],
....:                           ['x3', 'y1', 'y2']])
sage: M.is_isomorphic(matroids.catalog.P7())
True
```

```
>>> from sage.all import *
>>> M = matroids.Spike(Integer(3), C3=[['x1', 'x2', 'y3'],
...                           ['x1', 'x3', 'y2'],
...                           ['x2', 'x3', 'y1'],
...                           ['y1', 'y2', 'y3']])
```

(continues on next page)

(continued from previous page)

```
>>> M.is_isomorphic(matroids.catalog.Fano())
True
>>> M = matroids.Spike(Integer(3), C3=[['x1', 'x2', 'x3'],
...                                    ['x1', 'y2', 'y3'],
...                                    ['x2', 'y1', 'y3']])
>>> M.is_isomorphic(matroids.catalog.NonFano())
True
>>> M = matroids.Spike(Integer(3), C3=[['x1', 'x2', 'y3'],
...                                    ['x3', 'y1', 'y2']])
>>> M.is_isomorphic(matroids.catalog.P7())
True
```

Deleting any element gives a self-dual matroid. The tipless free spike (i.e., when C_3 is empty) is identically self-dual:

```
sage: M = matroids.Spike(6)
sage: e = random.choice(list(M.groundset()))
sage: Minor = M.delete(e)
sage: Minor.is_isomorphic(Minor.dual())
True
sage: r = random.choice(range(3, 8))
sage: M = matroids.Spike(r, False)
sage: M.equals(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.Spike(Integer(6))
>>> e = random.choice(list(M.groundset()))
>>> Minor = M.delete(e)
>>> Minor.is_isomorphic(Minor.dual())
True
>>> r = random.choice(range(Integer(3), Integer(8)))
>>> M = matroids.Spike(r, False)
>>> M.equals(M.dual())
True
```

REFERENCES:

[Oxl2011], p. 662.

sage.matroids.database_matroids.**T12**(*groundset='abcdefghijkl'*)

Return the matroid T_{12}.

The edges of the Petersen graph can be labeled by the 4-circuits of T_{12} so that two edges are adjacent if and only if the corresponding 4-circuits overlap in exactly two elements. Relaxing a circuit-hyperplane yields an excluded minor for the class of matroids that are either binary or ternary.

EXAMPLES:

```
sage: M = matroids.catalog.T12(); M
T12: Binary matroid of rank 6 on 12 elements, type (2, None)
sage: M.is_valid()
True
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
```

(continues on next page)

(continued from previous page)

```
True
sage: M.automorphism_group().is_transitive()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.T12(); M
T12: Binary matroid of rank 6 on 12 elements, type (2, None)
>>> M.is_valid()
True
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
True
```

REFERENCES:

[Oxl2011], p. 658-9.

sage.matroids.database_matroids.**T8**(*groundset='abcdefgh'*)

Return the matroid T_8, represented over $GF(3)$.

The matroid T_8 is an 8-element matroid of rank-4. It is representable over a field if and only if that field has characteristic three. It is an excluded minor for the dyadic matroids.

EXAMPLES:

```
sage: M = matroids.catalog.T8(); M
T8: Ternary matroid of rank 4 on 8 elements, type 0-
sage: M.truncation().is_isomorphic(matroids.Uniform(3, 8))
True
sage: M.contract('e').is_isomorphic(matroids.catalog.P7())
True
sage: M.has_minor(matroids.Uniform(3, 8))
False
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.T8(); M
T8: Ternary matroid of rank 4 on 8 elements, type 0-
>>> M.truncation().is_isomorphic(matroids.Uniform(Integer(3), Integer(8)))
True
>>> M.contract('e').is_isomorphic(matroids.catalog.P7())
True
>>> M.has_minor(matroids.Uniform(Integer(3), Integer(8)))
False
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 649.

sage.matroids.database_matroids.**TK10**(*groundset=None*)

Return the matroid $TK10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.TK10(); M
TK10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.TK10(); M
TK10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**TQ10**(*groundset=None*)

Return the matroid $TQ10$.

An excluded minor for K_2-representable matroids. UPF is G. Self-dual. Has *TQ8* as a minor (delete 'd' and contract 'c').

EXAMPLES:

```
sage: M = matroids.catalog.TQ10(); M
TQ10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
sage: N = M.delete('d').contract('c')
sage: N.is_isomorphic(matroids.catalog.TQ8())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.TQ10(); M
TQ10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
>>> N = M.delete('d').contract('c')
>>> N.is_isomorphic(matroids.catalog.TQ8())
True
```

sage.matroids.database_matroids.**TQ8**(*groundset=None*)

Return the matroid $TQ8$.

An excluded minor for 2-regular matroids. UPF is K_2. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.TQ8(); M
TQ8: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.TQ8(); M
TQ8: Quaternary matroid of rank 4 on 8 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**TQ9**(*groundset=None*)

Return the matroid $TQ9$.

An excluded minor for K_2-representable matroids, and a single-element extension of *TQ8*. The UPF is G. Uniquely $GF(5)$-representable. (An excluded minor for H_2-representable matroids.)

EXAMPLES:

```
sage: TQ8 = matroids.catalog.TQ8()
sage: TQ9 = matroids.catalog.TQ9(); TQ9
TQ9: Quaternary matroid of rank 4 on 9 elements
sage: for M in TQ8.extensions():
....:     if M.is_isomorphic(TQ9):
....:         print(True)
....:         break
True
```

```
>>> from sage.all import *
>>> TQ8 = matroids.catalog.TQ8()
>>> TQ9 = matroids.catalog.TQ9(); TQ9
TQ9: Quaternary matroid of rank 4 on 9 elements
>>> for M in TQ8.extensions():
...     if M.is_isomorphic(TQ9):
...         print(True)
...         break
True
```

sage.matroids.database_matroids.**TQ9p**(*groundset=None*)

Return the matroid $TQ9^-$.

An excluded minor for G- and K_2-representable matroids, and a single-element extension of *TQ8*. UPF is $U_1^{(2)}$. (An excluded minor for H_2- and $GF(5)$-representable matroids.)

EXAMPLES:

```
sage: TQ8 = matroids.catalog.TQ8()
sage: TQ9p = matroids.catalog.TQ9p(); TQ9p
TQ9': Quaternary matroid of rank 4 on 9 elements
sage: for M in TQ8.extensions():
....:     if M.is_isomorphic(TQ9p):
....:         print(True)
....:         break
True
```

```
>>> from sage.all import *
>>> TQ8 = matroids.catalog.TQ8()
>>> TQ9p = matroids.catalog.TQ9p(); TQ9p
TQ9': Quaternary matroid of rank 4 on 9 elements
```

(continues on next page)

(continued from previous page)

```
>>> for M in TQ8.extensions():
...     if M.is_isomorphic(TQ9p):
...         print(True)
...         break
True
```

sage.matroids.database_matroids.**TU10**(*groundset=None*)

Return the matroid $TU10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.TU10(); M
TU10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.TU10(); M
TU10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**TernaryDowling3**(*groundset='abcdefghi'*)

Return the matroid $Q_3(GF(3)^\times)$, represented over $GF(3)$.

The matroid $Q_3(GF(3)^\times)$ is a 9-element matroid of rank-3. It is the rank-3 ternary Dowling geometry. It is representable over a field if and only if that field does not have characteristic two.

EXAMPLES:

```
sage: M = matroids.catalog.TernaryDowling3(); M
Q3(GF(3)x): Ternary matroid of rank 3 on 9 elements, type 0-
sage: len(list(M.linear_subclasses()))
72
sage: M.fundamental_cycle('abc', 'd')
{'a': 2, 'b': 1, 'd': 1}
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.TernaryDowling3(); M
Q3(GF(3)x): Ternary matroid of rank 3 on 9 elements, type 0-
>>> len(list(M.linear_subclasses()))
72
>>> M.fundamental_cycle('abc', 'd')
{'a': 2, 'b': 1, 'd': 1}
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 654.

sage.matroids.database_matroids.**Terrahawk**(*groundset='abcdefghijklmnop'*)

Return the Terrahawk matroid.

The Terrahawk is a binary matroid that is a sporadic exception in a chain theorem for internally 4-connected binary matroids.

EXAMPLES:

```
sage: M = matroids.catalog.Terrahawk(); M
Terrahawk: Binary matroid of rank 8 on 16 elements, type (0, 4)
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Terrahawk(); M
Terrahawk: Binary matroid of rank 8 on 16 elements, type (0, 4)
>>> M.is_valid()
True
```

REFERENCES:

[CMO2011]

sage.matroids.database_matroids.**Theta**(*n*, *groundset=None*)

Return the matroid Θ_n.

Defined for all $n \geq 2$. $\Theta_2 \cong U_{1,2} \bigoplus U_{1,2}$ and $\Theta_3 \cong M(K_4)$.

INPUT:

- `n` – integer ($n \geq 2$); the rank of the matroid
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: matroid (Θ_n)

EXAMPLES:

```
sage: matroids.Theta(30)
Theta_30: Matroid of rank 30 on 60 elements with 16270 circuits
sage: M = matroids.Theta(2)
sage: U12 = matroids.Uniform(1, 2)
sage: U = U12.direct_sum(U12)
sage: M.is_isomorphic(U)
True
sage: M = matroids.Theta(3)
sage: M.is_isomorphic(matroids.catalog.K4())
True
```

```
>>> from sage.all import *
>>> matroids.Theta(Integer(30))
Theta_30: Matroid of rank 30 on 60 elements with 16270 circuits
>>> M = matroids.Theta(Integer(2))
>>> U12 = matroids.Uniform(Integer(1), Integer(2))
>>> U = U12.direct_sum(U12)
>>> M.is_isomorphic(U)
True
```

(continues on next page)

(continued from previous page)

```
>>> M = matroids.Theta(Integer(3))
>>> M.is_isomorphic(matroids.catalog.K4())
True
```

Θ_n is self-dual; identically self-dual if and only if $n = 2$:

```
sage: M = matroids.Theta(2)
sage: M.equals(M.dual())
True
sage: import random
sage: n = random.choice(range(3, 7))
sage: M = matroids.Theta(n)
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.Theta(Integer(2))
>>> M.equals(M.dual())
True
>>> import random
>>> n = random.choice(range(Integer(3), Integer(7)))
>>> M = matroids.Theta(n)
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
```

For $n \leq 3$, its automorphism group is transitive, while for $n \geq 4$ it is not:

```
sage: n = random.choice(range(4, 8))
sage: M = matroids.Theta(2 + n % 2)
sage: M.automorphism_group().is_transitive()
True
sage: M = matroids.Theta(n)
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> n = random.choice(range(Integer(4), Integer(8)))
>>> M = matroids.Theta(Integer(2) + n % Integer(2))
>>> M.automorphism_group().is_transitive()
True
>>> M = matroids.Theta(n)
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 663-4.

sage.matroids.database_matroids.**TicTacToe**(*groundset=None*)

Return the TicTacToe matroid.

The dual of the TicTacToe matroid is not algebraic; it is unknown whether the TicTacToe matroid itself is algebraic.

EXAMPLES:

```
sage: M = matroids.catalog.TicTacToe(); M
TicTacToe: Matroid of rank 5 on 9 elements with 8 nonspanning circuits
sage: M.is_valid() and M.is_paving()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.TicTacToe(); M
TicTacToe: Matroid of rank 5 on 9 elements with 8 nonspanning circuits
>>> M.is_valid() and M.is_paving()
True
```

REFERENCES:

[Hoc]

sage.matroids.database_matroids.**TippedFree3spike**(*groundset=None*)

Return the tipped free 3-spike.

Unique 3-connected extension of *U36*. Stabilizer for K_2.

EXAMPLES:

```
sage: M = matroids.catalog.TippedFree3spike(); M
Tipped rank-3 free spike: Quaternary matroid of rank 3 on 7 elements
sage: M.has_minor(matroids.Uniform(3,6))
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.TippedFree3spike(); M
Tipped rank-3 free spike: Quaternary matroid of rank 3 on 7 elements
>>> M.has_minor(matroids.Uniform(Integer(3),Integer(6)))
True
```

sage.matroids.database_matroids.**U24**(*groundset='abcd'*)

The uniform matroid of rank 2 on 4 elements.

The 4-point line; isomorphic to $\mathcal{W}^2$, the rank-2 whirl. The unique excluded minor for the class of binary matroids.

EXAMPLES:

```
sage: M = matroids.catalog.U24(); M
U(2, 4): Matroid of rank 2 on 4 elements with circuit-closures
{2: {{'a', 'b', 'c', 'd'}}}
sage: N = matroids.Uniform(2, 4)
sage: M.is_isomorphic(N)
True
sage: M.automorphism_group().structure_description()
'S4'
```

```
>>> from sage.all import *
>>> M = matroids.catalog.U24(); M
U(2, 4): Matroid of rank 2 on 4 elements with circuit-closures
{2: {{'a', 'b', 'c', 'd'}}}
>>> N = matroids.Uniform(Integer(2), Integer(4))
>>> M.is_isomorphic(N)
```

(continues on next page)

(continued from previous page)

```
True
>>> M.automorphism_group().structure_description()
'S4'
```

$U_{2,4}$ is isomorphic to $\mathcal{W}^2$:

```
sage: W2 = matroids.Whirl(2)
sage: W2.is_isomorphic(M)
True
```

```
>>> from sage.all import *
>>> W2 = matroids.Whirl(Integer(2))
>>> W2.is_isomorphic(M)
True
```

identically self-dual:

```
sage: M.equals(M.dual())
True
```

```
>>> from sage.all import *
>>> M.equals(M.dual())
True
```

and 3-connected:

```
sage: M.is_3connected()
True
```

```
>>> from sage.all import *
>>> M.is_3connected()
True
```

REFERENCES:

[Oxl2011], p. 639.

sage.matroids.database_matroids.**U25**(*groundset='abcde'*)

The uniform matroid of rank 2 on 5 elements.

$U_{2,5}$ is the 5-point line. Dual to $U_{3,5}$.

EXAMPLES:

```
sage: U25 = matroids.catalog.U25(); U25
U(2, 5): Matroid of rank 2 on 5 elements with circuit-closures
{2: {{'a', 'b', 'c', 'd', 'e'}}}
sage: U25.is_graphic() or U25.is_regular()
False
sage: U35 = matroids.catalog.U35()
sage: U25.is_isomorphic(U35.dual())
True
```

```
>>> from sage.all import *
>>> U25 = matroids.catalog.U25(); U25
U(2, 5): Matroid of rank 2 on 5 elements with circuit-closures
{2: {{'a', 'b', 'c', 'd', 'e'}}}
>>> U25.is_graphic() or U25.is_regular()
False
>>> U35 = matroids.catalog.U35()
>>> U25.is_isomorphic(U35.dual())
True
```

REFERENCES:

[Oxl2011], p. 640.

sage.matroids.database_matroids.**U35**(*groundset='abcde'*)

The uniform matroid of rank 3 on 5 elements.

$U_{3,5}$ is five points freely placed in the plane. Dual to $U_{2,5}$.

EXAMPLES:

```
sage: U35 = matroids.catalog.U35(); U35
U(3, 5): Matroid of rank 3 on 5 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd', 'e'}}}
sage: U35.is_graphic() or U35.is_regular()
False
sage: U25 = matroids.catalog.U25()
sage: U35.is_isomorphic(U25.dual())
True
```

```
>>> from sage.all import *
>>> U35 = matroids.catalog.U35(); U35
U(3, 5): Matroid of rank 3 on 5 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd', 'e'}}}
>>> U35.is_graphic() or U35.is_regular()
False
>>> U25 = matroids.catalog.U25()
>>> U35.is_isomorphic(U25.dual())
True
```

REFERENCES:

[Oxl2011], p. 640.

sage.matroids.database_matroids.**U36**(*groundset='abcdef'*)

The uniform matroid of rank 3 on 6 elements.

Six points freely placed in the plane; the tipless free 3-spike. Identically self-dual.

EXAMPLES:

```
sage: U36 = matroids.catalog.U36(); U36
U(3, 6): Matroid of rank 3 on 6 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd', 'e', 'f'}}}
sage: Z = matroids.Spike(3, False)
sage: U36.is_isomorphic(Z)
```

(continues on next page)

(continued from previous page)

```
True
sage: U36.equals(U36.dual())
True
sage: U36.automorphism_group().structure_description()
'S6'
```

```
>>> from sage.all import *
>>> U36 = matroids.catalog.U36(); U36
U(3, 6): Matroid of rank 3 on 6 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd', 'e', 'f'}}}
>>> Z = matroids.Spike(Integer(3), False)
>>> U36.is_isomorphic(Z)
True
>>> U36.equals(U36.dual())
True
>>> U36.automorphism_group().structure_description()
'S6'
```

REFERENCES:

[Oxl2011], p. 642.

sage.matroids.database_matroids.**UA12**(*groundset=None*)

Return the matroid $UA12$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Not self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.UA12(); M
UA12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.UA12(); M
UA12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**UG10**(*groundset=None*)

Return the matroid $UG10$.

An excluded minor for K_2- and P_4-representable matroids. Self-dual. An excluded minor for H_3- and H_2-representable matroids. Uniquely $GF(5)$-representable. Although not P_4-representable, it is O-representable, and hence is representable over all fields of size at least four.

EXAMPLES:

```
sage: M = matroids.catalog.UG10(); M
UG10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.UG10(); M
UG10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**UK10**(*groundset=None*)

Return the matroid $UK10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Not self-dual. UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.UK10(); M
UK10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.UK10(); M
UK10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**UK12**(*groundset=None*)

Return the matroid $UK12$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is I.

EXAMPLES:

```
sage: M = matroids.catalog.UK12(); M
UK12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.UK12(); M
UK12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**UP14**(*groundset=None*)

Return the matroid $UP14$.

An excluded minor for K_2-representable matroids. Has disjoint circuit-hyperplanes. UPF is W. Not self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.UP14(); M
UP14: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.UP14(); M
UP14: Quaternary matroid of rank 7 on 14 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**UQ10**(*groundset=None*)

Return the matroid $UQ10$.

An excluded minor for K_2- and G-representable matroids (and H_2- and $GF(5)$-representable matroids). Self-dual. UPF is $(Q(a, b), < -1, a, b, a - 1, b - 1, a - b, a + b, a + 1, ab + b - 1, ab - b + 1 >)$.

EXAMPLES:

```
sage: M = matroids.catalog.UQ10(); M
UQ10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.UQ10(); M
UQ10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**UQ12**(*groundset=None*)

Return the matroid $UQ12$.

An excluded minor for K_2 and G-representable matroids (and $H2$ and $GF(5)$-representable matroids). UPF is P_{pappus}. Self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.UQ12(); M
UQ12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.UQ12(); M
UQ12: Quaternary matroid of rank 6 on 12 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**UT10**(*groundset=None*)

Return the matroid $UT10$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is I.

EXAMPLES:

```
sage: M = matroids.catalog.UT10(); M
UT10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.UT10(); M
UT10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**Uniform**(*r*, *n*, *groundset=None*)

Return the uniform matroid of rank r on n elements.

All subsets of size r or less are independent; all larger subsets are dependent. Representable when the field is sufficiently large. The precise bound is the subject of the MDS conjecture from coding theory.

INPUT:

- `r` – nonnegative integer; the rank of the uniform matroid
- `n` – nonnegative integer; the number of elements of the uniform matroid
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: the uniform matroid $U_{r,n}$

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.Uniform(2, 5); M
U(2, 5): Matroid of rank 2 on 5 elements with circuit-closures
{2: {{0, 1, 2, 3, 4}}}
sage: M.dual().is_isomorphic(matroids.Uniform(3, 5))
True
sage: setprint(M.hyperplanes())
[{0}, {1}, {2}, {3}, {4}]
sage: M.has_line_minor(6)
False
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.Uniform(Integer(2), Integer(5)); M
U(2, 5): Matroid of rank 2 on 5 elements with circuit-closures
{2: {{0, 1, 2, 3, 4}}}
>>> M.dual().is_isomorphic(matroids.Uniform(Integer(3), Integer(5)))
True
>>> setprint(M.hyperplanes())
[{0}, {1}, {2}, {3}, {4}]
>>> M.has_line_minor(Integer(6))
False
>>> M.is_valid()
True
```

Check that bug Issue #15292 was fixed:

```
sage: M = matroids.Uniform(4,4)
sage: len(M.circuit_closures())
0
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(4),Integer(4))
>>> len(M.circuit_closures())
0
```

REFERENCES:

[Oxl2011], p. 660.

sage.matroids.database_matroids.**VP14**(*groundset=None*)

Return the matroid $VP14$.

An excluded minor for K_2-representable matroids. Has disjoint circuit-hyperplanes. UPF is W. Not self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.VP14(); M
VP14: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.VP14(); M
VP14: Quaternary matroid of rank 7 on 14 elements
>>> M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.**Vamos**(*groundset=None*)

Return the $Vámos$ matroid, represented as circuit closures.

The $Vámos$ matroid, or $Vámos$ cube, or V_8 is an 8-element matroid of rank-4. It violates Ingleton's condition for representability over a division ring. It is not algebraic.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Vamos(); M
Vamos: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
     {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: setprint(M.nonbases())
[{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
 {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}]
sage: M.is_dependent(['c', 'd', 'g', 'h'])
False
sage: M.is_valid() and M.is_paving()  # long time
True
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
```

(continues on next page)

(continued from previous page)

```
>>> M = matroids.catalog.Vamos(); M
Vamos: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
     {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
>>> setprint(M.nonbases())
[{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
 {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}]
>>> M.is_dependent(['c', 'd', 'g', 'h'])
False
>>> M.is_valid() and M.is_paving()  # long time
True
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 649.

sage.matroids.database_matroids.**WQ8**(*groundset=None*)

Return the matroid $WQ8$.

An excluded minor for G, K_2, H_4, and $GF(5)$-representable matroids. Self-dual. UPF is $(Z[\zeta, a], <\zeta, a-\zeta>)$ where ζ is solution to $x^2 - x + 1 = 0$ and a is an indeterminate.

EXAMPLES:

```
sage: M = matroids.catalog.WQ8(); M
WQ8: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.WQ8(); M
WQ8: Quaternary matroid of rank 4 on 8 elements
>>> M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.**Wheel**(*r*, *field=None*, *ring=None*, *groundset=None*)

Return the rank-r wheel.

INPUT:

- `r` – positive integer; the rank of the matroid
- `ring` – any ring; if provided, output will be a linear matroid over the ring or field `ring`. If the ring is **Z**, then output will be a regular matroid.
- `field` – any field; same as `ring`, but only fields are allowed
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: the rank-r wheel matroid, represented as a regular matroid

EXAMPLES:

```
sage: M = matroids.Wheel(5); M
Wheel(5): Regular matroid of rank 5 on 10 elements with 121 bases
sage: M.tutte_polynomial()
x^5 + y^5 + 5*x^4 + 5*x^3*y + 5*x^2*y^2 + 5*x*y^3 + 5*y^4 + 10*x^3 +
15*x^2*y + 15*x*y^2 + 10*y^3 + 10*x^2 + 16*x*y + 10*y^2 + 4*x + 4*y
sage: M.is_valid()
True
sage: M = matroids.Wheel(3)
sage: M.is_isomorphic(matroids.CompleteGraphic(4))
True
sage: M.is_isomorphic(matroids.Wheel(3, field=GF(3)))
True
sage: M = matroids.Wheel(3, field=GF(3)); M
Wheel(3): Ternary matroid of rank 3 on 6 elements, type 0+
```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(5)); M
Wheel(5): Regular matroid of rank 5 on 10 elements with 121 bases
>>> M.tutte_polynomial()
x^5 + y^5 + 5*x^4 + 5*x^3*y + 5*x^2*y^2 + 5*x*y^3 + 5*y^4 + 10*x^3 +
15*x^2*y + 15*x*y^2 + 10*y^3 + 10*x^2 + 16*x*y + 10*y^2 + 4*x + 4*y
>>> M.is_valid()
True
>>> M = matroids.Wheel(Integer(3))
>>> M.is_isomorphic(matroids.CompleteGraphic(Integer(4)))
True
>>> M.is_isomorphic(matroids.Wheel(Integer(3), field=GF(Integer(3))))
True
>>> M = matroids.Wheel(Integer(3), field=GF(Integer(3))); M
Wheel(3): Ternary matroid of rank 3 on 6 elements, type 0+
```

For $r \geq 2$, the wheel is self-dual but not identically self-dual, and for $r \geq 4$ it has a non-transitive automorphism group:

```
sage: import random
sage: r = random.choice(range(4, 8))
sage: M = matroids.Wheel(r)
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> import random
>>> r = random.choice(range(Integer(4), Integer(8)))
>>> M = matroids.Wheel(r)
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 659-60.

sage.matroids.database_matroids.**Wheel4**(*groundset='abcdefgh'*)

Return the rank-4 wheel.

A regular, graphic, and cographic matroid. Self-dual but not identically self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.Wheel4(); M
Wheel(4): Regular matroid of rank 4 on 8 elements with 45 bases
sage: M.is_valid() and M.is_graphic() and M.dual().is_graphic()
True
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Wheel4(); M
Wheel(4): Regular matroid of rank 4 on 8 elements with 45 bases
>>> M.is_valid() and M.is_graphic() and M.dual().is_graphic()
True
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 651-2.

sage.matroids.database_matroids.**Whirl**(*r*, *groundset=None*)

Return the rank-r whirl.

The whirl is the unique relaxation of the wheel.

INPUT:

- `r` – positive integer; the rank of the matroid
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: the rank-r whirl matroid, represented as a ternary matroid

EXAMPLES:

```
sage: M = matroids.Whirl(5); M
Whirl(5): Ternary matroid of rank 5 on 10 elements, type 0-
sage: M.is_valid()
True
sage: M.tutte_polynomial()
x^5 + y^5 + 5*x^4 + 5*x^3*y + 5*x^2*y^2 + 5*x*y^3 + 5*y^4 + 10*x^3 + 15*x^2*y +
 15*x*y^2 + 10*y^3 + 10*x^2 + 15*x*y + 10*y^2 + 5*x + 5*y
sage: M.is_isomorphic(matroids.Wheel(5))
False
sage: M = matroids.Whirl(3)
sage: M.is_isomorphic(matroids.CompleteGraphic(4))
False
```

```
>>> from sage.all import *
>>> M = matroids.Whirl(Integer(5)); M
Whirl(5): Ternary matroid of rank 5 on 10 elements, type 0-
>>> M.is_valid()
True
>>> M.tutte_polynomial()
x^5 + y^5 + 5*x^4 + 5*x^3*y + 5*x^2*y^2 + 5*x*y^3 + 5*y^4 + 10*x^3 + 15*x^2*y +
 15*x*y^2 + 10*y^3 + 10*x^2 + 15*x*y + 10*y^2 + 5*x + 5*y
>>> M.is_isomorphic(matroids.Wheel(Integer(5)))
False
>>> M = matroids.Whirl(Integer(3))
>>> M.is_isomorphic(matroids.CompleteGraphic(Integer(4)))
False
```

For $r \geq 3$, the whirl is self-dual but not identically self-dual:

```
sage: import random
sage: r = random.choice(range(3, 10))
sage: M = matroids.Whirl(r)
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
```

```
>>> from sage.all import *
>>> import random
>>> r = random.choice(range(Integer(3), Integer(10)))
>>> M = matroids.Whirl(r)
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
```

Except for $\mathcal{W}^2$, which is isomorphic to $U_{2,4}$, these matroids have non-transitive automorphism groups:

```
sage: r = random.choice(range(3, 8))
sage: M = matroids.Whirl(r)
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> r = random.choice(range(Integer(3), Integer(8)))
>>> M = matroids.Whirl(r)
>>> M.automorphism_group().is_transitive()
False
```

Todo

Optional arguments `ring` and `x`, such that the resulting matroid is represented over `ring` by a reduced matrix like `[-1 0 x] [ 1 -1 0] [ 0 1 -1]`

REFERENCES:

[Oxl2011], p. 659-60.

`sage.matroids.database_matroids.`**`Whirl3`**(*groundset='abcdef'*)

The rank-3 whirl.

The unique relaxation of $M(K_4)$. Self-dual but not identically self-dual.

EXAMPLES:

```
sage: W = matroids.catalog.Whirl3(); W
Whirl(3): Ternary matroid of rank 3 on 6 elements, type 0-
sage: W.equals(W.dual())
False
sage: W.is_isomorphic(W.dual())
True
sage: W.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> W = matroids.catalog.Whirl3(); W
Whirl(3): Ternary matroid of rank 3 on 6 elements, type 0-
>>> W.equals(W.dual())
False
>>> W.is_isomorphic(W.dual())
True
>>> W.automorphism_group().is_transitive()
False
```

For all elements e, neither $\mathcal{W}_3 \setminus \{e\}$ nor $\mathcal{W}_3/\{e\}$ is 3-connected:

```
sage: import random
sage: e = random.choice(list(W.groundset()))
sage: W.delete(e).is_3connected()
False
sage: W.contract(e).is_3connected()
False
```

```
>>> from sage.all import *
>>> import random
>>> e = random.choice(list(W.groundset()))
>>> W.delete(e).is_3connected()
False
>>> W.contract(e).is_3connected()
False
```

REFERENCES:

[Oxl2011], p. 641.

sage.matroids.database_matroids.**Whirl4**(*groundset='abcdefgh'*)

Return the rank-4 whirl.

A matroid which is not graphic, not cographic, and not regular. Self-dual but not identically self-dual.

EXAMPLES:

```
sage: M = matroids.catalog.Whirl4(); M
Whirl(4): Ternary matroid of rank 4 on 8 elements, type 0+
sage: M.is_valid()
True
```

(continues on next page)

(continued from previous page)

```
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
sage: M.automorphism_group().is_transitive()
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Whirl4(); M
Whirl(4): Ternary matroid of rank 4 on 8 elements, type 0+
>>> M.is_valid()
True
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 652.

sage.matroids.database_matroids.**XY13**(*groundset=None*)

Return the matroid $XY13$.

An excluded minor for G-representable matroids (and $GF(5)$-representable matroids). UPF is $GF(4)$.

EXAMPLES:

```
sage: M = matroids.catalog.XY13(); M
XY13: Quaternary matroid of rank 6 on 13 elements
sage: M.is_3connected()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.XY13(); M
XY13: Quaternary matroid of rank 6 on 13 elements
>>> M.is_3connected()
True
```

sage.matroids.database_matroids.**Z**(*r*, *t=True*, *groundset=None*)

Return the unique rank-r binary spike.

Defined for all $r \geq 3$.

INPUT:

- `r` – integer ($r \geq 3$); the rank of the spike
- `t` – boolean (default: `True`); whether the spike is tipped
- `groundset` – string (optional); the groundset of the matroid

OUTPUT: matroid; the unique rank-r binary spike (tipped or tipless)

EXAMPLES:

```
sage: matroids.Z(8)
Z_8: Binary matroid of rank 8 on 17 elements, type (7, 1)
sage: matroids.Z(9)
```

(continues on next page)

(continued from previous page)

```
Z_9: Binary matroid of rank 9 on 19 elements, type (9, None)
sage: matroids.Z(20, False)
Z_20\t: Binary matroid of rank 20 on 40 elements, type (20, 0)
```

```
>>> from sage.all import *
>>> matroids.Z(Integer(8))
Z_8: Binary matroid of rank 8 on 17 elements, type (7, 1)
>>> matroids.Z(Integer(9))
Z_9: Binary matroid of rank 9 on 19 elements, type (9, None)
>>> matroids.Z(Integer(20), False)
Z_20\t: Binary matroid of rank 20 on 40 elements, type (20, 0)
```

It holds that $Z_3 \setminus e \cong M(K4)$, for all e:

```
sage: import random
sage: Z3 = matroids.Z(3)
sage: E = sorted(Z3.groundset()); E
['t', 'x1', 'x2', 'x3', 'y1', 'y2', 'y3']
sage: e = random.choice(E)
sage: Z3.delete(e).is_isomorphic(matroids.catalog.K4())
True
```

```
>>> from sage.all import *
>>> import random
>>> Z3 = matroids.Z(Integer(3))
>>> E = sorted(Z3.groundset()); E
['t', 'x1', 'x2', 'x3', 'y1', 'y2', 'y3']
>>> e = random.choice(E)
>>> Z3.delete(e).is_isomorphic(matroids.catalog.K4())
True
```

$Z_3 \cong F_7$:

```
sage: F7 = matroids.catalog.Fano()
sage: Z3.is_isomorphic(F7)
True
```

```
>>> from sage.all import *
>>> F7 = matroids.catalog.Fano()
>>> Z3.is_isomorphic(F7)
True
```

$Z_4 \setminus t \cong AG(3,2)$:

```
sage: Z4 = matroids.Z(4, False)
sage: Z4.is_isomorphic(matroids.catalog.AG32())
True
```

```
>>> from sage.all import *
>>> Z4 = matroids.Z(Integer(4), False)
>>> Z4.is_isomorphic(matroids.catalog.AG32())
True
```

and $Z_4 \setminus e \cong S_8$, for all $e \neq t$:

```
sage: Z4 = matroids.Z(4)
sage: E = sorted(Z4.groundset())
sage: E.remove('t')
sage: e = random.choice(E)
sage: S8 = matroids.catalog.S8()
sage: Z4.delete(e).is_isomorphic(S8)
True
sage: Z4.delete('t').is_isomorphic(S8)
False
```

```
>>> from sage.all import *
>>> Z4 = matroids.Z(Integer(4))
>>> E = sorted(Z4.groundset())
>>> E.remove('t')
>>> e = random.choice(E)
>>> S8 = matroids.catalog.S8()
>>> Z4.delete(e).is_isomorphic(S8)
True
>>> Z4.delete('t').is_isomorphic(S8)
False
```

The tipless binary spike is self-dual; it is identically self-dual if and only if r is even. It also has a transitive automorphism group:

```
sage: r = random.choice(range(3, 8))
sage: Z = matroids.Z(r, False)
sage: Z.is_isomorphic(Z.dual())
True
sage: Z.equals(Z.dual()) != (r % 2 == 1)  # XOR
True
sage: Z.automorphism_group().is_transitive()  # long time
True
```

```
>>> from sage.all import *
>>> r = random.choice(range(Integer(3), Integer(8)))
>>> Z = matroids.Z(r, False)
>>> Z.is_isomorphic(Z.dual())
True
>>> Z.equals(Z.dual()) != (r % Integer(2) == Integer(1))  # XOR
True
>>> Z.automorphism_group().is_transitive()  # long time
True
```

REFERENCES:

[Oxl2011], p. 661-2.

CHAPTER

THREE

CONCRETE IMPLEMENTATIONS

3.1 Basis matroids

In a matroid, a basis is an inclusionwise maximal independent set. The common cardinality of all bases is the rank of the matroid. Matroids are uniquely determined by their set of bases.

This module defines the class *BasisMatroid*, which internally represents a matroid as a set of bases. It is a subclass of *BasisExchangeMatroid*, and as such it inherits all method from that class and from the class *Matroid*. Additionally, it provides the following methods:

- *is_distinguished()*
- *relabel()*

3.1.1 Construction

A `BasisMatroid` can be created from another matroid, from a list of bases, or from a list of nonbases. For a full description of allowed inputs, see *below*. It is recommended to use the *Matroid()* function for easy construction of a `BasisMatroid`. For direct access to the `BasisMatroid` constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See also *sage.matroids.advanced*.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M1 = BasisMatroid(groundset='abcd', bases=['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
sage: M2 = Matroid(['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
sage: M1 == M2
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M1 = BasisMatroid(groundset='abcd', bases=['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
>>> M2 = Matroid(['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
>>> M1 == M2
True
```

3.1.2 Implementation

The set of bases is compactly stored in a bitset which takes $O(binomial(N, R))$ bits of space, where N is the cardinality of the groundset and R is the rank. `BasisMatroid` inherits the matroid oracle from its parent class `BasisExchangeMatroid`, by providing the elementary functions for exploring the base exchange graph. In addition, `BasisMatroid` has methods for constructing minors, duals, single-element extensions, for testing matroid isomorphism and minor inclusion.

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

class sage.matroids.basis_matroid.**BasisMatroid**

Bases: *BasisExchangeMatroid*

Create general matroid, stored as a set of bases.

INPUT:

- `M` – matroid (optional)
- `groundset` – any iterable set (optional)
- `bases` – set of subsets of the `groundset` (optional)
- `nonbases` – set of subsets of the `groundset` (optional)
- `rank` – natural number (optional)

EXAMPLES:

The empty matroid:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid()
sage: M.groundset()
frozenset()
sage: M.full_rank()
0
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = BasisMatroid()
>>> M.groundset()
frozenset()
>>> M.full_rank()
0
```

Create a BasisMatroid instance out of any other matroid:

```
sage: from sage.matroids.advanced import *
sage: F = matroids.catalog.Fano()
sage: M = BasisMatroid(F)
sage: F.groundset() == M.groundset()
True
sage: len(set(F.bases()).difference(M.bases()))
0
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

(continues on next page)

(continued from previous page)

```
>>> F = matroids.catalog.Fano()
>>> M = BasisMatroid(F)
>>> F.groundset() == M.groundset()
True
>>> len(set(F.bases()).difference(M.bases()))
0
```

It is possible to provide either bases or nonbases:

```
sage: from sage.matroids.advanced import *
sage: M1 = BasisMatroid(groundset='abc', bases=['ab', 'ac'] )
sage: M2 = BasisMatroid(groundset='abc', nonbases=['bc'])
sage: M1 == M2
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M1 = BasisMatroid(groundset='abc', bases=['ab', 'ac'] )
>>> M2 = BasisMatroid(groundset='abc', nonbases=['bc'])
>>> M1 == M2
True
```

Providing only groundset and rank creates a uniform matroid:

```
sage: from sage.matroids.advanced import *
sage: M1 = BasisMatroid(matroids.Uniform(2, 5))
sage: M2 = BasisMatroid(groundset=range(5), rank=2)
sage: M1 == M2
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M1 = BasisMatroid(matroids.Uniform(Integer(2), Integer(5)))
>>> M2 = BasisMatroid(groundset=range(Integer(5)), rank=Integer(2))
>>> M1 == M2
True
```

We do not check if the provided input forms an actual matroid:

```
sage: from sage.matroids.advanced import *
sage: M1 = BasisMatroid(groundset='abcd', bases=['ab', 'cd'])
sage: M1.full_rank()
2
sage: M1.is_valid()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M1 = BasisMatroid(groundset='abcd', bases=['ab', 'cd'])
>>> M1.full_rank()
2
```

(continues on next page)

(continued from previous page)

```
>>> M1.is_valid()
False
```

bases()

Return the bases of the matroid.

A *basis* is a maximal independent set.

OUTPUT: iterable containing all bases of the matroid

EXAMPLES:

```
sage: M = Matroid(bases=matroids.catalog.Fano().bases())
sage: M
Matroid of rank 3 on 7 elements with 28 bases
sage: len(M.bases())
28
```

```
>>> from sage.all import *
>>> M = Matroid(bases=matroids.catalog.Fano().bases())
>>> M
Matroid of rank 3 on 7 elements with 28 bases
>>> len(M.bases())
28
```

bases_count()

Return the number of bases of the matroid.

OUTPUT: integer

EXAMPLES:

```
sage: M = Matroid(bases=matroids.catalog.Fano().bases())
sage: M
Matroid of rank 3 on 7 elements with 28 bases
sage: M.bases_count()
28
```

```
>>> from sage.all import *
>>> M = Matroid(bases=matroids.catalog.Fano().bases())
>>> M
Matroid of rank 3 on 7 elements with 28 bases
>>> M.bases_count()
28
```

dual()

Return the dual of the matroid.

Let M be a matroid with groundset E. If B is the set of bases of M, then the set $\{E - b : b \in B\}$ is the set of bases of another matroid, the *dual* of M.

EXAMPLES:

```
sage: M = Matroid(bases=matroids.catalog.Pappus().bases())
sage: M.dual()
Matroid of rank 6 on 9 elements with 75 bases
```

```
>>> from sage.all import *
>>> M = Matroid(bases=matroids.catalog.Pappus().bases())
>>> M.dual()
Matroid of rank 6 on 9 elements with 75 bases
```

ALGORITHM:

A BasisMatroid on n elements and of rank r is stored as a bitvector of length $\binom{n}{r}$. The i-th bit in this vector indicates that the i-th r-set in the lexicographic enumeration of r-subsets of the groundset is a basis. Reversing this bitvector yields a bitvector that indicates whether the complement of an $(n-r)$-set is a basis, i.e. gives the bitvector of the bases of the dual.

is_distinguished(*e*)

Return whether `e` is a 'distinguished' element of the groundset.

The set of distinguished elements is an isomorphism invariant. Each matroid has at least one distinguished element. The typical application of this method is the execution of an orderly algorithm for generating all matroids up to isomorphism in a minor-closed class, by successively enumerating the single-element extensions and coextensions of the matroids generated so far.

INPUT:

- `e` – element of the groundset

OUTPUT: boolean

See also

M.extensions(), *M.linear_subclasses()*, *sage.matroids.extension*

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid(matroids.catalog.N1())
sage: sorted([e for e in M.groundset() if M.is_distinguished(e)])
['c', 'g', 'h', 'j']
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = BasisMatroid(matroids.catalog.N1())
>>> sorted([e for e in M.groundset() if M.is_distinguished(e)])
['c', 'g', 'h', 'j']
```

nonbases()

Return the nonbases of the matroid.

A *nonbasis* is a set with cardinality `self.full_rank()` that is not a basis.

OUTPUT: iterable containing the nonbases of the matroid

See also

Matroid.basis()

EXAMPLES:

```
sage: M = Matroid(bases=matroids.catalog.Fano().bases())
sage: M
Matroid of rank 3 on 7 elements with 28 bases
sage: len(M.nonbases())
7
```

```
>>> from sage.all import *
>>> M = Matroid(bases=matroids.catalog.Fano().bases())
>>> M
Matroid of rank 3 on 7 elements with 28 bases
>>> len(M.nonbases())
7
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: from sage.matroids.advanced import BasisMatroid
sage: M = BasisMatroid(matroids.catalog.Fano())
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
sage: N = M.relabel({'a': 0, 'g': 'x'})
sage: from sage.matroids.utilities import cmp_elements_key
sage: sorted(N.groundset(), key=cmp_elements_key)
[0, 'b', 'c', 'd', 'e', 'f', 'x']
sage: N.is_isomorphic(M)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import BasisMatroid
>>> M = BasisMatroid(matroids.catalog.Fano())
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> N = M.relabel({'a': Integer(0), 'g': 'x'})
>>> from sage.matroids.utilities import cmp_elements_key
>>> sorted(N.groundset(), key=cmp_elements_key)
[0, 'b', 'c', 'd', 'e', 'f', 'x']
>>> N.is_isomorphic(M)
True
```

truncation()

Return a rank-1 truncation of the matroid.

Let M be a matroid of rank r. The *truncation* of M is the matroid obtained by declaring all subsets of size r dependent. It can be obtained by adding an element freely to the span of the matroid and then contracting that element.

OUTPUT: matroid

See also

M.extension(), *M.contract()*

EXAMPLES:

```
sage: M = Matroid(bases=matroids.catalog.N2().bases())
sage: M.truncation()
Matroid of rank 5 on 12 elements with 702 bases
sage: M.whitney_numbers2()
[1, 12, 66, 190, 258, 99, 1]
sage: M.truncation().whitney_numbers2()
[1, 12, 66, 190, 258, 1]
```

```
>>> from sage.all import *
>>> M = Matroid(bases=matroids.catalog.N2().bases())
>>> M.truncation()
Matroid of rank 5 on 12 elements with 702 bases
>>> M.whitney_numbers2()
[1, 12, 66, 190, 258, 99, 1]
>>> M.truncation().whitney_numbers2()
[1, 12, 66, 190, 258, 1]
```

3.2 Circuit closures matroids

Matroids are characterized by a list of all tuples (C, k), where C is the closure of a circuit, and k the rank of C. The CircuitClosuresMatroid class implements matroids using this information as data.

3.2.1 Construction

A `CircuitClosuresMatroid` can be created from another matroid or from a list of circuit-closures. For a full description of allowed inputs, see *below*. It is recommended to use the *Matroid()* function for a more flexible construction of a `CircuitClosuresMatroid`. For direct access to the `CircuitClosuresMatroid` constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See also *sage.matroids.advanced*.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M1 = CircuitClosuresMatroid(groundset='abcdef',
....:                 circuit_closures={2: ['abc', 'ade'], 3: ['abcdef']})
sage: M2 = Matroid(circuit_closures={2: ['abc', 'ade'], 3: ['abcdef']})
sage: M3 = Matroid(circuit_closures=[(2, 'abc'),
....:                                (3, 'abcdef'), (2, 'ade')])
sage: M1 == M2
True
sage: M1 == M3
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M1 = CircuitClosuresMatroid(groundset='abcdef',
...                 circuit_closures={Integer(2): ['abc', 'ade'], Integer(3): ['abcdef
↪']})
>>> M2 = Matroid(circuit_closures={Integer(2): ['abc', 'ade'], Integer(3): ['abcdef']}
↪)
>>> M3 = Matroid(circuit_closures=[(Integer(2), 'abc'),
...                                (Integer(3), 'abcdef'), (Integer(2), 'ade')])
>>> M1 == M2
True
>>> M1 == M3
True
```

Note that the class does not implement custom minor and dual operations:

```
sage: from sage.matroids.advanced import *
sage: M = CircuitClosuresMatroid(groundset='abcdef',
....:                 circuit_closures={2: ['abc', 'ade'], 3: ['abcdef']})
sage: isinstance(M.contract('a'), MinorMatroid)
True
sage: isinstance(M.dual(), DualMatroid)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = CircuitClosuresMatroid(groundset='abcdef',
...                 circuit_closures={Integer(2): ['abc', 'ade'], Integer(3): ['abcdef
↪']})
>>> isinstance(M.contract('a'), MinorMatroid)
True
>>> isinstance(M.dual(), DualMatroid)
True
```

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

class sage.matroids.circuit_closures_matroid.**CircuitClosuresMatroid**

Bases: *Matroid*

A general matroid M is characterized by its rank $r(M)$ and the set of pairs

$(k, \{ \text{closure } (C) : C`circuitof`M, r(C) = k\})$ for $k = 0, .., r(M) - 1$

As each independent set of size k is in at most one closure(C) of rank k, and each closure(C) of rank k contains at least $k+1$ independent sets of size k, there are at most $\binom{n}{k}/(k+1)$ such closures-of-circuits of rank k. Each closure(C) takes $O(n)$ bits to store, giving an upper bound of $O(2^n)$ on the space complexity of the entire matroid.

A subset X of the groundset is independent if and only if

$|X \cap \text{`closure`}(C)| \leq k$ for all circuits C of M with $r(C) = k$.

So determining whether a set is independent takes time proportional to the space complexity of the matroid.

INPUT:

- `M` – matroid (default: `None`)
- `groundset` – groundset of a matroid (default: `None`)
- `circuit_closures` – dictionary (default: `None`); the collection of circuit closures of a matroid presented as a dictionary whose keys are ranks, and whose values are sets of circuit closures of the specified rank

OUTPUT:

- If the input is a matroid `M`, return a `CircuitClosuresMatroid` instance representing `M`.
- Otherwise, return a `CircuitClosuresMatroid` instance based on `groundset` and `circuit_closures`.

Note

For a more flexible means of input, use the `Matroid()` function.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = CircuitClosuresMatroid(matroids.catalog.Fano())
sage: M
Matroid of rank 3 on 7 elements with circuit-closures
{2: {{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'},
     {'b', 'c', 'd'}, {'b', 'e', 'g'}, {'c', 'f', 'g'},
     {'d', 'e', 'f'}}, 3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g'}}}
sage: M = CircuitClosuresMatroid(groundset='abcdefgh',
....:             circuit_closures={3: ['edfg', 'acdg', 'bcfg', 'cefh',
....:                  'afgh', 'abce', 'abdf', 'begh', 'bcdh', 'adeh'],
....:                                4: ['abcdefgh']})
sage: M.equals(matroids.catalog.P8())
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = CircuitClosuresMatroid(matroids.catalog.Fano())
>>> M
Matroid of rank 3 on 7 elements with circuit-closures
{2: {{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'},
     {'b', 'c', 'd'}, {'b', 'e', 'g'}, {'c', 'f', 'g'},
     {'d', 'e', 'f'}}, 3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g'}}}
>>> M = CircuitClosuresMatroid(groundset='abcdefgh',
...             circuit_closures={Integer(3): ['edfg', 'acdg', 'bcfg', 'cefh',
...                  'afgh', 'abce', 'abdf', 'begh', 'bcdh', 'adeh'],
...                                Integer(4): ['abcdefgh']})
```

(continues on next page)

(continued from previous page)

```
>>> M.equals(matroids.catalog.P8())
True
```

circuit_closures()

Return the closures of circuits of the matroid.

A *circuit closure* is a closed set containing a circuit.

OUTPUT: dictionary containing the circuit closures of the matroid, indexed by their ranks

See also

Matroid.circuit(), *Matroid.closure()*

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = CircuitClosuresMatroid(matroids.catalog.Fano())
sage: CC = M.circuit_closures()
sage: len(CC[2])
7
sage: len(CC[3])
1
sage: len(CC[1])
Traceback (most recent call last):
...
KeyError: 1
sage: [sorted(X) for X in CC[3]]
[['a', 'b', 'c', 'd', 'e', 'f', 'g']]
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = CircuitClosuresMatroid(matroids.catalog.Fano())
>>> CC = M.circuit_closures()
>>> len(CC[Integer(2)])
7
>>> len(CC[Integer(3)])
1
>>> len(CC[Integer(1)])
Traceback (most recent call last):
...
KeyError: 1
>>> [sorted(X) for X in CC[Integer(3)]]
[['a', 'b', 'c', 'd', 'e', 'f', 'g']]
```

full_rank()

Return the rank of the matroid.

The *rank* of the matroid is the size of the largest independent subset of the groundset.

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M.full_rank()
4
sage: M.dual().full_rank()
4
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.full_rank()
4
>>> M.dual().full_rank()
4
```

groundset()

Return the groundset of the matroid.

The groundset is the set of elements that comprise the matroid.

OUTPUT: frozenset

EXAMPLES:

```
sage: M = matroids.catalog.Pappus()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus()
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: from sage.matroids.circuit_closures_matroid import␣
↪CircuitClosuresMatroid
sage: M = CircuitClosuresMatroid(matroids.catalog.RelaxedNonFano())
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
sage: N = M.relabel({'g': 'x', 0: 'z'})  # 'g': 'x' is ignored
sage: from sage.matroids.utilities import cmp_elements_key
sage: sorted(N.groundset(), key=cmp_elements_key)
[1, 2, 3, 4, 5, 6, 'z']
sage: M.is_isomorphic(N)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.circuit_closures_matroid import CircuitClosuresMatroid
>>> M = CircuitClosuresMatroid(matroids.catalog.RelaxedNonFano())
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
>>> N = M.relabel({'g': 'x', Integer(0): 'z'})  # 'g': 'x' is ignored
>>> from sage.matroids.utilities import cmp_elements_key
>>> sorted(N.groundset(), key=cmp_elements_key)
[1, 2, 3, 4, 5, 6, 'z']
>>> M.is_isomorphic(N)
True
```

3.3 Circuits matroids

Matroids are characterized by a list of circuits, which are minimal dependent sets. The `CircuitsMatroid` class implements matroids using this information as data.

A `CircuitsMatroid` can be created from another matroid or from a list of circuits. For a full description of allowed inputs, see *below*. It is recommended to use the *Matroid()* function for a more flexible way of constructing a `CircuitsMatroid` and other classes of matroids. For direct access to the `CircuitsMatroid` constructor, run:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
```

AUTHORS:

- Giorgos Mousa (2023-12-23): initial version

class sage.matroids.circuits_matroid.**CircuitsMatroid**

Bases: *Matroid*

A matroid defined by its circuits.

INPUT:

- `M` – matroid (default: `None`)
- `groundset` – list (default: `None`); the groundset of the matroid
- `circuits` – list (default: `None`); the collection of circuits of the matroid
- `nsc_defined` – boolean (default: `False`); whether the matroid was defined by its nonspanning circuits

Note

For a more flexible means of input, use the `Matroid()` function.

bases_iterator()

Return an iterator over the bases of the matroid.

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.Uniform(2, 4))
sage: it = M.bases_iterator()
sage: it.__next__()
frozenset({0, 1})
sage: sorted(M.bases_iterator(), key=str)
[frozenset({0, 1}),
 frozenset({0, 2}),
 frozenset({0, 3}),
 frozenset({1, 2}),
 frozenset({1, 3}),
 frozenset({2, 3})]
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.Uniform(Integer(2), Integer(4)))
>>> it = M.bases_iterator()
>>> it.__next__()
frozenset({0, 1})
>>> sorted(M.bases_iterator(), key=str)
[frozenset({0, 1}),
 frozenset({0, 2}),
 frozenset({0, 3}),
 frozenset({1, 2}),
 frozenset({1, 3}),
 frozenset({2, 3})]
```

broken_circuit_complex(*ordering=None*, *reduced=False*)

Return the broken circuit complex of `self`.

The broken circuit complex of a matroid with a total ordering $<$ on the groundset is obtained from the *NBC sets* under subset inclusion.

INPUT:

- `ordering` – list (optional); a total ordering of the groundset
- `reduced` – boolean (default: `False`); whether to return the reduced broken circuit complex (the link at the smallest element)

OUTPUT: a simplicial complex of the NBC sets under inclusion

EXAMPLES:

```
sage: M = Matroid(circuits=[[1, 2, 3], [3, 4, 5], [1, 2, 4, 5]])
sage: M.broken_circuit_complex()
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
sage: M.broken_circuit_complex([5, 4, 3, 2, 1])
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
sage: M.broken_circuit_complex([5, 4, 3, 2, 1], reduced=True)
Simplicial complex with vertex set (1, 2, 3, 4)
 and facets {(1, 3), (1, 4), (2, 3), (2, 4)}
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1), Integer(2), Integer(3)], [Integer(3),␣
↪Integer(4), Integer(5)], [Integer(1), Integer(2), Integer(4), Integer(5)]])
>>> M.broken_circuit_complex()
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
>>> M.broken_circuit_complex([Integer(5), Integer(4), Integer(3), Integer(2),␣
↪Integer(1)])
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
>>> M.broken_circuit_complex([Integer(5), Integer(4), Integer(3), Integer(2),␣
↪Integer(1)], reduced=True)
Simplicial complex with vertex set (1, 2, 3, 4)
 and facets {(1, 3), (1, 4), (2, 3), (2, 4)}
```

For a matroid with loops, the broken circuit complex is not defined, and the method yields an error:

```
sage: M = Matroid(groundset=[0, 1, 2], circuits=[[0]])
sage: M.broken_circuit_complex()
Traceback (most recent call last):
...
ValueError: broken circuit complex of matroid with loops is not defined
```

```
>>> from sage.all import *
>>> M = Matroid(groundset=[Integer(0), Integer(1), Integer(2)],␣
↪circuits=[[Integer(0)]])
>>> M.broken_circuit_complex()
Traceback (most recent call last):
...
ValueError: broken circuit complex of matroid with loops is not defined
```

circuits(*k=None*)

Return the circuits of the matroid.

INPUT:

- `k` – integer (optional); the length of the circuits

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.Uniform(2, 4))
sage: M.circuits()
SetSystem of 4 sets over 4 elements
sage: list(M.circuits(0))
[]
sage: sorted(M.circuits(3), key=str)
[frozenset({0, 1, 2}),
 frozenset({0, 1, 3}),
 frozenset({0, 2, 3}),
 frozenset({1, 2, 3})]
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.Uniform(Integer(2), Integer(4)))
>>> M.circuits()
SetSystem of 4 sets over 4 elements
>>> list(M.circuits(Integer(0)))
[]
>>> sorted(M.circuits(Integer(3)), key=str)
[frozenset({0, 1, 2}),
 frozenset({0, 1, 3}),
 frozenset({0, 2, 3}),
 frozenset({1, 2, 3})]
```

circuits_iterator(*k=None*)

Return an iterator over the circuits of the matroid.

INPUT:

- `k` – integer (optional); the length of the circuits

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.Uniform(2, 4))
sage: sum(1 for C in M.circuits_iterator())
4
sage: list(M.circuits_iterator(0))
[]
sage: sorted(M.circuits_iterator(3), key=str)
[frozenset({0, 1, 2}),
 frozenset({0, 1, 3}),
 frozenset({0, 2, 3}),
 frozenset({1, 2, 3})]
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.Uniform(Integer(2), Integer(4)))
>>> sum(Integer(1) for C in M.circuits_iterator())
4
>>> list(M.circuits_iterator(Integer(0)))
[]
>>> sorted(M.circuits_iterator(Integer(3)), key=str)
[frozenset({0, 1, 2}),
 frozenset({0, 1, 3}),
 frozenset({0, 2, 3}),
 frozenset({1, 2, 3})]
```

dependent_sets(*k*)

Return the dependent sets of fixed size.

INPUT:

- `k` – integer

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.catalog.Vamos())
sage: M.dependent_sets(3)
SetSystem of 0 sets over 8 elements
sage: sorted([sorted(X) for X in M.dependent_sets(4)])
[['a', 'b', 'c', 'd'], ['a', 'b', 'e', 'f'], ['a', 'b', 'g', 'h'],
 ['c', 'd', 'e', 'f'], ['e', 'f', 'g', 'h']]
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.catalog.Vamos())
>>> M.dependent_sets(Integer(3))
SetSystem of 0 sets over 8 elements
>>> sorted([sorted(X) for X in M.dependent_sets(Integer(4))])
[['a', 'b', 'c', 'd'], ['a', 'b', 'e', 'f'], ['a', 'b', 'g', 'h'],
 ['c', 'd', 'e', 'f'], ['e', 'f', 'g', 'h']]
```

full_rank()

Return the rank of the matroid.

The *rank* of the matroid is the size of the largest independent subset of the groundset.

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.Theta(20)
sage: M.full_rank()
20
```

```
>>> from sage.all import *
>>> M = matroids.Theta(Integer(20))
>>> M.full_rank()
20
```

girth()

Return the girth of the matroid.

The girth is the size of the smallest circuit. In case the matroid has no circuits the girth is ∞.

EXAMPLES:

```
sage: matroids.Theta(10).girth()
3
```

```
>>> from sage.all import *
>>> matroids.Theta(Integer(10)).girth()
3
```

REFERENCES:

[Oxl2011], p. 327.

groundset()

Return the groundset of the matroid.

The groundset is the set of elements that comprise the matroid.

OUTPUT: set

EXAMPLES:

```
sage: M = matroids.Theta(2)
sage: sorted(M.groundset())
['x0', 'x1', 'y0', 'y1']
```

```
>>> from sage.all import *
>>> M = matroids.Theta(Integer(2))
>>> sorted(M.groundset())
['x0', 'x1', 'y0', 'y1']
```

independent_sets(*k=-1*)

Return the independent sets of the matroid.

INPUT:

- `k` – integer (optional); if specified, return the size-k independent sets of the matroid

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.catalog.Pappus())
sage: M.independent_sets(4)
SetSystem of 0 sets over 9 elements
sage: M.independent_sets(3)
SetSystem of 75 sets over 9 elements
sage: frozenset({'a', 'c', 'e'}) in _
True
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.catalog.Pappus())
>>> M.independent_sets(Integer(4))
SetSystem of 0 sets over 9 elements
>>> M.independent_sets(Integer(3))
SetSystem of 75 sets over 9 elements
>>> frozenset({'a', 'c', 'e'}) in _
True
```

See also

`M.bases()`

is_paving()

Return if `self` is paving.

A matroid is paving if each of its circuits has size r or $r + 1$.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.catalog.Vamos())
sage: M.is_paving()
True
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.catalog.Vamos())
>>> M.is_paving()
True
```

is_valid(*certificate=False*)

Test if `self` obeys the matroid axioms.

For a matroid defined by its circuits, we check the circuit axioms.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: boolean, or (boolean, dictionary)

EXAMPLES:

```
sage: C = [[1, 2, 3], [3, 4, 5], [1, 2, 4, 5]]
sage: M = Matroid(circuits=C)
sage: M.is_valid()
True
sage: C = [[1, 2], [1, 2, 3], [3, 4, 5], [1, 2, 4, 5]]
sage: M = Matroid(circuits=C)
sage: M.is_valid()
False
sage: C = [[3, 6], [1, 2, 3], [3, 4, 5], [1, 2, 4, 5]]
sage: M = Matroid(circuits=C)
sage: M.is_valid()
False
sage: C = [[3, 6], [1, 2, 3], [3, 4, 5], [1, 2, 6], [6, 4, 5], [1, 2, 4, 5]]
sage: M = Matroid(circuits=C)
sage: M.is_valid()
True
sage: C = [[], [1, 2, 3], [3, 4, 5], [1, 2, 4, 5]]
sage: M = Matroid(circuits=C)
sage: M.is_valid()
False
sage: C = [[1, 2, 3], [3, 4, 5]]
sage: M = Matroid(circuits=C)
sage: M.is_valid(certificate=True)
(False,
 {'circuit 1': frozenset({...}),
  'circuit 2': frozenset({...}),
  'element': 3,
  'error': 'elimination axiom failed'})
```

```
>>> from sage.all import *
>>> C = [[Integer(1), Integer(2), Integer(3)], [Integer(3), Integer(4),␣
↪Integer(5)], [Integer(1), Integer(2), Integer(4), Integer(5)]]
>>> M = Matroid(circuits=C)
>>> M.is_valid()
True
>>> C = [[Integer(1), Integer(2)], [Integer(1), Integer(2), Integer(3)],␣
↪[Integer(3), Integer(4), Integer(5)], [Integer(1), Integer(2), Integer(4),␣
↪Integer(5)]]
>>> M = Matroid(circuits=C)
>>> M.is_valid()
False
>>> C = [[Integer(3), Integer(6)], [Integer(1), Integer(2), Integer(3)],␣
↪[Integer(3), Integer(4), Integer(5)], [Integer(1), Integer(2), Integer(4),␣
↪Integer(5)]]
>>> M = Matroid(circuits=C)
>>> M.is_valid()
False
>>> C = [[Integer(3), Integer(6)], [Integer(1), Integer(2), Integer(3)],␣
↪[Integer(3), Integer(4), Integer(5)], [Integer(1), Integer(2), Integer(6)],␣
↪[Integer(6), Integer(4), Integer(5)], [Integer(1), Integer(2), Integer(4),␣
↪Integer(5)]]
>>> M = Matroid(circuits=C)
>>> M.is_valid()
True
>>> C = [[], [Integer(1), Integer(2), Integer(3)], [Integer(3), Integer(4),␣
↪Integer(5)], [Integer(1), Integer(2), Integer(4), Integer(5)]]
>>> M = Matroid(circuits=C)
>>> M.is_valid()
False
>>> C = [[Integer(1), Integer(2), Integer(3)], [Integer(3), Integer(4),␣
↪Integer(5)]]
>>> M = Matroid(circuits=C)
>>> M.is_valid(certificate=True)
(False,
 {'circuit 1': frozenset({...}),
  'circuit 2': frozenset({...}),
  'element': 3,
  'error': 'elimination axiom failed'})
```

no_broken_circuits_facets(*ordering=None*, *reduced=False*)

Return the no broken circuits (NBC) facets of `self`.

INPUT:

- `ordering` – list (optional); a total ordering of the groundset
- `reduced` – boolean (default: `False`)

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: M = Matroid(circuits=[[0, 1, 2]])
sage: M.no_broken_circuits_facets(ordering=[1, 0, 2])
```

(continues on next page)

(continued from previous page)

```
SetSystem of 2 sets over 3 elements
sage: sorted([sorted(X) for X in _])
[[0, 1], [1, 2]]
sage: M.no_broken_circuits_facets(ordering=[1, 0, 2], reduced=True)
SetSystem of 2 sets over 3 elements
sage: sorted([sorted(X) for X in _])
[[0], [2]]
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(0), Integer(1), Integer(2)]])
>>> M.no_broken_circuits_facets(ordering=[Integer(1), Integer(0), Integer(2)])
SetSystem of 2 sets over 3 elements
>>> sorted([sorted(X) for X in _])
[[0, 1], [1, 2]]
>>> M.no_broken_circuits_facets(ordering=[Integer(1), Integer(0), Integer(2)],
↪ reduced=True)
SetSystem of 2 sets over 3 elements
>>> sorted([sorted(X) for X in _])
[[0], [2]]
```

no_broken_circuits_sets(*ordering=None*, *reduced=False*)

Return the no broken circuits (NBC) sets of `self`.

An NBC set is a subset A of the groundset under some total ordering $<$ such that A contains no broken circuit.

INPUT:

- `ordering` – list (optional); a total ordering of the groundset
- `reduced` – boolean (default: `False`)

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: M = Matroid(circuits=[[1, 2, 3], [3, 4, 5], [1, 2, 4, 5]])
sage: SimplicialComplex(M.no_broken_circuits_sets())
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
sage: SimplicialComplex(M.no_broken_circuits_sets([5, 4, 3, 2, 1]))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1), Integer(2), Integer(3)], [Integer(3),␣
↪Integer(4), Integer(5)], [Integer(1), Integer(2), Integer(4), Integer(5)]])
>>> SimplicialComplex(M.no_broken_circuits_sets())
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5)}
>>> SimplicialComplex(M.no_broken_circuits_sets([Integer(5), Integer(4),␣
↪Integer(3), Integer(2), Integer(1)]))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

```
sage: M = Matroid(circuits=[[1, 2, 3], [1, 4, 5], [2, 3, 4, 5]])
sage: SimplicialComplex(M.no_broken_circuits_sets([5, 4, 3, 2, 1]))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (2, 3, 5), (2, 4, 5), (3, 4, 5)}
```

```
>>> from sage.all import *
>>> M = Matroid(circuits=[[Integer(1), Integer(2), Integer(3)], [Integer(1),␣
↪Integer(4), Integer(5)], [Integer(2), Integer(3), Integer(4), Integer(5)]])
>>> SimplicialComplex(M.no_broken_circuits_sets([Integer(5), Integer(4),␣
↪Integer(3), Integer(2), Integer(1)]))
Simplicial complex with vertex set (1, 2, 3, 4, 5)
 and facets {(1, 3, 5), (2, 3, 5), (2, 4, 5), (3, 4, 5)}
```

nonspanning_circuits()

Return the nonspanning circuits of the matroid.

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.Uniform(2, 4))
sage: M.nonspanning_circuits()
SetSystem of 0 sets over 4 elements
sage: M = matroids.Theta(5)
sage: M.nonspanning_circuits()
SetSystem of 15 sets over 10 elements
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.Uniform(Integer(2), Integer(4)))
>>> M.nonspanning_circuits()
SetSystem of 0 sets over 4 elements
>>> M = matroids.Theta(Integer(5))
>>> M.nonspanning_circuits()
SetSystem of 15 sets over 10 elements
```

nonspanning_circuits_iterator()

Return an iterator over the nonspanning circuits of the matroid.

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.Uniform(2, 4))
sage: list(M.nonspanning_circuits_iterator())
[]
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.Uniform(Integer(2), Integer(4)))
>>> list(M.nonspanning_circuits_iterator())
[]
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.catalog.RelaxedNonFano())
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
sage: N = M.relabel({'g': 'x', 0: 'z'})  # 'g': 'x' is ignored
sage: from sage.matroids.utilities import cmp_elements_key
sage: sorted(N.groundset(), key=cmp_elements_key)
[1, 2, 3, 4, 5, 6, 'z']
sage: M.is_isomorphic(N)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.circuits_matroid import CircuitsMatroid
>>> M = CircuitsMatroid(matroids.catalog.RelaxedNonFano())
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
>>> N = M.relabel({'g': 'x', Integer(0): 'z'})  # 'g': 'x' is ignored
>>> from sage.matroids.utilities import cmp_elements_key
>>> sorted(N.groundset(), key=cmp_elements_key)
[1, 2, 3, 4, 5, 6, 'z']
>>> M.is_isomorphic(N)
True
```

3.4 Flats matroids

Matroids are characterized by a set of flats, which are sets invariant under closure. The `FlatsMatroid` class implements matroids using this information as data.

A `FlatsMatroid` can be created from another matroid or from a dictionary of flats. For a full description of allowed inputs, see *below*. It is recommended to use the *Matroid()* function for a more flexible way of constructing a `FlatsMatroid` and other classes of matroids. For direct access to the `FlatsMatroid` constructor, run:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
```

AUTHORS:

- Giorgos Mousa (2024-01-01): initial version

class sage.matroids.flats_matroid.**FlatsMatroid**

Bases: *Matroid*

INPUT:

- `M` – matroid (default: `None`)
- `groundset` – list (default: `None`); the groundset of the matroid
- `flats` – (default: `None`) the dictionary of the lists of flats (indexed by their rank), or the list of all flats, or the lattice of flats of the matroid

Note

For a more flexible means of input, use the `Matroid()` function.

flats(*k*)

Return the flats of the matroid of specified rank.

INPUT:

- `k` – integer

OUTPUT: `SetSystem`

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.Uniform(3, 4))
sage: sorted(M.flats(2), key=str)
[frozenset({0, 1}),
 frozenset({0, 2}),
 frozenset({0, 3}),
 frozenset({1, 2}),
 frozenset({1, 3}),
 frozenset({2, 3})]
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.Uniform(Integer(3), Integer(4)))
>>> sorted(M.flats(Integer(2)), key=str)
[frozenset({0, 1}),
 frozenset({0, 2}),
 frozenset({0, 3}),
 frozenset({1, 2}),
 frozenset({1, 3}),
 frozenset({2, 3})]
```

flats_iterator(*k*)

Return an iterator over the flats of the matroid of specified rank.

INPUT:

- `k` – integer

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.Uniform(3, 4))
sage: sorted([list(F) for F in M.flats_iterator(2)])
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.Uniform(Integer(3), Integer(4)))
>>> sorted([list(F) for F in M.flats_iterator(Integer(2))])
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

full_rank()

Return the rank of the matroid.

The *rank* of the matroid is the size of the largest independent subset of the groundset.

OUTPUT: integer

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.Theta(6))
sage: M.full_rank()
6
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.Theta(Integer(6)))
>>> M.full_rank()
6
```

groundset()

Return the groundset of the matroid.

The groundset is the set of elements that comprise the matroid.

OUTPUT: `frozenset`

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.Theta(2))
sage: sorted(M.groundset())
['x0', 'x1', 'y0', 'y1']
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.Theta(Integer(2)))
>>> sorted(M.groundset())
['x0', 'x1', 'y0', 'y1']
```

is_valid(*certificate=False*)

Test if `self` obeys the matroid axioms.

For a matroid defined by its flats, we check the flats axioms.

If the lattice of flats has already been computed, we instead perform the equivalent check of whether it forms a geometric lattice.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: boolean, or (boolean, dictionary)

EXAMPLES:

```
sage: Matroid(flats={0: [[]], 1: [[0], [1]], 2: [[0, 1]]}).is_valid()
True
sage: Matroid(flats={0: [''], 1: ['a', 'b'], 2: ['ab']}).is_valid()
True
sage: M = Matroid(flats={0: [[]], 1: [[0], [1]]})  # missing groundset
sage: M.is_valid()
False
sage: M = Matroid(flats={0: [''],
....:                    1: ['0','1','2','3','4','5','6','7','8','9','a','b',
↪'c'],
....:                    2: ['45','46','47','4c','56','57','5c','67','6c','7c
↪',
....:                        '048','149','24a','34b','059','15a','25b','358',
....:                        '06a','16b','268','369','07b','178','279','37a',
....:                        '0123c','89abc'],
....:                    3: ['0123456789abc']})
sage: M.is_valid()
True
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: FlatsMatroid(matroids.catalog.NonVamos()).is_valid()
True
```

```
>>> from sage.all import *
>>> Matroid(flats={Integer(0): [[]], Integer(1): [[Integer(0)], [Integer(1)]],
↪ Integer(2): [[Integer(0), Integer(1)]]}).is_valid()
True
>>> Matroid(flats={Integer(0): [''], Integer(1): ['a', 'b'], Integer(2): ['ab
↪']}).is_valid()
True
>>> M = Matroid(flats={Integer(0): [[]], Integer(1): [[Integer(0)],␣
↪[Integer(1)]]})  # missing groundset
>>> M.is_valid()
False
>>> M = Matroid(flats={Integer(0): [''],
...                    Integer(1): ['0','1','2','3','4','5','6','7','8','9','a
↪','b','c'],
...                    Integer(2): ['45','46','47','4c','56','57','5c','67',
↪'6c','7c',
...                        '048','149','24a','34b','059','15a','25b','358',
...                        '06a','16b','268','369','07b','178','279','37a',
...                        '0123c','89abc'],
...                    Integer(3): ['0123456789abc']})
>>> M.is_valid()
True
```

(continues on next page)

(continued from previous page)

```
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> FlatsMatroid(matroids.catalog.NonVamos()).is_valid()
True
```

If we input a list or a lattice of flats, the method checks whether the lattice of flats is geometric:

```
sage: Matroid(flats=[[], [0], [1], [0, 1]]).is_valid()
True
sage: Matroid(flats=['', 'a', 'b', 'ab']).is_valid()
True
sage: M = Matroid(flats=['',  # missing an extension of flat ['5'] by '6'
....:                    '0','1','2','3','4','5','6','7','8','9','a','b','c',
....:                    '45','46','47','4c','57','5c','67','6c','7c',
....:                    '048','149','24a','34b','059','15a','25b','358',
....:                    '06a','16b','268','369','07b','178','279','37a',
....:                    '0123c','89abc',
....:                    '0123456789abc'])
sage: M.is_valid(certificate=True)
(False, {'error': 'the lattice of flats is not geometric'})
sage: Matroid(matroids.catalog.Fano().lattice_of_flats()).is_valid()
True
```

```
>>> from sage.all import *
>>> Matroid(flats=[[], [Integer(0)], [Integer(1)], [Integer(0), Integer(1)]]).
↪is_valid()
True
>>> Matroid(flats=['', 'a', 'b', 'ab']).is_valid()
True
>>> M = Matroid(flats=['',  # missing an extension of flat ['5'] by '6'
...                    '0','1','2','3','4','5','6','7','8','9','a','b','c',
...                    '45','46','47','4c','57','5c','67','6c','7c',
...                    '048','149','24a','34b','059','15a','25b','358',
...                    '06a','16b','268','369','07b','178','279','37a',
...                    '0123c','89abc',
...                    '0123456789abc'])
>>> M.is_valid(certificate=True)
(False, {'error': 'the lattice of flats is not geometric'})
>>> Matroid(matroids.catalog.Fano().lattice_of_flats()).is_valid()
True
```

Some invalid lists of flats are recognized before calling `is_valid`, upon the attempted matroid definition:

```
sage: M = Matroid(flats=[[], [0], [1]])  # missing groundset
Traceback (most recent call last):
...
ValueError: not a join-semilattice: no top element
sage: Matroid(flats=[[0], [1], [0, 1]])  # missing an intersection
Traceback (most recent call last):
...
ValueError: not a meet-semilattice: no bottom element
sage: Matroid(flats=[[], [0, 1], [2], [0], [1], [0, 1, 2]])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: the poset is not ranked
```

```
>>> from sage.all import *
>>> M = Matroid(flats=[[], [Integer(0)], [Integer(1)]])  # missing groundset
Traceback (most recent call last):
...
ValueError: not a join-semilattice: no top element
>>> Matroid(flats=[[Integer(0)], [Integer(1)], [Integer(0), Integer(1)]])  #␣
↪missing an intersection
Traceback (most recent call last):
...
ValueError: not a meet-semilattice: no bottom element
>>> Matroid(flats=[[], [Integer(0), Integer(1)], [Integer(2)], [Integer(0)],␣
↪[Integer(1)], [Integer(0), Integer(1), Integer(2)]])
Traceback (most recent call last):
...
ValueError: the poset is not ranked
```

lattice_of_flats()

Return the lattice of flats of the matroid.

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.catalog.Fano())
sage: M.lattice_of_flats()
Finite lattice containing 16 elements
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.catalog.Fano())
>>> M.lattice_of_flats()
Finite lattice containing 16 elements
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.catalog.RelaxedNonFano())
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
sage: N = M.relabel({'g': 'x', 0: 'z'})  # 'g': 'x' is ignored
```

(continues on next page)

(continued from previous page)

```
sage: from sage.matroids.utilities import cmp_elements_key
sage: sorted(N.groundset(), key=cmp_elements_key)
[1, 2, 3, 4, 5, 6, 'z']
sage: M.is_isomorphic(N)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.catalog.RelaxedNonFano())
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
>>> N = M.relabel({'g': 'x', Integer(0): 'z'})  # 'g': 'x' is ignored
>>> from sage.matroids.utilities import cmp_elements_key
>>> sorted(N.groundset(), key=cmp_elements_key)
[1, 2, 3, 4, 5, 6, 'z']
>>> M.is_isomorphic(N)
True
```

whitney_numbers()

Return the Whitney numbers of the first kind of the matroid.

The Whitney numbers of the first kind – here encoded as a vector $(w_0 = 1, \ldots, w_r)$ – are numbers of alternating sign, where w_i is the value of the coefficient of the $(r-i)$-th degree term of the matroid's characteristic polynomial. Moreover, $|w_i|$ is the number of $(i-1)$-dimensional faces of the broken circuit complex of the matroid.

OUTPUT: list of integers

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.catalog.BetsyRoss())
sage: M.whitney_numbers()
[1, -11, 35, -25]
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.catalog.BetsyRoss())
>>> M.whitney_numbers()
[1, -11, 35, -25]
```

whitney_numbers2()

Return the Whitney numbers of the second kind of the matroid.

The Whitney numbers of the second kind are here encoded as a vector $(W_0, ..., W_r)$, where W_i is the number of flats of rank i, and r is the rank of the matroid.

OUTPUT: list of integers

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.catalog.XY13())
sage: M.whitney_numbers2()
[1, 13, 78, 250, 394, 191, 1]
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.catalog.XY13())
>>> M.whitney_numbers2()
[1, 13, 78, 250, 394, 191, 1]
```

3.5 Gammoids

Let D be a directed graph and let E and T be not necessarily disjoint sets of vertices of D. Say a subset X of E is in a collection I if X can be linked into a subset of T. This defines a gammoid (E, I), where E is the groundset of a matroid and I is its independent sets.

Some authors use a reverse convention, where instead of a set T of roots, they have a starting set S that is linked into subsets of E. Here we use the convention that the vertices T are at the end of the directed paths, not the beginning.

To construct a gammoid, first import Gammoid from *sage.matroids.gammoid*. A digraph and a list of roots from the vertex set are required for input:

```
sage: from sage.matroids.gammoid import *
sage: edgelist = [(0,1),(1,2),(2,3),(3,4)]
sage: D = DiGraph(edgelist)
sage: M = Gammoid(D, roots=[4]); M
Gammoid of rank 1 on 5 elements
sage: N = Gammoid(D, roots=[4], groundset=range(1,5)); N
Gammoid of rank 1 on 4 elements
sage: M.delete(0) == N
True
sage: N.is_isomorphic(matroids.Uniform(1,4))
True
sage: O = Gammoid(D, roots=[3]); O
Gammoid of rank 1 on 5 elements
sage: O.rank([0])
1
sage: O.rank([4])
0
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import *
>>> edgelist = [(Integer(0),Integer(1)),(Integer(1),Integer(2)),(Integer(2),
↪Integer(3)),(Integer(3),Integer(4))]
>>> D = DiGraph(edgelist)
>>> M = Gammoid(D, roots=[Integer(4)]); M
Gammoid of rank 1 on 5 elements
>>> N = Gammoid(D, roots=[Integer(4)], groundset=range(Integer(1),Integer(5))); N
Gammoid of rank 1 on 4 elements
>>> M.delete(Integer(0)) == N
True
>>> N.is_isomorphic(matroids.Uniform(Integer(1),Integer(4)))
True
>>> O = Gammoid(D, roots=[Integer(3)]); O
Gammoid of rank 1 on 5 elements
>>> O.rank([Integer(0)])
```

(continues on next page)

(continued from previous page)

```
1
>>> O.rank([Integer(4)])
0
```

AUTHORS:

- Zachary Gershkoff (2017-08-25): initial version

class sage.matroids.gammoid.**Gammoid**(*D*, *roots*, *groundset=None*)

Bases: *Matroid*

The gammoid class.

INPUT:

- `D` – a loopless digraph representing the gammoid
- `roots` – a subset of the vertices
- `groundset` – (optional) a subset of the vertices

OUTPUT: *Gammoid*; if `groundset` is not specified, the entire vertex set is used (and the gammoid will be strict)

EXAMPLES:

```
sage: from sage.matroids.gammoid import Gammoid
sage: D = digraphs.TransitiveTournament(5)
sage: M = Gammoid(D, roots=[3, 4]); M
Gammoid of rank 2 on 5 elements
sage: M.is_isomorphic(matroids.Uniform(2, 5))
True
sage: D.add_vertex(6)
sage: N = Gammoid(D, roots=[3, 4])
sage: N.loops()
frozenset({6})
sage: O = Gammoid(D, roots=[3, 4, 6])
sage: O.coloops()
frozenset({6})
sage: O.full_rank()
3
sage: P = Gammoid(D, roots=[3, 4], groundset=[0, 2, 3]); P
Gammoid of rank 2 on 3 elements
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import Gammoid
>>> D = digraphs.TransitiveTournament(Integer(5))
>>> M = Gammoid(D, roots=[Integer(3), Integer(4)]); M
Gammoid of rank 2 on 5 elements
>>> M.is_isomorphic(matroids.Uniform(Integer(2), Integer(5)))
True
>>> D.add_vertex(Integer(6))
>>> N = Gammoid(D, roots=[Integer(3), Integer(4)])
>>> N.loops()
frozenset({6})
>>> O = Gammoid(D, roots=[Integer(3), Integer(4), Integer(6)])
>>> O.coloops()
```

(continues on next page)

(continued from previous page)

```
frozenset({6})
>>> O.full_rank()
3
>>> P = Gammoid(D, roots=[Integer(3), Integer(4)], groundset=[Integer(0),␣
↪Integer(2), Integer(3)]); P
Gammoid of rank 2 on 3 elements
```

digraph()

Return the digraph associated with the gammoid.

EXAMPLES:

```
sage: from sage.matroids.gammoid import Gammoid
sage: edgelist = [(0, 4), (0, 5), (1, 0), (1, 4), (2, 0), (2, 1), (2, 3),
....:             (2, 6), (3, 4), (3, 5), (4, 0), (5, 2), (6, 5)]
sage: D = DiGraph(edgelist)
sage: M = Gammoid(D, roots=[4, 5, 6])
sage: M.digraph() == D
True
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import Gammoid
>>> edgelist = [(Integer(0), Integer(4)), (Integer(0), Integer(5)),␣
↪(Integer(1), Integer(0)), (Integer(1), Integer(4)), (Integer(2),␣
↪Integer(0)), (Integer(2), Integer(1)), (Integer(2), Integer(3)),
...              (Integer(2), Integer(6)), (Integer(3), Integer(4)),␣
↪(Integer(3), Integer(5)), (Integer(4), Integer(0)), (Integer(5),␣
↪Integer(2)), (Integer(6), Integer(5))]
>>> D = DiGraph(edgelist)
>>> M = Gammoid(D, roots=[Integer(4), Integer(5), Integer(6)])
>>> M.digraph() == D
True
```

digraph_plot()

Plot the graph with color-coded vertices.

Vertices that are elements but not roots will be shown as blue. Vertices that are roots but not elements are red. Vertices that are both are pink, and vertices that are neither are grey.

EXAMPLES:

```
sage: from sage.matroids.gammoid import Gammoid
sage: D = digraphs.TransitiveTournament(4)
sage: M = Gammoid(D, roots=[2, 3])
sage: M.digraph_plot()
Graphics object consisting of 11 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import Gammoid
>>> D = digraphs.TransitiveTournament(Integer(4))
>>> M = Gammoid(D, roots=[Integer(2), Integer(3)])
>>> M.digraph_plot()
Graphics object consisting of 11 graphics primitives
```

gammoid_extension(*vertex*, *neighbors=[]*)

Return a gammoid extended by an element.

The new element can be a vertex of the digraph that is not in the starting set, or it can be a new source vertex.

INPUT:

- `vertex` – a vertex of the gammoid's digraph that is not already in the groundset, or a new vertex
- `neighbors` – (optional) a set of vertices of the digraph

OUTPUT:

A *Gammoid*. If `vertex` is not already in the graph, then the new vertex will be have edges to `neighbors`. The new vertex will have in degree 0 regardless of whether or not `neighbors` is specified.

EXAMPLES:

```
sage: from sage.matroids.gammoid import Gammoid
sage: edgedict = {1: [2], 2: [3], 4: [1, 5], 5: [2, 3, 8],
....:             6: [4, 7], 7: [5, 8]}
sage: D = DiGraph(edgedict)
sage: M = Gammoid(D, roots=[2, 3, 4], groundset=range(2, 9))
sage: M1 = M.gammoid_extension(1)
sage: M1.groundset()
frozenset({1, 2, 3, 4, 5, 6, 7, 8})
sage: N = Gammoid(D, roots=[2, 3, 4])
sage: M1 == N
True
sage: M1.delete(1)
Gammoid of rank 3 on 7 elements
sage: M == M1.delete(1)
True
sage: M2 = M.gammoid_extension(9); sorted(M2.loops())
[8, 9]
sage: M4 = M.gammoid_extension(9, [1, 2, 3])
sage: M4.digraph().neighbors_out(9)
[1, 2, 3]
sage: M4.digraph().neighbors_in(9)
[]
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import Gammoid
>>> edgedict = {Integer(1): [Integer(2)], Integer(2): [Integer(3)],␣
↪Integer(4): [Integer(1), Integer(5)], Integer(5): [Integer(2), Integer(3),␣
↪Integer(8)],
...             Integer(6): [Integer(4), Integer(7)], Integer(7): [Integer(5),
↪ Integer(8)]}
>>> D = DiGraph(edgedict)
>>> M = Gammoid(D, roots=[Integer(2), Integer(3), Integer(4)],␣
↪groundset=range(Integer(2), Integer(9)))
>>> M1 = M.gammoid_extension(Integer(1))
>>> M1.groundset()
frozenset({1, 2, 3, 4, 5, 6, 7, 8})
>>> N = Gammoid(D, roots=[Integer(2), Integer(3), Integer(4)])
>>> M1 == N
```

(continues on next page)

(continued from previous page)

```
True
>>> M1.delete(Integer(1))
Gammoid of rank 3 on 7 elements
>>> M == M1.delete(Integer(1))
True
>>> M2 = M.gammoid_extension(Integer(9)); sorted(M2.loops())
[8, 9]
>>> M4 = M.gammoid_extension(Integer(9), [Integer(1), Integer(2), Integer(3)])
>>> M4.digraph().neighbors_out(Integer(9))
[1, 2, 3]
>>> M4.digraph().neighbors_in(Integer(9))
[]
```

gammoid_extensions(*vertices=None*)

Return an iterator of Gammoid extensions.

This will only consider extensions from vertices that are already present in the digraph.

INPUT:

- `vertices` – (optional) a list of vertices not in the digraph

OUTPUT:

An iterator of Gammoids. If `vertices` is not specified, every vertex not already in the groundset will be considered.

EXAMPLES:

```
sage: from sage.matroids.gammoid import Gammoid
sage: M = Gammoid(digraphs.TransitiveTournament(5), roots=[3, 4],
....:             groundset=[0, 1, 4])
sage: [sorted(M1.groundset()) for M1 in M.gammoid_extensions()]
[[0, 1, 2, 4], [0, 1, 3, 4]]
sage: N = Gammoid(digraphs.TransitiveTournament(5), roots=[3, 4])
sage: [sorted(N1.groundset()) for N1 in N.gammoid_extensions()]
[]
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import Gammoid
>>> M = Gammoid(digraphs.TransitiveTournament(Integer(5)), roots=[Integer(3),␣
↪Integer(4)],
...             groundset=[Integer(0), Integer(1), Integer(4)])
>>> [sorted(M1.groundset()) for M1 in M.gammoid_extensions()]
[[0, 1, 2, 4], [0, 1, 3, 4]]
>>> N = Gammoid(digraphs.TransitiveTournament(Integer(5)), roots=[Integer(3),␣
↪Integer(4)])
>>> [sorted(N1.groundset()) for N1 in N.gammoid_extensions()]
[]
```

```
sage: from sage.matroids.gammoid import Gammoid
sage: edgelist =[(i, i+1) for i in range(10)]
sage: M = Gammoid(DiGraph(edgelist), roots=[9], groundset=[0])
sage: sum(1 for M1 in M.gammoid_extensions(vertices=[3, 4, 5]))
```

(continues on next page)

(continued from previous page)

```
3
sage: sum(1 for M1 in M.gammoid_extensions())
9
sage: set([M1.is_isomorphic(matroids.Uniform(1, 2))
....:      for M1 in M.gammoid_extensions()])
{True}
sage: len([M1 for M1 in M.gammoid_extensions([0, 1, 2])])
Traceback (most recent call last):
...
ValueError: vertices must be in the digraph and not already in the groundset
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import Gammoid
>>> edgelist =[(i, i+Integer(1)) for i in range(Integer(10))]
>>> M = Gammoid(DiGraph(edgelist), roots=[Integer(9)], groundset=[Integer(0)])
>>> sum(Integer(1) for M1 in M.gammoid_extensions(vertices=[Integer(3),␣
↪Integer(4), Integer(5)]))
3
>>> sum(Integer(1) for M1 in M.gammoid_extensions())
9
>>> set([M1.is_isomorphic(matroids.Uniform(Integer(1), Integer(2)))
...      for M1 in M.gammoid_extensions()])
{True}
>>> len([M1 for M1 in M.gammoid_extensions([Integer(0), Integer(1),␣
↪Integer(2)])])
Traceback (most recent call last):
...
ValueError: vertices must be in the digraph and not already in the groundset
```

groundset()

Return the groundset of the matroid.

EXAMPLES:

```
sage: from sage.matroids.gammoid import Gammoid
sage: edgelist = [(0, 4), (0, 5), (1, 0), (1, 4), (2, 0), (2, 1), (2, 3),
....:             (2, 6), (3, 4), (3, 5), (4, 0), (5, 2), (6, 5)]
sage: D = DiGraph(edgelist)
sage: M = Gammoid(D, roots=[4, 5, 6])
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import Gammoid
>>> edgelist = [(Integer(0), Integer(4)), (Integer(0), Integer(5)),␣
↪(Integer(1), Integer(0)), (Integer(1), Integer(4)), (Integer(2),␣
↪Integer(0)), (Integer(2), Integer(1)), (Integer(2), Integer(3)),
...              (Integer(2), Integer(6)), (Integer(3), Integer(4)),␣
↪(Integer(3), Integer(5)), (Integer(4), Integer(0)), (Integer(5),␣
↪Integer(2)), (Integer(6), Integer(5))]
>>> D = DiGraph(edgelist)
>>> M = Gammoid(D, roots=[Integer(4), Integer(5), Integer(6)])
```

(continues on next page)

(continued from previous page)

```
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6]
```

3.6 Graphic matroids

Let $G = (V, E)$ be a graph and let C be the collection of the edge sets of cycles in G. The corresponding graphic matroid $M(G)$ has groundset E and circuits C.

3.6.1 Construction

The recommended way to create a graphic matroid is by using the *Matroid()* function, with a graph G as input. This function can accept many different kinds of input to get a graphic matroid if the `graph` keyword is used, similar to the `Graph()` constructor. However, invoking the class directly is possible too. To get access to it, type:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See also *sage.matroids.advanced*.

Graphic matroids do not have a representation matrix or any of the functionality of regular matroids. It is possible to get an instance of the *RegularMatroid* class by using the `regular` keyword when constructing the matroid. It is also possible to cast a *GraphicMatroid* as a `RegularMatroid` with the `regular_matroid()` method:

```
sage: M1 = Matroid(graphs.DiamondGraph(), regular=True)
sage: M2 = Matroid(graphs.DiamondGraph())
sage: M3 = M2.regular_matroid()
```

```
>>> from sage.all import *
>>> M1 = Matroid(graphs.DiamondGraph(), regular=True)
>>> M2 = Matroid(graphs.DiamondGraph())
>>> M3 = M2.regular_matroid()
```

Below are some examples of constructing a graphic matroid.

```
sage: from sage.matroids.advanced import *
sage: edgelist = [(0, 1, 'a'), (0, 2, 'b'), (1, 2, 'c')]
sage: G = Graph(edgelist)
sage: M1 = Matroid(G)
sage: M2 = Matroid(graph=edgelist)
sage: M3 = Matroid(graphs.CycleGraph(3))
sage: M1 == M3
False
sage: M1.is_isomorphic(M3)
True
sage: M1.equals(M2)
True
sage: M1 == M2
True
sage: isinstance(M1, GraphicMatroid)
```

(continues on next page)

(continued from previous page)

```
True
sage: isinstance(M1, RegularMatroid)
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> edgelist = [(Integer(0), Integer(1), 'a'), (Integer(0), Integer(2), 'b'),␣
↪(Integer(1), Integer(2), 'c')]
>>> G = Graph(edgelist)
>>> M1 = Matroid(G)
>>> M2 = Matroid(graph=edgelist)
>>> M3 = Matroid(graphs.CycleGraph(Integer(3)))
>>> M1 == M3
False
>>> M1.is_isomorphic(M3)
True
>>> M1.equals(M2)
True
>>> M1 == M2
True
>>> isinstance(M1, GraphicMatroid)
True
>>> isinstance(M1, RegularMatroid)
False
```

Note that if there is not a complete set of unique edge labels, and there are no parallel edges, then vertex tuples will be used for the groundset. The user may wish to override this by specifying the groundset, as the vertex tuples will not be updated if the matroid is modified:

```
sage: G = graphs.DiamondGraph()
sage: M1 = Matroid(G)
sage: N1 = M1.contract((0,1))
sage: N1.graph().edges_incident(0, sort=True)
[(0, 2, (0, 2)), (0, 2, (1, 2)), (0, 3, (1, 3))]
sage: M2 = Matroid(range(G.num_edges()), G)
sage: N2 = M2.contract(0)
sage: N1.is_isomorphic(N2)
True
```

```
>>> from sage.all import *
>>> G = graphs.DiamondGraph()
>>> M1 = Matroid(G)
>>> N1 = M1.contract((Integer(0),Integer(1)))
>>> N1.graph().edges_incident(Integer(0), sort=True)
[(0, 2, (0, 2)), (0, 2, (1, 2)), (0, 3, (1, 3))]
>>> M2 = Matroid(range(G.num_edges()), G)
>>> N2 = M2.contract(Integer(0))
>>> N1.is_isomorphic(N2)
True
```

AUTHORS:

- Zachary Gershkoff (2017-07-07): initial version

class sage.matroids.graphic_matroid.**GraphicMatroid**

Bases: *Matroid*

The graphic matroid class.

INPUT:

- G – Graph
- groundset – list (optional); in 1-1 correspondence with G.edge_iterator()

OUTPUT: *GraphicMatroid* where the groundset elements are the edges of G

Note

If a disconnected graph is given as input, the instance of *GraphicMatroid* will connect the graph components and store this as its graph.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = GraphicMatroid(graphs.BullGraph()); M
Graphic matroid of rank 4 on 5 elements
sage: N = GraphicMatroid(graphs.CompleteBipartiteGraph(3,3)); N
Graphic matroid of rank 5 on 9 elements
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = GraphicMatroid(graphs.BullGraph()); M
Graphic matroid of rank 4 on 5 elements
>>> N = GraphicMatroid(graphs.CompleteBipartiteGraph(Integer(3),Integer(3))); N
Graphic matroid of rank 5 on 9 elements
```

A disconnected input will get converted to a connected graph internally:

```
sage: G1 = graphs.CycleGraph(3); G2 = graphs.DiamondGraph()
sage: G = G1.disjoint_union(G2)
sage: len(G)
7
sage: G.is_connected()
False
sage: M = GraphicMatroid(G)
sage: M
Graphic matroid of rank 5 on 8 elements
sage: H = M.graph()
sage: H
Looped multi-graph on 6 vertices
sage: H.is_connected()
True
sage: M.is_connected()
False
```

```
>>> from sage.all import *
>>> G1 = graphs.CycleGraph(Integer(3)); G2 = graphs.DiamondGraph()
```

(continues on next page)

(continued from previous page)

```
>>> G = G1.disjoint_union(G2)
>>> len(G)
7
>>> G.is_connected()
False
>>> M = GraphicMatroid(G)
>>> M
Graphic matroid of rank 5 on 8 elements
>>> H = M.graph()
>>> H
Looped multi-graph on 6 vertices
>>> H.is_connected()
True
>>> M.is_connected()
False
```

You can still locate an edge using the vertices of the input graph:

```
sage: G1 = graphs.CycleGraph(3); G2 = graphs.DiamondGraph()
sage: G = G1.disjoint_union(G2)
sage: M = Matroid(G)
sage: H = M.graph()
sage: vm = M.vertex_map()
sage: (u, v, l) = G.random_edge()
sage: H.has_edge(vm[u], vm[v])
True
```

```
>>> from sage.all import *
>>> G1 = graphs.CycleGraph(Integer(3)); G2 = graphs.DiamondGraph()
>>> G = G1.disjoint_union(G2)
>>> M = Matroid(G)
>>> H = M.graph()
>>> vm = M.vertex_map()
>>> (u, v, l) = G.random_edge()
>>> H.has_edge(vm[u], vm[v])
True
```

graph()

Return the graph that represents the matroid.

The graph will always have loops and multiedges enabled.

OUTPUT: graph

EXAMPLES:

```
sage: M = Matroid(Graph([(0, 1, 'a'), (0, 2, 'b'), (0, 3, 'c')]))
sage: M.graph().edges(sort=True)
[(0, 1, 'a'), (0, 2, 'b'), (0, 3, 'c')]
sage: M = Matroid(graphs.CompleteGraph(5))
sage: M.graph()
Looped multi-graph on 5 vertices
```

```
>>> from sage.all import *
>>> M = Matroid(Graph([(Integer(0), Integer(1), 'a'), (Integer(0), Integer(2),
↪ 'b'), (Integer(0), Integer(3), 'c')]))
>>> M.graph().edges(sort=True)
[(0, 1, 'a'), (0, 2, 'b'), (0, 3, 'c')]
>>> M = Matroid(graphs.CompleteGraph(Integer(5)))
>>> M.graph()
Looped multi-graph on 5 vertices
```

graphic_coextension(*u*, *v=None*, *X=None*, *element=None*)

Return a matroid coextended by a new element.

A coextension in a graphic matroid is the opposite of contracting an edge; that is, a vertex is split, and a new edge is added between the resulting vertices. This method will create a new vertex v adjacent to u, and move the edges indicated by X from u to v.

INPUT:

- `u` – the vertex to be split
- `v` – (optional) the name of the new vertex after splitting
- `X` – (optional) a list of the matroid elements corresponding to edges incident to `u` that move to the new vertex after splitting
- `element` – (optional) the name of the newly added element

OUTPUT:

An instance of *GraphicMatroid* coextended by the new element. If `X` is not specified, the new element will be a coloop.

Note

A loop on `u` will stay a loop unless it is in `X`.

EXAMPLES:

```
sage: G = Graph([(0, 1, 0), (0, 2, 1), (0, 3, 2), (0, 4, 3),
....:            (1, 2, 4), (1, 4, 5), (2, 3, 6), (3, 4, 7)])
sage: M = Matroid(G)
sage: M1 = M.graphic_coextension(0, X=[1,2], element='a')
sage: M1.graph().edges(sort=True)
[(0, 1, 0),
 (0, 4, 3),
 (0, 5, 'a'),
 (1, 2, 4),
 (1, 4, 5),
 (2, 3, 6),
 (2, 5, 1),
 (3, 4, 7),
 (3, 5, 2)]
```

```
>>> from sage.all import *
>>> G = Graph([(Integer(0), Integer(1), Integer(0)), (Integer(0), Integer(2),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)), (Integer(0), Integer(3), Integer(2)), (Integer(0), Integer(4),␣
↪Integer(3)),
...             (Integer(1), Integer(2), Integer(4)), (Integer(1), Integer(4),␣
↪Integer(5)), (Integer(2), Integer(3), Integer(6)), (Integer(3), Integer(4),␣
↪Integer(7))])
>>> M = Matroid(G)
>>> M1 = M.graphic_coextension(Integer(0), X=[Integer(1),Integer(2)], element=
↪'a')
>>> M1.graph().edges(sort=True)
[(0, 1, 0),
 (0, 4, 3),
 (0, 5, 'a'),
 (1, 2, 4),
 (1, 4, 5),
 (2, 3, 6),
 (2, 5, 1),
 (3, 4, 7),
 (3, 5, 2)]
```

graphic_coextensions(*vertices=None*, *v=None*, *element=None*, *cosimple=False*)

Return an iterator of graphic coextensions.

This method iterates over the vertices in the input. If `cosimple == False`, it first coextends by a coloop and series edge for every edge incident with the vertices. For vertices of degree four or higher, it will consider the ways to partition the vertex into two sets of cardinality at least two, and these will be the edges incident with the vertices after splitting.

At most one series coextension will be taken for each series class.

INPUT:

- `vertices` – (optional) the vertices to be split
- `v` – (optional) the name of the new vertex
- `element` – (optional) the name of the new element
- `cosimple` – boolean (default: `False`); if `True`, coextensions by a coloop or series elements will not be taken

OUTPUT:

An iterable containing instances of *GraphicMatroid*. If `vertices` is not specified, the method iterates over all vertices.

EXAMPLES:

```
sage: G = Graph([(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 4), (2, 3), (3,␣
↪4)])
sage: M = Matroid(range(8), G)
sage: I = M.graphic_coextensions(vertices=[0], element='a')
sage: sorted([N.graph().edges_incident(0, sort=True) for N in I], key=str)
[[(0, 1, 0), (0, 2, 1), (0, 3, 2), (0, 4, 3), (0, 5, 'a')],
 [(0, 1, 0), (0, 2, 1), (0, 3, 2), (0, 5, 'a')],
 [(0, 1, 0), (0, 2, 1), (0, 4, 3), (0, 5, 'a')],
 [(0, 1, 0), (0, 2, 1), (0, 5, 'a')],
 [(0, 1, 0), (0, 3, 2), (0, 4, 3), (0, 5, 'a')],
```

(continues on next page)

(continued from previous page)

```
 [(0, 1, 0), (0, 3, 2), (0, 5, 'a')],
 [(0, 2, 1), (0, 3, 2), (0, 4, 3), (0, 5, 'a')],
 [(0, 2, 1), (0, 3, 2), (0, 5, 'a')]]
```

```
>>> from sage.all import *
>>> G = Graph([(Integer(0), Integer(1)), (Integer(0), Integer(2)),␣
↪(Integer(0), Integer(3)), (Integer(0), Integer(4)), (Integer(1),␣
↪Integer(2)), (Integer(1), Integer(4)), (Integer(2), Integer(3)),␣
↪(Integer(3), Integer(4))])
>>> M = Matroid(range(Integer(8)), G)
>>> I = M.graphic_coextensions(vertices=[Integer(0)], element='a')
>>> sorted([N.graph().edges_incident(Integer(0), sort=True) for N in I],␣
↪key=str)
[[(0, 1, 0), (0, 2, 1), (0, 3, 2), (0, 4, 3), (0, 5, 'a')],
 [(0, 1, 0), (0, 2, 1), (0, 3, 2), (0, 5, 'a')],
 [(0, 1, 0), (0, 2, 1), (0, 4, 3), (0, 5, 'a')],
 [(0, 1, 0), (0, 2, 1), (0, 5, 'a')],
 [(0, 1, 0), (0, 3, 2), (0, 4, 3), (0, 5, 'a')],
 [(0, 1, 0), (0, 3, 2), (0, 5, 'a')],
 [(0, 2, 1), (0, 3, 2), (0, 4, 3), (0, 5, 'a')],
 [(0, 2, 1), (0, 3, 2), (0, 5, 'a')]]
```

```
sage: N = Matroid(range(4), graphs.CycleGraph(4))
sage: I = N.graphic_coextensions(element='a')
sage: for N1 in I:  # random
....:     N1.graph().edges(sort=True)
[(0, 1, 0), (0, 3, 1), (0, 4, 'a'), (1, 2, 2), (2, 3, 3)]
[(0, 1, 0), (0, 3, 1), (1, 4, 2), (2, 3, 3), (2, 4, 'a')]
sage: sum(1 for n in N.graphic_coextensions(cosimple=True))
0
```

```
>>> from sage.all import *
>>> N = Matroid(range(Integer(4)), graphs.CycleGraph(Integer(4)))
>>> I = N.graphic_coextensions(element='a')
>>> for N1 in I:  # random
...     N1.graph().edges(sort=True)
[(0, 1, 0), (0, 3, 1), (0, 4, 'a'), (1, 2, 2), (2, 3, 3)]
[(0, 1, 0), (0, 3, 1), (1, 4, 2), (2, 3, 3), (2, 4, 'a')]
>>> sum(Integer(1) for n in N.graphic_coextensions(cosimple=True))
0
```

graphic_extension(*u*, *v=None*, *element=None*)

Return a graphic matroid extended by a new element.

A new edge will be added between `u` and `v`. If `v` is not specified, then a loop is added on `u`.

INPUT:

- `u` – vertex in the matroid's graph
- `v` – (optional) another vertex
- `element` – (optional) the label of the new element

OUTPUT:

A *GraphicMatroid* with the specified element added. Note that if `v` is not specified or if `v` is `u`, then the new element will be a loop. If the new element's label is not specified, it will be generated automatically.

EXAMPLES:

```
sage: M = matroids.CompleteGraphic(4)
sage: M1 = M.graphic_extension(0,1,'a'); M1
Graphic matroid of rank 3 on 7 elements
sage: list(M1.graph().edge_iterator())
[(0, 1, 'a'), (0, 1, 0), (0, 2, 1), (0, 3, 2), (1, 2, 3), (1, 3, 4), (2, 3,␣
↪5)]
sage: M2 = M1.graphic_extension(3); M2
Graphic matroid of rank 3 on 8 elements
```

```
>>> from sage.all import *
>>> M = matroids.CompleteGraphic(Integer(4))
>>> M1 = M.graphic_extension(Integer(0),Integer(1),'a'); M1
Graphic matroid of rank 3 on 7 elements
>>> list(M1.graph().edge_iterator())
[(0, 1, 'a'), (0, 1, 0), (0, 2, 1), (0, 3, 2), (1, 2, 3), (1, 3, 4), (2, 3,␣
↪5)]
>>> M2 = M1.graphic_extension(Integer(3)); M2
Graphic matroid of rank 3 on 8 elements
```

```
sage: M = Matroid(range(10), graphs.PetersenGraph())
sage: sorted(M.graphic_extension(0, 'b', 'c').graph().vertex_iterator(),␣
↪key=str)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'b']
sage: M.graphic_extension('a', 'b', 'c').graph().vertices(sort=False)
Traceback (most recent call last):
...
ValueError: u must be an existing vertex
```

```
>>> from sage.all import *
>>> M = Matroid(range(Integer(10)), graphs.PetersenGraph())
>>> sorted(M.graphic_extension(Integer(0), 'b', 'c').graph().vertex_
↪iterator(), key=str)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'b']
>>> M.graphic_extension('a', 'b', 'c').graph().vertices(sort=False)
Traceback (most recent call last):
...
ValueError: u must be an existing vertex
```

graphic_extensions(*element=None*, *vertices=None*, *simple=False*)

Return an iterable containing the graphic extensions.

This method iterates over the vertices in the input. If `simple == False`, it first extends by a loop. It will then add an edge between every pair of vertices in the input, skipping pairs of vertices with an edge already between them if `simple == True`.

This method only considers the current graph presentation, and does not take 2-isomorphism into account. Use *twist* or *one_sum* if you wish to change the graph presentation.

INPUT:

- `element` – (optional) the name of the newly added element in each extension

- `vertices` – (optional) a set of vertices over which the extension may be taken
- `simple` – boolean (default: `False`); if `True`, extensions by loops and parallel elements are not taken

OUTPUT:

An iterable containing instances of *GraphicMatroid*. If `vertices` is not specified, every vertex is used.

Note

The extension by a loop will always occur unless `simple == True`. The extension by a coloop will never occur.

EXAMPLES:

```
sage: M = Matroid(range(5), graphs.DiamondGraph())
sage: I = M.graphic_extensions('a')
sage: for N in I:
....:     list(N.graph().edge_iterator())
[(0, 0, 'a'), (0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 'a'), (0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 'a'), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (0, 3, 'a'), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (1, 2, 'a'), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 'a'), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 'a'), (2, 3, 4)]
```

```
>>> from sage.all import *
>>> M = Matroid(range(Integer(5)), graphs.DiamondGraph())
>>> I = M.graphic_extensions('a')
>>> for N in I:
...     list(N.graph().edge_iterator())
[(0, 0, 'a'), (0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 'a'), (0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 'a'), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (0, 3, 'a'), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (1, 2, 'a'), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 'a'), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 'a'), (2, 3, 4)]
```

```
sage: M = Matroid(graphs.CompleteBipartiteGraph(3,3))
sage: I = M.graphic_extensions(simple=True)
sage: sum (1 for i in I)
6
sage: I = M.graphic_extensions(vertices=[0,1,2])
sage: sum (1 for i in I)
4
```

```
>>> from sage.all import *
>>> M = Matroid(graphs.CompleteBipartiteGraph(Integer(3),Integer(3)))
>>> I = M.graphic_extensions(simple=True)
>>> sum (Integer(1) for i in I)
6
```

(continues on next page)

(continued from previous page)

```
>>> I = M.graphic_extensions(vertices=[Integer(0),Integer(1),Integer(2)])
>>> sum (Integer(1) for i in I)
4
```

groundset()

Return the groundset of the matroid as a frozenset.

EXAMPLES:

```
sage: M = Matroid(graphs.DiamondGraph())
sage: sorted(M.groundset())
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]
sage: G = graphs.CompleteGraph(3).disjoint_union(graphs.CompleteGraph(4))
sage: M = Matroid(range(G.num_edges()), G); sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6, 7, 8]
sage: M = Matroid(Graph([(0, 1, 'a'), (0, 2, 'b'), (0, 3, 'c')]))
sage: sorted(M.groundset())
['a', 'b', 'c']
```

```
>>> from sage.all import *
>>> M = Matroid(graphs.DiamondGraph())
>>> sorted(M.groundset())
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]
>>> G = graphs.CompleteGraph(Integer(3)).disjoint_union(graphs.
↪CompleteGraph(Integer(4)))
>>> M = Matroid(range(G.num_edges()), G); sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> M = Matroid(Graph([(Integer(0), Integer(1), 'a'), (Integer(0), Integer(2),
↪ 'b'), (Integer(0), Integer(3), 'c')]))
>>> sorted(M.groundset())
['a', 'b', 'c']
```

groundset_to_edges(X)

Return a list of edges corresponding to a set of groundset elements.

INPUT:

- `X` – subset of the groundset

OUTPUT: list of graph edges

EXAMPLES:

```
sage: M = Matroid(range(5), graphs.DiamondGraph())
sage: M.groundset_to_edges([2, 3, 4])
[(1, 2, 2), (1, 3, 3), (2, 3, 4)]
sage: M.groundset_to_edges([2, 3, 4, 5])
Traceback (most recent call last):
...
ValueError: input must be a subset of the groundset
```

```
>>> from sage.all import *
>>> M = Matroid(range(Integer(5)), graphs.DiamondGraph())
>>> M.groundset_to_edges([Integer(2), Integer(3), Integer(4)])
```

(continues on next page)

(continued from previous page)

```
[(1, 2, 2), (1, 3, 3), (2, 3, 4)]
>>> M.groundset_to_edges([Integer(2), Integer(3), Integer(4), Integer(5)])
Traceback (most recent call last):
...
ValueError: input must be a subset of the groundset
```

is_graphic()

Return if `self` is graphic.

This is trivially `True` for a *GraphicMatroid*.

EXAMPLES:

```
sage: M = Matroid(graphs.PetersenGraph())
sage: M.is_graphic()
True
```

```
>>> from sage.all import *
>>> M = Matroid(graphs.PetersenGraph())
>>> M.is_graphic()
True
```

is_regular()

Return if `self` is regular.

This is always `True` for a *GraphicMatroid*.

EXAMPLES:

```
sage: M = Matroid(graphs.DesarguesGraph())
sage: M.is_regular()
True
```

```
>>> from sage.all import *
>>> M = Matroid(graphs.DesarguesGraph())
>>> M.is_regular()
True
```

is_valid(*certificate=False*)

Test if the data obey the matroid axioms.

Since a graph is used for the data, this is always the case.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: `True`, or `(True, {})`

EXAMPLES:

```
sage: M = matroids.CompleteGraphic(4); M
M(K4): Graphic matroid of rank 3 on 6 elements
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = matroids.CompleteGraphic(Integer(4)); M
M(K4): Graphic matroid of rank 3 on 6 elements
>>> M.is_valid()
True
```

one_sum(X, u, v)

Arrange matroid components in the graph.

The matroid's graph must be connected even if the matroid is not connected, but if there are multiple matroid components, the user may choose how they are arranged in the graph. This method will take the block of the graph that represents X and attach it by vertex u to another vertex v in the graph.

INPUT:

- `X` – subset of the groundset
- `u` – vertex spanned by the edges of the elements in `X`
- `v` – vertex spanned by the edges of the elements not in `X`

OUTPUT: *GraphicMatroid* isomorphic to this matroid but with a graph that is not necessarily isomorphic

EXAMPLES:

```
sage: edgedict = {0:[1, 2], 1:[2, 3], 2:[3], 3:[4, 5], 6:[4, 5]}
sage: M = Matroid(range(9), Graph(edgedict))
sage: M.graph().edges(sort=True)
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 3, 3),
 (2, 3, 4),
 (3, 4, 5),
 (3, 5, 6),
 (4, 6, 7),
 (5, 6, 8)]
sage: M1 = M.one_sum(u=3, v=1, X=[5, 6, 7, 8])
sage: M1.graph().edges(sort=True)
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 3, 3),
 (1, 4, 5),
 (1, 5, 6),
 (2, 3, 4),
 (4, 6, 7),
 (5, 6, 8)]
sage: M2 = M.one_sum(u=4, v=3, X=[5, 6, 7, 8])
sage: M2.graph().edges(sort=True)
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 3, 3),
 (2, 3, 4),
 (3, 6, 7),
```

(continues on next page)

(continued from previous page)

```
 (3, 7, 5),
 (5, 6, 8),
 (5, 7, 6)]
```

```
>>> from sage.all import *
>>> edgedict = {Integer(0):[Integer(1), Integer(2)], Integer(1):[Integer(2),␣
↪Integer(3)], Integer(2):[Integer(3)], Integer(3):[Integer(4), Integer(5)],␣
↪Integer(6):[Integer(4), Integer(5)]}
>>> M = Matroid(range(Integer(9)), Graph(edgedict))
>>> M.graph().edges(sort=True)
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 3, 3),
 (2, 3, 4),
 (3, 4, 5),
 (3, 5, 6),
 (4, 6, 7),
 (5, 6, 8)]
>>> M1 = M.one_sum(u=Integer(3), v=Integer(1), X=[Integer(5), Integer(6),␣
↪Integer(7), Integer(8)])
>>> M1.graph().edges(sort=True)
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 3, 3),
 (1, 4, 5),
 (1, 5, 6),
 (2, 3, 4),
 (4, 6, 7),
 (5, 6, 8)]
>>> M2 = M.one_sum(u=Integer(4), v=Integer(3), X=[Integer(5), Integer(6),␣
↪Integer(7), Integer(8)])
>>> M2.graph().edges(sort=True)
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 3, 3),
 (2, 3, 4),
 (3, 6, 7),
 (3, 7, 5),
 (5, 6, 8),
 (5, 7, 6)]
```

regular_matroid()

Return an instance of `RegularMatroid` isomorphic to this *GraphicMatroid*.

EXAMPLES:

```
sage: M = matroids.CompleteGraphic(5); M
M(K5): Graphic matroid of rank 4 on 10 elements
sage: N = M.regular_matroid(); N
```

(continues on next page)

(continued from previous page)

```
Regular matroid of rank 4 on 10 elements with 125 bases
sage: M.equals(N)
True
sage: M == N
False
```

```
>>> from sage.all import *
>>> M = matroids.CompleteGraphic(Integer(5)); M
M(K5): Graphic matroid of rank 4 on 10 elements
>>> N = M.regular_matroid(); N
Regular matroid of rank 4 on 10 elements with 125 bases
>>> M.equals(N)
True
>>> M == N
False
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: M = matroids.CompleteGraphic(4)
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
sage: N = M.relabel({0: 6, 5: 'e'})
sage: sorted(N.groundset(), key=str)
[1, 2, 3, 4, 6, 'e']
sage: N.is_isomorphic(M)
True
```

```
>>> from sage.all import *
>>> M = matroids.CompleteGraphic(Integer(4))
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
>>> N = M.relabel({Integer(0): Integer(6), Integer(5): 'e'})
>>> sorted(N.groundset(), key=str)
[1, 2, 3, 4, 6, 'e']
>>> N.is_isomorphic(M)
True
```

subgraph_from_set(X)

Return the subgraph corresponding to the matroid restricted to X.

INPUT:

- `X` – subset of the groundset

OUTPUT: graph

EXAMPLES:

```
sage: M = Matroid(range(5), graphs.DiamondGraph())
sage: M.subgraph_from_set([0,1,2])
Looped multi-graph on 3 vertices
sage: M.subgraph_from_set([3,4,5])
Traceback (most recent call last):
...
ValueError: input must be a subset of the groundset
```

```
>>> from sage.all import *
>>> M = Matroid(range(Integer(5)), graphs.DiamondGraph())
>>> M.subgraph_from_set([Integer(0),Integer(1),Integer(2)])
Looped multi-graph on 3 vertices
>>> M.subgraph_from_set([Integer(3),Integer(4),Integer(5)])
Traceback (most recent call last):
...
ValueError: input must be a subset of the groundset
```

twist(X)

Perform a Whitney twist on the graph.

X must be part of a 2-separation. The connectivity of X must be 1, and the subgraph induced by X must intersect the subgraph induced by the rest of the elements on exactly two vertices.

INPUT:

- X – the set of elements to be twisted with respect to the rest of the matroid

OUTPUT: *GraphicMatroid* isomorphic to this matroid but with a graph that is not necessarily isomorphic

EXAMPLES:

```
sage: edgelist = [(0, 1, 0), (1, 2, 1), (1, 2, 2), (2, 3, 3),
....:             (2, 3, 4), (2, 3, 5), (3, 0, 6)]
sage: M = Matroid(Graph(edgelist, multiedges=True))
sage: M1 = M.twist([0, 1, 2]); M1.graph().edges(sort=True)
[(0, 1, 1), (0, 1, 2), (0, 3, 6), (1, 2, 0), (2, 3, 3), (2, 3, 4), (2, 3, 5)]
sage: M2 = M.twist([0, 1, 3])
Traceback (most recent call last):
...
ValueError: the input must display a 2-separation that is not a 1-separation
```

```
>>> from sage.all import *
>>> edgelist = [(Integer(0), Integer(1), Integer(0)), (Integer(1), Integer(2),
↪ Integer(1)), (Integer(1), Integer(2), Integer(2)), (Integer(2), Integer(3),
↪ Integer(3)),
...             (Integer(2), Integer(3), Integer(4)), (Integer(2), Integer(3),
↪ Integer(5)), (Integer(3), Integer(0), Integer(6))]
>>> M = Matroid(Graph(edgelist, multiedges=True))
>>> M1 = M.twist([Integer(0), Integer(1), Integer(2)]); M1.graph().
↪edges(sort=True)
[(0, 1, 1), (0, 1, 2), (0, 3, 6), (1, 2, 0), (2, 3, 3), (2, 3, 4), (2, 3, 5)]
>>> M2 = M.twist([Integer(0), Integer(1), Integer(3)])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: the input must display a 2-separation that is not a 1-separation
```

vertex_map()

Return a dictionary mapping the input vertices to the current vertices.

The graph for the matroid is always connected. If the constructor is given a graph with multiple components, it will connect them. The Python dictionary given by this method has the vertices from the input graph as keys, and the corresponding vertex label after any merging as values.

OUTPUT: dictionary

EXAMPLES:

```
sage: G = Graph([(0, 1), (0, 2), (1, 2), (3, 4), (3, 5), (4, 5),
....: (6, 7), (6, 8), (7, 8), (8, 8), (7, 8)], multiedges=True, loops=True)
sage: M = Matroid(range(G.num_edges()), G)
sage: M.graph().edges(sort=True)
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 4, 3),
 (1, 5, 4),
 (4, 5, 5),
 (4, 7, 6),
 (4, 8, 7),
 (7, 8, 8),
 (7, 8, 9),
 (8, 8, 10)]
sage: M.vertex_map()
{0: 0, 1: 1, 2: 2, 3: 1, 4: 4, 5: 5, 6: 4, 7: 7, 8: 8}
```

```
>>> from sage.all import *
>>> G = Graph([(Integer(0), Integer(1)), (Integer(0), Integer(2)),␣
↪(Integer(1), Integer(2)), (Integer(3), Integer(4)), (Integer(3),␣
↪Integer(5)), (Integer(4), Integer(5)),
... (Integer(6), Integer(7)), (Integer(6), Integer(8)), (Integer(7),␣
↪Integer(8)), (Integer(8), Integer(8)), (Integer(7), Integer(8))],␣
↪multiedges=True, loops=True)
>>> M = Matroid(range(G.num_edges()), G)
>>> M.graph().edges(sort=True)
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 4, 3),
 (1, 5, 4),
 (4, 5, 5),
 (4, 7, 6),
 (4, 8, 7),
 (7, 8, 8),
 (7, 8, 9),
 (8, 8, 10)]
```

(continues on next page)

(continued from previous page)

```
>>> M.vertex_map()
{0: 0, 1: 1, 2: 2, 3: 1, 4: 4, 5: 5, 6: 4, 7: 7, 8: 8}
```

3.7 Linear matroids

When A is an r times E matrix, the linear matroid $M[A]$ has groundset E and, for independent sets, all F subset of E such that the columns of $M[A]$ indexed by F are linearly independent.

3.7.1 Construction

The recommended way to create a linear matroid is by using the *Matroid()* function, with a representation matrix A as input. This function will intelligently choose one of the dedicated classes *BinaryMatroid*, *TernaryMatroid*, *QuaternaryMatroid*, *RegularMatroid* when appropriate. However, invoking the classes directly is possible too. To get access to them, type:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See also *sage.matroids.advanced*. In both cases, it is possible to provide a reduced matrix B, to create the matroid induced by $A = [IB]$:

```
sage: from sage.matroids.advanced import *
sage: A = Matrix(GF(2), [[1, 0, 0, 1, 1, 0, 1], [0, 1, 0, 1, 0, 1, 1],
....:                    [0, 0, 1, 0, 1, 1, 1]])
sage: B = Matrix(GF(2), [[1, 1, 0, 1], [1, 0, 1, 1], [0, 1, 1, 1]])
sage: M1 = Matroid(A)
sage: M2 = LinearMatroid(A)
sage: M3 = BinaryMatroid(A)
sage: M4 = Matroid(reduced_matrix=B)
sage: M5 = LinearMatroid(reduced_matrix=B)
sage: isinstance(M1, BinaryMatroid)
True
sage: M1.equals(M2)
True
sage: M1.equals(M3)
True
sage: M1 == M4
True
sage: M1.is_field_isomorphic(M5)
True
sage: M2 == M3  # comparing LinearMatroid and BinaryMatroid always yields False
False
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> A = Matrix(GF(Integer(2)), [[Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1)],
...                    [Integer(0), Integer(0), Integer(1), Integer(0), Integer(1),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1), Integer(1)]])
>>> B = Matrix(GF(Integer(2)), [[Integer(1), Integer(1), Integer(0), Integer(1)],␣
↪[Integer(1), Integer(0), Integer(1), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(1), Integer(1)]])
>>> M1 = Matroid(A)
>>> M2 = LinearMatroid(A)
>>> M3 = BinaryMatroid(A)
>>> M4 = Matroid(reduced_matrix=B)
>>> M5 = LinearMatroid(reduced_matrix=B)
>>> isinstance(M1, BinaryMatroid)
True
>>> M1.equals(M2)
True
>>> M1.equals(M3)
True
>>> M1 == M4
True
>>> M1.is_field_isomorphic(M5)
True
>>> M2 == M3  # comparing LinearMatroid and BinaryMatroid always yields False
False
```

3.7.2 Class methods

The `LinearMatroid` class and its derivatives inherit all methods from the *Matroid* and *BasisExchangeMatroid* classes. See the documentation for these classes for an overview. In addition, the following methods are available:

- *LinearMatroid*
  - *base_ring()*
  - *characteristic()*
  - *representation()*
  - *representation_vectors()*
  - *is_field_equivalent()*
  - *is_field_isomorphism()*
  - *has_field_minor()*
  - *fundamental_cycle()*
  - *fundamental_cocycle()*
  - *cross_ratios()*
  - *cross_ratio()*
  - *linear_extension()*
  - *linear_coextension()*
  - *linear_extension_chains()*
  - *linear_coextension_cochains()*
  - *linear_extensions()*

  - *linear_coextensions()*
  - *orlik_terao_algebra()*
- *BinaryMatroid* has all of the *LinearMatroid* ones, and
  - *bicycle_dimension()*
  - *brown_invariant()*
  - *is_graphic()*
- *TernaryMatroid* has all of the *LinearMatroid* ones, and
  - *bicycle_dimension()*
  - *character()*
- *QuaternaryMatroid* has all of the *LinearMatroid* ones, and
  - *bicycle_dimension()*
- *RegularMatroid* has all of the *LinearMatroid* ones, and
  - *is_graphic()*

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

3.7.3 Methods

class sage.matroids.linear_matroid.**BinaryMatroid**

Bases: *LinearMatroid*

Binary matroids.

A binary matroid is a linear matroid represented over the finite field with two elements. See *LinearMatroid* for a definition.

The simplest way to create a `BinaryMatroid` is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object E can be given as a groundset. If E is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a *reduced* matrix B, to create the matroid induced by $A = [IB]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \quad B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.
- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) ignored
- `keep_initial_representation` – boolean (default: `True`); whether or not an internal copy of the input matrix should be preserved. This can help to see the structure of the matroid (e.g. in the case of graphic matroids), and makes it easier to look at extensions. However, the input matrix may have redundant rows, and sometimes it is desirable to store only a row-reduced copy.
- `basis` – (default: `None`) when provided, this is an ordered subset of `groundset`, such that the submatrix of `matrix` indexed by `basis` is an identity matrix. In this case, no row reduction takes place in the initialization phase.

OUTPUT: a *BinaryMatroid* instance based on the data above

Note

An indirect way to generate a binary matroid is through the *Matroid()* function. This is usually the preferred way, since it automatically chooses between *BinaryMatroid* and other classes. For direct access to the BinaryMatroid constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

EXAMPLES:

```
sage: A = Matrix(GF(2), 2, 4, [[1, 0, 1, 1], [0, 1, 1, 1]])
sage: M = Matroid(A)
sage: M
Binary matroid of rank 2 on 4 elements, type (0, 6)
sage: sorted(M.groundset())
[0, 1, 2, 3]
sage: Matrix(M)
[1 0 1 1]
[0 1 1 1]
sage: M = Matroid(matrix=A, groundset='abcd')
sage: sorted(M.groundset())
['a', 'b', 'c', 'd']
sage: B = Matrix(GF(2), 2, 2, [[1, 1], [1, 1]])
sage: N = Matroid(reduced_matrix=B, groundset='abcd')
sage: M == N
True
```

```
>>> from sage.all import *
>>> A = Matrix(GF(Integer(2)), Integer(2), Integer(4), [[Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(1)]])
>>> M = Matroid(A)
>>> M
Binary matroid of rank 2 on 4 elements, type (0, 6)
>>> sorted(M.groundset())
[0, 1, 2, 3]
>>> Matrix(M)
[1 0 1 1]
[0 1 1 1]
>>> M = Matroid(matrix=A, groundset='abcd')
>>> sorted(M.groundset())
['a', 'b', 'c', 'd']
>>> B = Matrix(GF(Integer(2)), Integer(2), Integer(2), [[Integer(1), Integer(1)],␣
↪[Integer(1), Integer(1)]])
>>> N = Matroid(reduced_matrix=B, groundset='abcd')
>>> M == N
True
```

base_ring()

Return the base ring of the matrix representing the matroid, in this case $\mathbf{F}_2$.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.base_ring()
Finite Field of size 2
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.base_ring()
Finite Field of size 2
```

bicycle_dimension()

Return the bicycle dimension of the binary matroid.

The *bicycle dimension* of a linear subspace V is $\dim(V \cap V^{\perp})$. The bicycle dimension of a matroid equals the bicycle dimension of its cocycle-space, and is an invariant for binary matroids. See [Pen2012], [GR2001] for more information.

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.bicycle_dimension()
3
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.bicycle_dimension()
3
```

binary_matroid(*randomized_tests=1*, *verify=True*)

Return a binary matroid representing `self`.

INPUT:

- `randomized_tests` – ignored
- `verify` – ignored

OUTPUT: a binary matroid

ALGORITHM:

`self` is a binary matroid, so just return `self`.

See also

M.binary_matroid()

EXAMPLES:

```
sage: N = matroids.catalog.Fano()
sage: N.binary_matroid() is N
True
```

```
>>> from sage.all import *
>>> N = matroids.catalog.Fano()
>>> N.binary_matroid() is N
True
```

brown_invariant()

Return the value of Brown's invariant for the binary matroid.

For a binary space V, consider the sum $B(V) := \sum_{v \in V} i^{|v|}$, where $|v|$ denotes the number of nonzero entries of a binary vector v. The value of the Tutte Polynomial in the point $(-i, i)$ can be expressed in terms of $B(V)$, see [Pen2012]. If $|v|$ equals 2 modulo 4 for some $v \in V \cap V^{\perp}$, then $B(V) = 0$. In this case, Browns invariant is not defined. Otherwise, $B(V) = \sqrt{2}^k \exp(\sigma\pi i/4)$ for some integers k, σ. In that case, k equals the bycycle dimension of V, and Browns invariant for V is defined as σ modulo 8.

The Brown invariant of a binary matroid equals the Brown invariant of its cocycle-space.

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.brown_invariant()
0
sage: M = Matroid(Matrix(GF(2), 3, 8, [[1, 0, 0, 1, 1, 1, 1, 1],
....:                                  [0, 1, 0, 1, 1, 0, 0, 0],
....:                                  [0, 0, 1, 0, 0, 1, 1, 0]]))
sage: M.brown_invariant() is None
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.brown_invariant()
0
>>> M = Matroid(Matrix(GF(Integer(2)), Integer(3), Integer(8), [[Integer(1),␣
↪Integer(0), Integer(0), Integer(1), Integer(1), Integer(1), Integer(1),␣
↪Integer(1)],
...                                  [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)],
...                                  [Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1), Integer(1), Integer(0)]]))
>>> M.brown_invariant() is None
True
```

characteristic()

Return the characteristic of the base ring of the matrix representing the matroid, in this case 2.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.characteristic()
2
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
```

(continues on next page)

(continued from previous page)

```
>>> M.characteristic()
2
```

is_binary(*randomized_tests=1*)

Decide if `self` is a binary matroid.

INPUT:

- `randomized_tests` – ignored

OUTPUT: boolean

ALGORITHM:

`self` is a binary matroid, so just return `True`.

See also

M.is_binary()

EXAMPLES:

```
sage: N = matroids.catalog.Fano()
sage: N.is_binary()
True
```

```
>>> from sage.all import *
>>> N = matroids.catalog.Fano()
>>> N.is_binary()
True
```

is_graphic()

Test if the binary matroid is graphic.

A matroid is *graphic* if there exists a graph whose edge set equals the groundset of the matroid, such that a subset of elements of the matroid is independent if and only if the corresponding subgraph is acyclic.

OUTPUT: boolean

EXAMPLES:

```
sage: R10 = matroids.catalog.R10()
sage: M = Matroid(ring=GF(2), reduced_matrix=R10.representation(
....:                                 reduced=True, labels=False))
sage: M.is_graphic()
False
sage: K5 = Matroid(graphs.CompleteGraph(5), regular=True)                   #␣
↪needs sage.graphs
sage: M = Matroid(ring=GF(2), reduced_matrix=K5.representation(             #␣
↪needs sage.graphs sage.rings.finite_rings
....:                                 reduced=True, labels=False))
sage: M.is_graphic()                                                        #␣
↪needs sage.graphs sage.rings.finite_rings
True
sage: M.dual().is_graphic()                                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
False
```

```
>>> from sage.all import *
>>> R10 = matroids.catalog.R10()
>>> M = Matroid(ring=GF(Integer(2)), reduced_matrix=R10.representation(
...                                  reduced=True, labels=False))
>>> M.is_graphic()
False
>>> K5 = Matroid(graphs.CompleteGraph(Integer(5)), regular=True)
↪     # needs sage.graphs
>>> M = Matroid(ring=GF(Integer(2)), reduced_matrix=K5.representation(
↪     # needs sage.graphs sage.rings.finite_rings
...                                  reduced=True, labels=False))
>>> M.is_graphic()                                                    #␣
↪needs sage.graphs sage.rings.finite_rings
True
>>> M.dual().is_graphic()                                             #␣
↪needs sage.graphs
False
```

ALGORITHM:

In a recent paper, Geelen and Gerards [GG2012] reduced the problem to testing if a system of linear equations has a solution. While not the fastest method, and not necessarily constructive (in the presence of 2-separations especially), it is easy to implement.

is_valid(*certificate=False*)

Test if the data obey the matroid axioms.

Since this is a linear matroid over the field $\mathbf{F}_2$, this is always the case.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: `True`, or `(True, {})`

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(2), [[]]))
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(2)), [[]]))
>>> M.is_valid()
True
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
sage: N = M.relabel({'g': 'x'})
sage: sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'x']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> N = M.relabel({'g': 'x'})
>>> sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'x']
```

class sage.matroids.linear_matroid.**LinearMatroid**

Bases: *BasisExchangeMatroid*

Linear matroids.

When A is an r times E matrix, the linear matroid $M[A]$ has ground set E and set of independent sets

$$I(A) = \{F \subseteq E : \text{the columns of } A \text{ indexed by } F \text{ are linearly independent } \}$$

The simplest way to create a LinearMatroid is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object `E` can be given as a groundset. If `E` is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a *reduced* matrix B, to create the matroid induced by $A = [IB]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \quad B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.
- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) the desired base ring of the matrix. If the base ring is different, an attempt will be made to create a new matrix with the correct base ring.
- `keep_initial_representation` – boolean (default: `True`); whether or not an internal copy of the input matrix should be preserved. This can help to see the structure of the matroid (e.g. in the case of graphic matroids), and makes it easier to look at extensions. However, the input matrix may have redundant rows, and sometimes it is desirable to store only a row-reduced copy.

OUTPUT: a `LinearMatroid` instance based on the data above

Note

The recommended way to generate a linear matroid is through the *Matroid()* function. It will automatically choose more optimized classes when present (currently *BinaryMatroid*, *TernaryMatroid*, *QuaternaryMatroid*, *RegularMatroid*). For direct access to the `LinearMatroid` constructor, run:

```
sage: from sage.matroids.advanced import *
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: A = Matrix(GF(3), 2, 4, [[1, 0, 1, 1], [0, 1, 1, 2]])
sage: M = LinearMatroid(A)
sage: M
Linear matroid of rank 2 on 4 elements represented over the Finite
Field of size 3
sage: sorted(M.groundset())
[0, 1, 2, 3]
sage: Matrix(M)
[1 0 1 1]
[0 1 1 2]
sage: M = LinearMatroid(A, 'abcd')
sage: sorted(M.groundset())
['a', 'b', 'c', 'd']
sage: B = Matrix(GF(3), 2, 2, [[1, 1], [1, 2]])
sage: N = LinearMatroid(reduced_matrix=B, groundset='abcd')
sage: M == N
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> A = Matrix(GF(Integer(3)), Integer(2), Integer(4), [[Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2)]])
>>> M = LinearMatroid(A)
>>> M
Linear matroid of rank 2 on 4 elements represented over the Finite
Field of size 3
>>> sorted(M.groundset())
[0, 1, 2, 3]
>>> Matrix(M)
[1 0 1 1]
[0 1 1 2]
>>> M = LinearMatroid(A, 'abcd')
>>> sorted(M.groundset())
['a', 'b', 'c', 'd']
>>> B = Matrix(GF(Integer(3)), Integer(2), Integer(2), [[Integer(1), Integer(1)],␣
↪[Integer(1), Integer(2)]])
>>> N = LinearMatroid(reduced_matrix=B, groundset='abcd')
>>> M == N
True
```

base_ring()

Return the base ring of the matrix representing the matroid.

EXAMPLES:

```
sage: M = Matroid(matrix=Matrix(GF(5), [[1, 0, 1, 1, 1],
....:                                   [0, 1, 1, 2, 3]]))
sage: M.base_ring()
Finite Field of size 5
```

```
>>> from sage.all import *
>>> M = Matroid(matrix=Matrix(GF(Integer(5)), [[Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)],
...                                   [Integer(0), Integer(1), Integer(1),␣
↪Integer(2), Integer(3)]]))
>>> M.base_ring()
Finite Field of size 5
```

characteristic()

Return the characteristic of the base ring of the matrix representing the matroid.

EXAMPLES:

```
sage: M = Matroid(matrix=Matrix(GF(5), [[1, 0, 1, 1, 1],
....:                                   [0, 1, 1, 2, 3]]))
sage: M.characteristic()
5
```

```
>>> from sage.all import *
>>> M = Matroid(matrix=Matrix(GF(Integer(5)), [[Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)],
...                                   [Integer(0), Integer(1), Integer(1),␣
↪Integer(2), Integer(3)]]))
>>> M.characteristic()
5
```

cross_ratio(F, a, b, c, d)

Return the cross ratio of the four ordered points `a, b, c, d` after contracting a flat `F` of codimension 2.

Consider the following matrix with columns labeled by $\{a, b, c, d\}$.

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & x & 1 \end{bmatrix}$$

The cross ratio of the ordered tuple (a, b, c, d) equals x. This method looks at such minors where `F` is a flat to be contracted, and all remaining elements other than `a, b, c, d` are deleted.

INPUT:

- `F` – a flat of codimension 2
- `a`, `b`, `c`, `d` – elements of the groundset

OUTPUT:

The cross ratio of the four points on the line obtained by contracting `F`.

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 0, 1, 1, 1],
....:                            [0, 1, 0, 1, 2, 4],
....:                            [0, 0, 1, 3, 2, 6]]))
sage: M.cross_ratio([0], 1, 2, 3, 5)
4

sage: M = Matroid(ring=GF(7), matrix=[[1, 0, 1, 1], [0, 1, 1, 1]])
sage: M.cross_ratio(set(), 0, 1, 2, 3)
Traceback (most recent call last):
...
ValueError: points a, b, c, d do not form a 4-point line in M/F
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(7)), [[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)],
...                            [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(4)],
...                            [Integer(0), Integer(0), Integer(1),␣
↪Integer(3), Integer(2), Integer(6)]]))
>>> M.cross_ratio([Integer(0)], Integer(1), Integer(2), Integer(3),␣
↪Integer(5))
4

>>> M = Matroid(ring=GF(Integer(7)), matrix=[[Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(1)]])
>>> M.cross_ratio(set(), Integer(0), Integer(1), Integer(2), Integer(3))
Traceback (most recent call last):
...
ValueError: points a, b, c, d do not form a 4-point line in M/F
```

cross_ratios(*hyperlines=None*)

Return the set of cross ratios that occur in this linear matroid.

Consider the following matrix with columns labeled by $\{a, b, c, d\}$.

$$\begin{matrix} 1 & 0 & 1 & 1 \\ 0 & 1 & x & 1 \end{matrix}$$

The cross ratio of the ordered tuple (a, b, c, d) equals x. The set of all cross ratios of a matroid is the set of cross ratios of all such minors.

INPUT:

- `hyperlines` – (optional) a set of flats of the matroid, of rank $r - 2$, where r is the rank of the matroid. If not given, then `hyperlines` defaults to all such flats.

OUTPUT:

A list of all cross ratios of this linearly represented matroid that occur in rank-2 minors that arise by contracting a flat `F` in `hyperlines` (so by default, those are all cross ratios).

See also

M.cross_ratio()

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 0, 1, 1, 1],
....:                            [0, 1, 0, 1, 2, 4],
....:                            [0, 0, 1, 3, 2, 5]]))
sage: sorted(M.cross_ratios())
[2, 3, 4, 5, 6]
sage: M = Matroid(graphs.CompleteGraph(5), regular=True)                    #␣
↪needs sage.graphs
sage: M.cross_ratios()                                                      #␣
↪needs sage.graphs
set()
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(7)), [[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)],
...                            [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(4)],
...                            [Integer(0), Integer(0), Integer(1),␣
↪Integer(3), Integer(2), Integer(5)]]))
>>> sorted(M.cross_ratios())
[2, 3, 4, 5, 6]
>>> M = Matroid(graphs.CompleteGraph(Integer(5)), regular=True)              ␣
↪     # needs sage.graphs
>>> M.cross_ratios()                                                       #␣
↪needs sage.graphs
set()
```

dual()

Return the dual of the matroid.

Let M be a matroid with groundset E. If B is the set of bases of M, then the set $\{E - b : b \in B\}$ is the set of bases of another matroid, the *dual* of M.

If the matroid is represented by $[I_1 \quad A]$, then the dual is represented by $[-A^T \quad I_2]$ for appropriately sized identity matrices I_1, I_2.

OUTPUT: the dual matroid

EXAMPLES:

```
sage: A = Matrix(GF(7), [[1, 1, 0, 1],
....:                    [1, 0, 1, 1],
....:                    [0, 1, 1, 1]])
sage: B = - A.transpose()
sage: Matroid(reduced_matrix=A).dual() == Matroid(
....:                               reduced_matrix=B,
....:                               groundset=[3, 4, 5, 6, 0, 1, 2])
True
```

```
>>> from sage.all import *
>>> A = Matrix(GF(Integer(7)), [[Integer(1), Integer(1), Integer(0),␣
↪Integer(1)],
...                     [Integer(1), Integer(0), Integer(1), Integer(1)],
...                     [Integer(0), Integer(1), Integer(1), Integer(1)]])
>>> B = - A.transpose()
```

(continues on next page)

(continued from previous page)

```
>>> Matroid(reduced_matrix=A).dual() == Matroid(
...                               reduced_matrix=B,
...                               groundset=[Integer(3), Integer(4), Integer(5),
↪ Integer(6), Integer(0), Integer(1), Integer(2)])
True
```

fundamental_cocycle(B, e)

Return the fundamental cycle, relative to `B`, containing element `e`.

This is the *fundamental cocircuit* together with an appropriate signing from the field, such that $Av = 0$, where A is a representation matrix of the dual, and v the vector corresponding to the output.

INPUT:

- `B` – a basis of the matroid
- `e` – an element of the basis

OUTPUT:

A dictionary mapping elements of `M.fundamental_cocircuit(B, e)` to elements of the ring.

See also

M.fundamental_cocircuit()

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 1, 1], [0, 1, 1, 4]]))
sage: v = M.fundamental_cocycle([0, 1], 0)
sage: [v[0], v[2], v[3]]
[1, 1, 1]
sage: frozenset(v.keys()) == M.fundamental_cocircuit([0, 1], 0)
True
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(7)), [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(4)]]))
>>> v = M.fundamental_cocycle([Integer(0), Integer(1)], Integer(0))
>>> [v[Integer(0)], v[Integer(2)], v[Integer(3)]]
[1, 1, 1]
>>> frozenset(v.keys()) == M.fundamental_cocircuit([Integer(0), Integer(1)],␣
↪Integer(0))
True
```

fundamental_cycle(B, e)

Return the fundamental cycle, relative to `B`, containing element `e`.

This is the *fundamental circuit* together with an appropriate signing from the field, such that $Av = 0$, where A is the representation matrix, and v the vector corresponding to the output.

INPUT:

- `B` – a basis of the matroid
- `e` – an element outside the basis

OUTPUT:

A dictionary mapping elements of `M.fundamental_circuit(B, e)` to elements of the ring.

See also

M.fundamental_circuit()

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 1, 1], [0, 1, 1, 4]]))
sage: v = M.fundamental_cycle([0, 1], 3)
sage: [v[0], v[1], v[3]]
[6, 3, 1]
sage: frozenset(v.keys()) == M.fundamental_circuit([0, 1], 3)
True
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(7)), [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(4)]]))
>>> v = M.fundamental_cycle([Integer(0), Integer(1)], Integer(3))
>>> [v[Integer(0)], v[Integer(1)], v[Integer(3)]]
[6, 3, 1]
>>> frozenset(v.keys()) == M.fundamental_circuit([Integer(0), Integer(1)],␣
↪Integer(3))
True
```

has_field_minor(*N*)

Check if `self` has a minor field isomorphic to `N`.

INPUT:

- `N` – matroid

OUTPUT: boolean

See also

M.minor(), *M.is_field_isomorphic()*

Todo

This important method can (and should) be optimized considerably. See [Hli2006] p.1219 for hints to that end.

EXAMPLES:

```
sage: M = matroids.Whirl(3)
sage: matroids.catalog.Fano().has_field_minor(M)
False
sage: matroids.catalog.NonFano().has_field_minor(M)
True
```

```
>>> from sage.all import *
>>> M = matroids.Whirl(Integer(3))
>>> matroids.catalog.Fano().has_field_minor(M)
False
>>> matroids.catalog.NonFano().has_field_minor(M)
True
```

has_line_minor(*k*, *hyperlines=None*, *certificate=False*)

Test if the matroid has a $U_{2,k}$-minor.

The matroid $U_{2,k}$ is a matroid on k elements in which every subset of at most 2 elements is independent, and every subset of more than two elements is dependent.

The optional argument `hyperlines` restricts the search space: this method returns `True` if $si(M/F)$ is isomorphic to $U_{2,l}$ with $l \geq k$ for some F in `hyperlines`, and `False` otherwise.

INPUT:

- `k` – the length of the line minor
- `hyperlines` – (default: `None`) a set of flats of codimension 2. Defaults to the set of all flats of codimension 2.
- `certificate` – (default: `False`) if `True` returns `True, F`, where `F` is a flat and `self.minor(contractions=F)` has a $U_{2,k}$ restriction or `False, None`

OUTPUT: boolean or tuple

EXAMPLES:

```
sage: M = matroids.catalog.N1()
sage: M.has_line_minor(4)
True
sage: M.has_line_minor(5)
False
sage: M.has_line_minor(k=4, hyperlines=[['a', 'b', 'c']])
False
sage: M.has_line_minor(k=4, hyperlines=[['a', 'b', 'c'],
....:                                  ['a', 'b', 'd' ]])
True
sage: M.has_line_minor(4, certificate=True)
(True, frozenset({'a', 'b', 'd'}))
sage: M.has_line_minor(5, certificate=True)
(False, None)
sage: M.has_line_minor(k=4, hyperlines=[['a', 'b', 'c'],
....:                                  ['a', 'b', 'd' ]], certificate=True)
(True, frozenset({'a', 'b', 'd'}))
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N1()
>>> M.has_line_minor(Integer(4))
True
>>> M.has_line_minor(Integer(5))
False
>>> M.has_line_minor(k=Integer(4), hyperlines=[['a', 'b', 'c']])
False
```

(continues on next page)

(continued from previous page)

```
>>> M.has_line_minor(k=Integer(4), hyperlines=[['a', 'b', 'c'],
...                                            ['a', 'b', 'd' ]])
True
>>> M.has_line_minor(Integer(4), certificate=True)
(True, frozenset({'a', 'b', 'd'}))
>>> M.has_line_minor(Integer(5), certificate=True)
(False, None)
>>> M.has_line_minor(k=Integer(4), hyperlines=[['a', 'b', 'c'],
...                                            ['a', 'b', 'd' ]], certificate=True)
(True, frozenset({'a', 'b', 'd'}))
```

is_field_equivalent(*other*)

Test for matroid representation equality.

Two linear matroids M and N with representation matrices A and B are *field equivalent* if they have the same groundset, and the identity map between the groundsets is an isomorphism between the representations A and B. That is, one can be turned into the other using only row operations and column scaling.

INPUT:

- `other` – matroid

OUTPUT: boolean

See also

M.equals(), *M.is_field_isomorphism()*, *M.is_field_isomorphic()*

EXAMPLES:

A *BinaryMatroid* and *LinearMatroid* use different representations of the matroid internally, so `` == `` yields `False`, even if the matroids are equal:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.catalog.Fano()
sage: M1 = LinearMatroid(Matrix(M), groundset=M.groundset_list())
sage: M2 = Matroid(groundset='abcdefg',
....:              reduced_matrix=[[0, 1, 1, 1],
....:                              [1, 0, 1, 1],
....:                              [1, 1, 0, 1]], field=GF(2))
sage: M.equals(M1)
True
sage: M.equals(M2)
True
sage: M.is_field_equivalent(M1)
True
sage: M.is_field_equivalent(M2)
True
sage: M == M1
False
sage: M == M2
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.Fano()
>>> M1 = LinearMatroid(Matrix(M), groundset=M.groundset_list())
>>> M2 = Matroid(groundset='abcdefg',
...              reduced_matrix=[[Integer(0), Integer(1), Integer(1),␣
↪Integer(1)],
...                              [Integer(1), Integer(0), Integer(1),␣
↪Integer(1)],
...                              [Integer(1), Integer(1), Integer(0),␣
↪Integer(1)]], field=GF(Integer(2)))
>>> M.equals(M1)
True
>>> M.equals(M2)
True
>>> M.is_field_equivalent(M1)
True
>>> M.is_field_equivalent(M2)
True
>>> M == M1
False
>>> M == M2
True
```

`LinearMatroid` instances `M` and `N` satisfy `M == N` if the representations are equivalent up to row operations and column scaling:

```
sage: M1 = Matroid(groundset='abcd',
....:              matrix=Matrix(GF(7), [[1, 0, 1, 1], [0, 1, 1, 2]]))
sage: M2 = Matroid(groundset='abcd',
....:              matrix=Matrix(GF(7), [[1, 0, 1, 1], [0, 1, 1, 3]]))
sage: M3 = Matroid(groundset='abcd',
....:              matrix=Matrix(GF(7), [[2, 6, 1, 0], [6, 1, 0, 1]]))
sage: M1.equals(M2)
True
sage: M1.equals(M3)
True
sage: M1 == M2
False
sage: M1 == M3
True
sage: M1.is_field_equivalent(M2)
False
sage: M1.is_field_equivalent(M3)
True
sage: M1.is_field_equivalent(M1)
True
```

```
>>> from sage.all import *
>>> M1 = Matroid(groundset='abcd',
...              matrix=Matrix(GF(Integer(7)), [[Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2)]]))
>>> M2 = Matroid(groundset='abcd',
```

(continues on next page)

(continued from previous page)

```
...           matrix=Matrix(GF(Integer(7)), [[Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(3)]]))
>>> M3 = Matroid(groundset='abcd',
...           matrix=Matrix(GF(Integer(7)), [[Integer(2), Integer(6),␣
↪Integer(1), Integer(0)], [Integer(6), Integer(1), Integer(0), Integer(1)]]))
>>> M1.equals(M2)
True
>>> M1.equals(M3)
True
>>> M1 == M2
False
>>> M1 == M3
True
>>> M1.is_field_equivalent(M2)
False
>>> M1.is_field_equivalent(M3)
True
>>> M1.is_field_equivalent(M1)
True
```

is_field_isomorphic(*other*)

Test isomorphism between matroid representations.

Two represented matroids are *field isomorphic* if there is a bijection between their groundsets that induces a field equivalence between their representation matrices: the matrices are equal up to row operations and column scaling. This implies that the matroids are isomorphic, but the converse is false: two isomorphic matroids can be represented by matrices that are not field equivalent.

INPUT:

- `other` – matroid

OUTPUT: boolean

See also

M.is_isomorphic(), *M.is_field_isomorphism()*, *M.is_field_equivalent()*

EXAMPLES:

```
sage: M1 = matroids.Wheel(3)
sage: M2 = Matroid(graphs.CompleteGraph(4), regular=True)                    #␣
↪needs sage.graphs
sage: M1.is_field_isomorphic(M2)                                             #␣
↪needs sage.graphs
True
sage: M3 = Matroid(bases=M1.bases())
sage: M1.is_field_isomorphic(M3)
Traceback (most recent call last):
...
AttributeError: 'sage.matroids.basis_matroid.BasisMatroid' object
has no attribute 'base_ring'...
sage: from sage.matroids.advanced import *
```

(continues on next page)

(continued from previous page)

```
sage: M4 = BinaryMatroid(Matrix(M1))
sage: M5 = LinearMatroid(reduced_matrix=Matrix(GF(2), [[-1, 0, 1],
....:                                   [1, -1, 0], [0, 1, -1]]))
sage: M4.is_field_isomorphic(M5)
True

sage: M1 = Matroid(groundset=[0, 1, 2, 3], matrix=Matrix(GF(7),
....:                               [[1, 0, 1, 1], [0, 1, 1, 2]]))
sage: M2 = Matroid(groundset=[0, 1, 2, 3], matrix=Matrix(GF(7),
....:                               [[1, 0, 1, 1], [0, 1, 2, 1]]))
sage: M1.is_field_isomorphic(M2)
True
sage: M1.is_field_equivalent(M2)
False
```

```
>>> from sage.all import *
>>> M1 = matroids.Wheel(Integer(3))
>>> M2 = Matroid(graphs.CompleteGraph(Integer(4)), regular=True)␣
↪     # needs sage.graphs
>>> M1.is_field_isomorphic(M2)                                          #␣
↪needs sage.graphs
True
>>> M3 = Matroid(bases=M1.bases())
>>> M1.is_field_isomorphic(M3)
Traceback (most recent call last):
...
AttributeError: 'sage.matroids.basis_matroid.BasisMatroid' object
has no attribute 'base_ring'...
>>> from sage.matroids.advanced import *
>>> M4 = BinaryMatroid(Matrix(M1))
>>> M5 = LinearMatroid(reduced_matrix=Matrix(GF(Integer(2)), [[-Integer(1),␣
↪Integer(0), Integer(1)],
...                                   [Integer(1), -Integer(1), Integer(0)],␣
↪[Integer(0), Integer(1), -Integer(1)]]))
>>> M4.is_field_isomorphic(M5)
True

>>> M1 = Matroid(groundset=[Integer(0), Integer(1), Integer(2), Integer(3)],␣
↪matrix=Matrix(GF(Integer(7)),
...                                [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2)]]))
>>> M2 = Matroid(groundset=[Integer(0), Integer(1), Integer(2), Integer(3)],␣
↪matrix=Matrix(GF(Integer(7)),
...                                [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(2), Integer(1)]]))
>>> M1.is_field_isomorphic(M2)
True
>>> M1.is_field_equivalent(M2)
False
```

is_field_isomorphism(*other*, *morphism*)

Test if a provided morphism induces a bijection between represented matroids.

Two represented matroids are *field isomorphic* if the bijection `morphism` between them induces a field equivalence between their representation matrices: the matrices are equal up to row operations and column scaling. This implies that the matroids are isomorphic, but the converse is false: two isomorphic matroids can be represented by matrices that are not field equivalent.

INPUT:

- `other` – matroid
- `morphism` – a map from the groundset of `self` to the groundset of `other`. See documentation of the *M.is_isomorphism()* method for more on what is accepted as input.

OUTPUT: boolean

See also

M.is_isomorphism(), *M.is_field_equivalent()*, *M.is_field_isomorphic()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: N = matroids.catalog.NonFano()
sage: N.is_field_isomorphism(M, {e:e for e in M.groundset()})
False

sage: from sage.matroids.advanced import *
sage: M = matroids.catalog.Fano().delete(['g'])
sage: N = LinearMatroid(reduced_matrix=Matrix(GF(2),
....:                       [[-1, 0, 1], [1, -1, 0], [0, 1, -1]]))
sage: morphism = {'a':0, 'b':1, 'c': 2, 'd':4, 'e':5, 'f':3}
sage: M.is_field_isomorphism(N, morphism)
True

sage: M1 = Matroid(groundset=[0, 1, 2, 3], matrix=Matrix(GF(7),
....:                                 [[1, 0, 1, 1], [0, 1, 1, 2]]))
sage: M2 = Matroid(groundset=[0, 1, 2, 3], matrix=Matrix(GF(7),
....:                                 [[1, 0, 1, 1], [0, 1, 2, 1]]))
sage: mf1 = {0:0, 1:1, 2:2, 3:3}
sage: mf2 = {0:0, 1:1, 2:3, 3:2}
sage: M1.is_field_isomorphism(M2, mf1)
False
sage: M1.is_field_isomorphism(M2, mf2)
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> N = matroids.catalog.NonFano()
>>> N.is_field_isomorphism(M, {e:e for e in M.groundset()})
False

>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.Fano().delete(['g'])
>>> N = LinearMatroid(reduced_matrix=Matrix(GF(Integer(2)),
...                       [[-Integer(1), Integer(0), Integer(1)], [Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪ -Integer(1), Integer(0)], [Integer(0), Integer(1), -Integer(1)]]))
>>> morphism = {'a':Integer(0), 'b':Integer(1), 'c': Integer(2), 'd
↪':Integer(4), 'e':Integer(5), 'f':Integer(3)}
>>> M.is_field_isomorphism(N, morphism)
True

>>> M1 = Matroid(groundset=[Integer(0), Integer(1), Integer(2), Integer(3)],␣
↪matrix=Matrix(GF(Integer(7)),
...                                [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2)]]))
>>> M2 = Matroid(groundset=[Integer(0), Integer(1), Integer(2), Integer(3)],␣
↪matrix=Matrix(GF(Integer(7)),
...                                [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(2), Integer(1)]]))
>>> mf1 = {Integer(0):Integer(0), Integer(1):Integer(1),␣
↪Integer(2):Integer(2), Integer(3):Integer(3)}
>>> mf2 = {Integer(0):Integer(0), Integer(1):Integer(1),␣
↪Integer(2):Integer(3), Integer(3):Integer(2)}
>>> M1.is_field_isomorphism(M2, mf1)
False
>>> M1.is_field_isomorphism(M2, mf2)
True
```

is_valid(*certificate=False*)

Test if the data represent an actual matroid.

Since this matroid is linear, we test the representation matrix.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: boolean, or (boolean, dictionary)

The boolean output value is:

- `True` if the matrix is over a field.
- `True` if the matrix is over a ring and all cross ratios are invertible.
- `False` otherwise.

Note

This function does NOT test if the cross ratios are contained in the appropriate set of fundamentals. To that end, use

`M.cross_ratios().issubset(F)`

where `F` is the set of fundamentals.

See also

M.cross_ratios()

EXAMPLES:

```
sage: M = Matroid(ring=QQ, reduced_matrix=Matrix(ZZ,
....:                           [[1, 0, 1], [1, 1, 0], [0, 1, 1]]))
sage: M.is_valid()
True
sage: from sage.matroids.advanced import *  # LinearMatroid
sage: M = LinearMatroid(ring=ZZ, reduced_matrix=Matrix(ZZ,
....:                           [[1, 0, 1], [1, 1, 0], [0, 1, 1]]))
sage: M.is_valid(certificate=True)
(False, {'error': 'not all cross ratios are invertible'})
```

```
>>> from sage.all import *
>>> M = Matroid(ring=QQ, reduced_matrix=Matrix(ZZ,
...                           [[Integer(1), Integer(0), Integer(1)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(0), Integer(1),␣
↪Integer(1)]]))
>>> M.is_valid()
True
>>> from sage.matroids.advanced import *  # LinearMatroid
>>> M = LinearMatroid(ring=ZZ, reduced_matrix=Matrix(ZZ,
...                           [[Integer(1), Integer(0), Integer(1)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(0), Integer(1),␣
↪Integer(1)]]))
>>> M.is_valid(certificate=True)
(False, {'error': 'not all cross ratios are invertible'})
```

linear_coextension(*element*, *cochain=None*, *row=None*)

Return a linear coextension of this matroid.

A *linear coextension* of the represented matroid M by element e is a matroid represented by

$$\begin{bmatrix} A & 0 \\ -c & 1 \end{bmatrix},$$

where A is a representation matrix of M, c is a new row, and the last column is labeled by e.

This is the dual method of *M.linear_extension()*.

INPUT:

- `element` – the name of the new element
- `row` – (default: `None`) a row to be appended to `self.representation()`; can be any iterable
- `cochain` – (default: `None`) a dictionary that maps elements of the groundset to elements of the base ring

OUTPUT:

A linear matroid $N = M([A0; -c1])$, where A is a matrix such that the current matroid is $M[A]$, and c is either given by `row` (relative to `self.representation()`) or has nonzero entries given by `cochain`.

Note

The minus sign is to ensure this method commutes with dualizing. See the last example.

See also

M.coextension(), *M.linear_extension()*, *M.dual()*

EXAMPLES:

```
sage: M = Matroid(ring=GF(2), matrix=[[1, 1, 0, 1, 0, 0],
....:                                 [1, 0, 1, 0, 1, 0],
....:                                 [0, 1, 1, 0, 0, 1],
....:                                 [0, 0, 0, 1, 1, 1]])
sage: M.linear_coextension(6, {0:1, 5: 1}).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 0]
[0 1 1 0 0 1 0]
[0 0 0 1 1 1 0]
[1 0 0 0 0 1 1]
sage: M.linear_coextension(6, row=[0,1,1,1,0,1]).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 0]
[0 1 1 0 0 1 0]
[0 0 0 1 1 1 0]
[0 1 1 1 0 1 1]
```

```
>>> from sage.all import *
>>> M = Matroid(ring=GF(Integer(2)), matrix=[[Integer(1), Integer(1),␣
↪Integer(0), Integer(1), Integer(0), Integer(0)],
...                                 [Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(0)],
...                                 [Integer(0), Integer(1), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)],
...                                 [Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)]])
>>> M.linear_coextension(Integer(6), {Integer(0):Integer(1), Integer(5):␣
↪Integer(1)}).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 0]
[0 1 1 0 0 1 0]
[0 0 0 1 1 1 0]
[1 0 0 0 0 1 1]
>>> M.linear_coextension(Integer(6), row=[Integer(0),Integer(1),Integer(1),
↪Integer(1),Integer(0),Integer(1)]).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 0]
[0 1 1 0 0 1 0]
[0 0 0 1 1 1 0]
[0 1 1 1 0 1 1]
```

Coextending commutes with dualizing:

```
sage: M = matroids.catalog.NonFano()
sage: chain = {'a': 1, 'b': -1, 'f': 1}
sage: M1 = M.linear_coextension('x', chain)
sage: M2 = M.dual().linear_extension('x', chain)
```

(continues on next page)

(continued from previous page)

```
sage: M1 == M2.dual()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonFano()
>>> chain = {'a': Integer(1), 'b': -Integer(1), 'f': Integer(1)}
>>> M1 = M.linear_coextension('x', chain)
>>> M2 = M.dual().linear_extension('x', chain)
>>> M1 == M2.dual()
True
```

linear_coextension_cochains(*F=None*, *cosimple=False*, *fundamentals=None*)

Create a list of cochains that determine the single-element coextensions of this linear matroid representation.

A cochain is a dictionary, mapping elements from the groundset to elements of the base ring. If A represents the current matroid, then the coextension is given by $N = M([A0; -c1])$, with the entries of c given by the cochain. Note that the matroid does not change when row operations are carried out on A.

INPUT:

- `F` – (default: `self.groundset()`) a subset of the groundset
- `cosimple` – boolean (default: `False`)
- `fundamentals` – (default: `None`) a set elements of the base ring

OUTPUT:

A list of cochains, so each single-element coextension of this linear matroid representation is given by one of these cochains.

If one or more of the above inputs is given, the list is restricted to chains

- so that the support of each cochain lies in `F`, if given
- so that the cochain does not generate a series extension or coloop, if `cosimple = True`
- so that in the coextension generated by this cochain, the cross ratios are restricted to `fundamentals`, if given.

See also

M.linear_coextension(), M.linear_coextensions(), M.cross_ratios()

EXAMPLES:

```
sage: M = Matroid(reduced_matrix=Matrix(GF(2),
....:                           [[1, 1, 0], [1, 0, 1], [0, 1, 1]]))
sage: len(M.linear_coextension_cochains())
8
sage: len(M.linear_coextension_cochains(F=[0, 1]))
4
sage: len(M.linear_coextension_cochains(F=[0, 1], cosimple=True))
0
sage: M.linear_coextension_cochains(F=[3, 4, 5], cosimple=True)
[{3: 1, 4: 1, 5: 1}]
```

(continues on next page)

(continued from previous page)

```
sage: N = Matroid(ring=QQ,
....:             reduced_matrix=[[-1, -1, 0], [1, 0, -1], [0, 1, 1]])
sage: N.linear_coextension_cochains(F=[0, 1], cosimple=True,
....:                               fundamentals=set([1, -1, 1/2, 2]))
[{0: 2, 1: 1}, {0: -1, 1: 1}, {0: 1/2, 1: 1}]
```

```
>>> from sage.all import *
>>> M = Matroid(reduced_matrix=Matrix(GF(Integer(2)),
...                           [[Integer(1), Integer(1), Integer(0)],␣
↪[Integer(1), Integer(0), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(1)]]))
>>> len(M.linear_coextension_cochains())
8
>>> len(M.linear_coextension_cochains(F=[Integer(0), Integer(1)]))
4
>>> len(M.linear_coextension_cochains(F=[Integer(0), Integer(1)],␣
↪cosimple=True))
0
>>> M.linear_coextension_cochains(F=[Integer(3), Integer(4), Integer(5)],␣
↪cosimple=True)
[{3: 1, 4: 1, 5: 1}]
>>> N = Matroid(ring=QQ,
...             reduced_matrix=[[-Integer(1), -Integer(1), Integer(0)],␣
↪[Integer(1), Integer(0), -Integer(1)], [Integer(0), Integer(1),␣
↪Integer(1)]])
>>> N.linear_coextension_cochains(F=[Integer(0), Integer(1)], cosimple=True,
...                               fundamentals=set([Integer(1), -Integer(1),␣
↪Integer(1)/Integer(2), Integer(2)]))
[{0: 2, 1: 1}, {0: -1, 1: 1}, {0: 1/2, 1: 1}]
```

linear_coextensions(*element=None*, *F=None*, *cosimple=False*, *fundamentals=None*)

Create a list of linear matroids represented by corank-preserving single-element coextensions of this linear matroid representation.

INPUT:

- `element` – (default: `None`) the name of the new element of the groundset
- `F` – (default: `None`) a subset of the groundset
- `cosimple` – boolean (default: `False`)
- `fundamentals` – (default: `None`) a set elements of the base ring

OUTPUT:

A list of linear matroids represented by corank-preserving single-element coextensions of this linear matroid representation. In particular, the coextension by a loop is not generated.

If one or more of the above inputs is given, the list is restricted to coextensions

- so that the new element lies in the cospan of `F`, if given.
- so that the new element is no coloop and is not in series with another element, if `cosimple = True`.
- so that in the representation of the coextension, the cross ratios are restricted to `fundamentals`, if given. Note that it is assumed that the cross ratios of the input matroid already satisfy this condition.

See also

M.linear_coextension(), *M.linear_coextension_cochains()*, *M.cross_ratios()*

EXAMPLES:

```
sage: M = Matroid(ring=GF(2),
....:         reduced_matrix=[[-1, 0, 1], [1, -1, 0], [0, 1, -1]])
sage: len(M.linear_coextensions())
8
sage: S = M.linear_coextensions(cosimple=True)
sage: S
[Binary matroid of rank 4 on 7 elements, type (3, 7)]
sage: F7 = matroids.catalog.Fano()
sage: S[0].is_field_isomorphic(F7.dual())
True
sage: M = Matroid(ring=QQ,
....:             reduced_matrix=[[1, 0, 1], [1, 1, 0], [0, 1, 1]])
sage: S = M.linear_coextensions(cosimple=True,
....:                           fundamentals=[1, -1, 1/2, 2])
sage: len(S)
7
sage: NF7 = matroids.catalog.NonFano()
sage: any(N.is_isomorphic(NF7.dual()) for N in S)
True
sage: len(M.linear_coextensions(cosimple=True,
....:                           fundamentals=[1, -1, 1/2, 2],
....:                           F=[3, 4]))
1
```

```
>>> from sage.all import *
>>> M = Matroid(ring=GF(Integer(2)),
...         reduced_matrix=[[-Integer(1), Integer(0), Integer(1)],␣
↪[Integer(1), -Integer(1), Integer(0)], [Integer(0), Integer(1), -
↪Integer(1)]])
>>> len(M.linear_coextensions())
8
>>> S = M.linear_coextensions(cosimple=True)
>>> S
[Binary matroid of rank 4 on 7 elements, type (3, 7)]
>>> F7 = matroids.catalog.Fano()
>>> S[Integer(0)].is_field_isomorphic(F7.dual())
True
>>> M = Matroid(ring=QQ,
...             reduced_matrix=[[Integer(1), Integer(0), Integer(1)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(0), Integer(1), Integer(1)]])
>>> S = M.linear_coextensions(cosimple=True,
...                           fundamentals=[Integer(1), -Integer(1),␣
↪Integer(1)/Integer(2), Integer(2)])
>>> len(S)
7
>>> NF7 = matroids.catalog.NonFano()
```

(continues on next page)

(continued from previous page)

```
>>> any(N.is_isomorphic(NF7.dual()) for N in S)
True
>>> len(M.linear_coextensions(cosimple=True,
...                           fundamentals=[Integer(1), -Integer(1),␣
↪Integer(1)/Integer(2), Integer(2)],
...                           F=[Integer(3), Integer(4)]))
1
```

linear_extension(*element*, *chain=None*, *col=None*)

Return a linear extension of this matroid.

A *linear extension* of the represented matroid M by element e is a matroid represented by $[A\ \ b]$, where A is a representation matrix of M and b is a new column labeled by e.

INPUT:

- `element` – the name of the new element
- `col` – (default: `None`) a column to be appended to `self.representation()`; can be any iterable
- `chain` – (default: `None`) a dictionary that maps elements of the groundset to elements of the base ring

OUTPUT:

A linear matroid $N = M([A\ \ b])$, where A is a matrix such that the current matroid is $M[A]$, and b is either given by `col` or is a weighted combination of columns of A, the weights being given by `chain`.

See also

M.extension().

EXAMPLES:

```
sage: M = Matroid(ring=GF(2), matrix=[[1, 1, 0, 1, 0, 0],
....:                                 [1, 0, 1, 0, 1, 0],
....:                                 [0, 1, 1, 0, 0, 1],
....:                                 [0, 0, 0, 1, 1, 1]])
sage: M.linear_extension(6, {0:1, 5: 1}).representation()
[1 1 0 1 0 0 1]
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
sage: M.linear_extension(6, col=[0, 1, 1, 1]).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
```

```
>>> from sage.all import *
>>> M = Matroid(ring=GF(Integer(2)), matrix=[[Integer(1), Integer(1),␣
↪Integer(0), Integer(1), Integer(0), Integer(0)],
...                                 [Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(0)],
...                                 [Integer(0), Integer(1), Integer(1),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0), Integer(0), Integer(1)],
...                                  [Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)]])
>>> M.linear_extension(Integer(6), {Integer(0):Integer(1), Integer(5):␣
↪Integer(1)}).representation()
[1 1 0 1 0 0 1]
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
>>> M.linear_extension(Integer(6), col=[Integer(0), Integer(1), Integer(1),␣
↪Integer(1)]).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
```

linear_extension_chains(*F=None*, *simple=False*, *fundamentals=None*)

Create a list of chains that determine the single-element extensions of this linear matroid representation.

A *chain* is a dictionary, mapping elements from the groundset to elements of the base ring, indicating a linear combination of columns to form the new column. Think of chains as vectors, only independent of representation.

INPUT:

- `F` – (default: `self.groundset()`) a subset of the groundset
- `simple` – boolean (default: `False`)
- `fundamentals` – (default: `None`) a set elements of the base ring

OUTPUT:

A list of chains, so each single-element extension of this linear matroid representation is given by one of these chains.

If one or more of the above inputs is given, the list is restricted to chains

- so that the support of each chain lies in `F`, if given
- so that the chain does not generate a parallel extension or loop, if `simple = True`
- so that in the extension generated by this chain, the cross ratios are restricted to `fundamentals`, if given.

See also

M.linear_extension(), M.linear_extensions(), M.cross_ratios()

EXAMPLES:

```
sage: M = Matroid(reduced_matrix=Matrix(GF(2),
....:                           [[1, 1, 0], [1, 0, 1], [0, 1, 1]]))
sage: len(M.linear_extension_chains())
8
sage: len(M.linear_extension_chains(F=[0, 1]))
4
```

(continues on next page)

(continued from previous page)

```
sage: len(M.linear_extension_chains(F=[0, 1], simple=True))
0
sage: M.linear_extension_chains(F=[0, 1, 2], simple=True)
[{0: 1, 1: 1, 2: 1}]
sage: N = Matroid(ring=QQ,
....:             reduced_matrix=[[-1, -1, 0], [1, 0, -1], [0, 1, 1]])
sage: L = N.linear_extension_chains(F=[0, 1], simple=True,
....:                               fundamentals=set([1, -1, 1/2, 2]))
sage: result = [{0: 1, 1: 1}, {0: -1/2, 1: 1}, {0: -2, 1: 1}]
sage: all(D in L for D in result)
True
```

```
>>> from sage.all import *
>>> M = Matroid(reduced_matrix=Matrix(GF(Integer(2)),
...                           [[Integer(1), Integer(1), Integer(0)],␣
↪[Integer(1), Integer(0), Integer(1)], [Integer(0), Integer(1),␣
↪Integer(1)]]))
>>> len(M.linear_extension_chains())
8
>>> len(M.linear_extension_chains(F=[Integer(0), Integer(1)]))
4
>>> len(M.linear_extension_chains(F=[Integer(0), Integer(1)], simple=True))
0
>>> M.linear_extension_chains(F=[Integer(0), Integer(1), Integer(2)],␣
↪simple=True)
[{0: 1, 1: 1, 2: 1}]
>>> N = Matroid(ring=QQ,
...             reduced_matrix=[[-Integer(1), -Integer(1), Integer(0)],␣
↪[Integer(1), Integer(0), -Integer(1)], [Integer(0), Integer(1),␣
↪Integer(1)]])
>>> L = N.linear_extension_chains(F=[Integer(0), Integer(1)], simple=True,
...                               fundamentals=set([Integer(1), -Integer(1),␣
↪Integer(1)/Integer(2), Integer(2)]))
>>> result = [{Integer(0): Integer(1), Integer(1): Integer(1)}, {Integer(0): -
↪Integer(1)/Integer(2), Integer(1): Integer(1)}, {Integer(0): -Integer(2),␣
↪Integer(1): Integer(1)}]
>>> all(D in L for D in result)
True
```

linear_extensions(*element=None*, *F=None*, *simple=False*, *fundamentals=None*)

Create a list of linear matroids represented by rank-preserving single-element extensions of this linear matroid representation.

INPUT:

- `element` – (default: `None`) the name of the new element of the groundset
- `F` – (default: `None`) a subset of the groundset
- `simple` – boolean (default: `False`)
- `fundamentals` – (default: `None`) a set elements of the base ring

OUTPUT:

A list of linear matroids represented by rank-preserving single-element extensions of this linear matroid representation. In particular, the extension by a coloop is not generated.

If one or more of the above inputs is given, the list is restricted to matroids

- so that the new element is spanned by `F`, if given
- so that the new element is not a loop or in a parallel pair, if `simple=True`
- so that in the representation of the extension, the cross ratios are restricted to `fundamentals`, if given. Note that it is assumed that the cross ratios of the input matroid already satisfy this condition.

See also

M.linear_extension(), M.linear_extension_chains(), M.cross_ratios()

EXAMPLES:

```
sage: M = Matroid(ring=GF(2),
....:             reduced_matrix=[[-1, 0, 1], [1, -1, 0], [0, 1, -1]])
sage: len(M.linear_extensions())
8
sage: S = M.linear_extensions(simple=True); S
[Binary matroid of rank 3 on 7 elements, type (3, 0)]
sage: S[0].is_field_isomorphic(matroids.catalog.Fano())
True
sage: M = Matroid(ring=QQ,
....:             reduced_matrix=[[1, 0, 1], [1, 1, 0], [0, 1, 1]])
sage: S = M.linear_extensions(simple=True,
....:                         fundamentals=[1, -1, 1/2, 2])
sage: len(S)
7
sage: any(N.is_isomorphic(matroids.catalog.NonFano())
....:     for N in S)
True
sage: len(M.linear_extensions(simple=True,
....:                         fundamentals=[1, -1, 1/2, 2], F=[0, 1]))
1
```

```
>>> from sage.all import *
>>> M = Matroid(ring=GF(Integer(2)),
...             reduced_matrix=[[-Integer(1), Integer(0), Integer(1)],␣
↪[Integer(1), -Integer(1), Integer(0)], [Integer(0), Integer(1), -
↪Integer(1)]])
>>> len(M.linear_extensions())
8
>>> S = M.linear_extensions(simple=True); S
[Binary matroid of rank 3 on 7 elements, type (3, 0)]
>>> S[Integer(0)].is_field_isomorphic(matroids.catalog.Fano())
True
>>> M = Matroid(ring=QQ,
...             reduced_matrix=[[Integer(1), Integer(0), Integer(1)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(0), Integer(1), Integer(1)]])
>>> S = M.linear_extensions(simple=True,
```

(continues on next page)

(continued from previous page)

```
...                          fundamentals=[Integer(1), -Integer(1), Integer(1)/
↪Integer(2), Integer(2)])
>>> len(S)
7
>>> any(N.is_isomorphic(matroids.catalog.NonFano())
...      for N in S)
True
>>> len(M.linear_extensions(simple=True,
...                          fundamentals=[Integer(1), -Integer(1), Integer(1)/
↪Integer(2), Integer(2)], F=[Integer(0), Integer(1)]))
1
```

orlik_terao_algebra(*R=None*, *ordering=None*, ***kwargs*)

Return the Orlik-Terao algebra of `self`.

INPUT:

- `R` – (default: the base ring of `self`) the base ring
- `ordering` – (optional) an ordering of the groundset

See also

`OrlikTeraoAlgebra`

EXAMPLES:

```
sage: M = matroids.Wheel(3)
sage: OS = M.orlik_terao_algebra(); OS
Orlik-Terao algebra of Wheel(3):
 Regular matroid of rank 3 on 6 elements with 16 bases
 over Integer Ring
sage: OS.base_ring()
Integer Ring
sage: M.orlik_terao_algebra(QQ).base_ring()
Rational Field

sage: G = SymmetricGroup(3);                                                    #␣
↪needs sage.groups
sage: OTG = M.orlik_terao_algebra(QQ, invariant=G)                              #␣
↪needs sage.groups

sage: # needs sage.groups
sage: G = SymmetricGroup(4)
sage: action = lambda g, x: g(x + 1) - 1
sage: OTG1 = M.orlik_terao_algebra(QQ, invariant=(G, action))
sage: OTG2 = M.orlik_terao_algebra(QQ, invariant=(action, G))
sage: OTG1 is OTG2
True
```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> OS = M.orlik_terao_algebra(); OS
Orlik-Terao algebra of Wheel(3):
 Regular matroid of rank 3 on 6 elements with 16 bases
 over Integer Ring
>>> OS.base_ring()
Integer Ring
>>> M.orlik_terao_algebra(QQ).base_ring()
Rational Field

>>> G = SymmetricGroup(Integer(3));
↪     # needs sage.groups
>>> OTG = M.orlik_terao_algebra(QQ, invariant=G)
↪needs sage.groups

>>> # needs sage.groups
>>> G = SymmetricGroup(Integer(4))
>>> action = lambda g, x: g(x + Integer(1)) - Integer(1)
>>> OTG1 = M.orlik_terao_algebra(QQ, invariant=(G, action))
>>> OTG2 = M.orlik_terao_algebra(QQ, invariant=(action, G))
>>> OTG1 is OTG2
True
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
sage: N = M.relabel({'g': 'x'})
sage: sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'x']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> N = M.relabel({'g': 'x'})
>>> sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'x']
```

representation(*B=None*, *reduced=False*, *labels=None*, *order=None*, *lift_map=None*)

Return a matrix representing the matroid.

Let M be a matroid on n elements with rank r. Let E be an ordering of the groundset, as output by *M.groundset_list()*. A *representation* of the matroid is an $r \times n$ matrix with the following property. Consider column i to be labeled by $E[i]$, and denote by $A[F]$ the submatrix formed by the columns labeled by the subset $F \subseteq E$. Then for all $F \subseteq E$, the columns of $A[F]$ are linearly independent if and only if F is an independent set in the matroid.

A *reduced representation* is a matrix D such that $[I \ D]$ is a representation of the matroid, where I is an $r \times r$ identity matrix. In this case, the rows of D are considered to be labeled by the first r elements of the list `E`, and the columns by the remaining $n - r$ elements.

INPUT:

- `B` – (default: `None`) a subset of elements. When provided, the representation is such that a basis B' that maximally intersects B is an identity matrix.
- `reduced` – boolean (default: `False`); when `True`, return a reduced matrix D (so $[I \ D]$ is a representation of the matroid). Otherwise return a full representation matrix.
- `labels` – (default: `None`) when `True`, return additionally a list of column labels (if `reduced=False`) or a list of row labels and a list of column labels (if `reduced=True`). The default setting, `None`, will not return the labels for a full matrix, but will return the labels for a reduced matrix.
- `order` – sequence or `None` or `True` (default: `None`)
  - when a sequence, it should be an ordering of the groundset elements, and the columns (and, in case of a reduced representation, rows) will be presented in the given order,
  - when `None`, use the same ordering that `groundset_list()` uses,
  - when `True`, return a morphism of free modules instead of a matrix.
- `lift_map` – (default: `None`) a dictionary containing the cross ratios of the representing matrix in its domain. If provided, the representation will be transformed by mapping its cross ratios according to `lift_map`.

OUTPUT:

- `A` – a full or reduced representation matrix of `self`; or
- `(A, E)` – a full representation matrix `A` and a list `E` of column labels; or
- `(A, R, C)` – a reduced representation matrix and a list `R` of row labels and a list `C` of column labels

If `B == None` and `reduced == False` and `order == None` then this method will always output the same matrix (except when `M._forget()` is called): either the matrix used as input to create the matroid, or a matrix in which the lexicographically least basis corresponds to an identity. If only `order` is not `None`, the columns of this matrix will be permuted accordingly.

If a `lift_map` is provided, then the resulting matrix will be lifted using the method *lift_cross_ratios()* See the docstring of this method for further details.

Note

A shortcut for `M.representation()` is `Matrix(M)`.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.representation()
[1 0 0 0 1 1 1]
```

(continues on next page)

(continued from previous page)

```
[0 1 0 1 0 1 1]
[0 0 1 1 1 0 1]
sage: Matrix(M) == M.representation()
True
sage: M.representation(labels=True)
(
[1 0 0 0 1 1 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 0 1], ['a', 'b', 'c', 'd', 'e', 'f', 'g']
)
sage: M.representation(B='efg')
[1 1 0 1 1 0 0]
[1 0 1 1 0 1 0]
[1 1 1 0 0 0 1]
sage: M.representation(B='efg', order='efgabcd')
[1 0 0 1 1 0 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 1 0]
sage: M.representation(B='abc', reduced=True)
(
[0 1 1 1]
[1 0 1 1]
[1 1 0 1], ['a', 'b', 'c'], ['d', 'e', 'f', 'g']
)
sage: M.representation(B='efg', reduced=True, labels=False,
....:                  order='gfeabcd')
[1 1 1 0]
[1 0 1 1]
[1 1 0 1]

sage: from sage.matroids.advanced import lift_cross_ratios, lift_map,␣
↪LinearMatroid
sage: R = GF(7)
sage: A = Matrix(R, [[1, 0, 6, 1, 2],[6, 1, 0, 0, 1],[0, 6, 3, 6, 0]])
sage: M = LinearMatroid(reduced_matrix=A)
sage: M.representation(lift_map=lift_map('sru'))                              #␣
↪needs sage.rings.finite_rings
[     1      0      0      1      0      1      1      1]
[     0      1      0 -z + 1      1      0      0      1]
[     0      0      1      0      1      -1 z - 1      0]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.representation()
[1 0 0 0 1 1 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 0 1]
>>> Matrix(M) == M.representation()
True
>>> M.representation(labels=True)
(
[1 0 0 0 1 1 1]
```

(continues on next page)

(continued from previous page)

```
[0 1 0 1 0 1 1]
[0 0 1 1 1 0 1], ['a', 'b', 'c', 'd', 'e', 'f', 'g']
)
>>> M.representation(B='efg')
[1 1 0 1 1 0 0]
[1 0 1 1 0 1 0]
[1 1 1 0 0 0 1]
>>> M.representation(B='efg', order='efgabcd')
[1 0 0 1 1 0 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 1 0]
>>> M.representation(B='abc', reduced=True)
(
[0 1 1 1]
[1 0 1 1]
[1 1 0 1], ['a', 'b', 'c'], ['d', 'e', 'f', 'g']
)
>>> M.representation(B='efg', reduced=True, labels=False,
...                  order='gfeabcd')
[1 1 1 0]
[1 0 1 1]
[1 1 0 1]

>>> from sage.matroids.advanced import lift_cross_ratios, lift_map,␣
↪LinearMatroid
>>> R = GF(Integer(7))
>>> A = Matrix(R, [[Integer(1), Integer(0), Integer(6), Integer(1),␣
↪Integer(2)],[Integer(6), Integer(1), Integer(0), Integer(0), Integer(1)],
↪[Integer(0), Integer(6), Integer(3), Integer(6), Integer(0)]])
>>> M = LinearMatroid(reduced_matrix=A)
>>> M.representation(lift_map=lift_map('sru'))                              #␣
↪needs sage.rings.finite_rings
[     1      0      0      1      0      1      1      1]
[     0      1      0 -z + 1      1      0      0      1]
[     0      0      1      0      1     -1 z - 1      0]
```

As morphisms:

```
sage: M = matroids.catalog.Fano()
sage: A = M.representation(order=True); A
Generic morphism:
 From: Free module generated by {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
       over Finite Field of size 2
 To:   Free module generated by {0, 1, 2} over Finite Field of size 2
sage: print(A._unicode_art_matrix())
  a b c d e f g
0⎛1 0 0 0 1 1 1⎞
1⎜0 1 0 1 0 1 1⎟
2⎝0 0 1 1 1 0 1⎠
sage: A = M.representation(B='efg', order=True); A
Generic morphism:
 From: Free module generated by {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
```

(continues on next page)

(continued from previous page)

```
      over Finite Field of size 2
 To:   Free module generated by {0, 1, 2} over Finite Field of size 2
sage: print(A._unicode_art_matrix())
  a b c d e f g
0⎛1 1 0 1 1 0 0⎞
1⎜1 0 1 1 0 1 0⎟
2⎝1 1 1 0 0 0 1⎠
sage: A = M.representation(B='abc', order=True, reduced=True); A
Generic morphism:
 From: Free module generated by {'d', 'e', 'f', 'g'}
       over Finite Field of size 2
 To:   Free module generated by {'a', 'b', 'c'} over Finite Field of size 2
sage: print(A._unicode_art_matrix())
  d e f g
a⎛0 1 1 1⎞
b⎜1 0 1 1⎟
c⎝1 1 0 1⎠
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> A = M.representation(order=True); A
Generic morphism:
 From: Free module generated by {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
       over Finite Field of size 2
 To:   Free module generated by {0, 1, 2} over Finite Field of size 2
>>> print(A._unicode_art_matrix())
  a b c d e f g
0⎛1 0 0 0 1 1 1⎞
1⎜0 1 0 1 0 1 1⎟
2⎝0 0 1 1 1 0 1⎠
>>> A = M.representation(B='efg', order=True); A
Generic morphism:
 From: Free module generated by {'a', 'b', 'c', 'd', 'e', 'f', 'g'}
       over Finite Field of size 2
 To:   Free module generated by {0, 1, 2} over Finite Field of size 2
>>> print(A._unicode_art_matrix())
  a b c d e f g
0⎛1 1 0 1 1 0 0⎞
1⎜1 0 1 1 0 1 0⎟
2⎝1 1 1 0 0 0 1⎠
>>> A = M.representation(B='abc', order=True, reduced=True); A
Generic morphism:
 From: Free module generated by {'d', 'e', 'f', 'g'}
       over Finite Field of size 2
 To:   Free module generated by {'a', 'b', 'c'} over Finite Field of size 2
>>> print(A._unicode_art_matrix())
  d e f g
a⎛0 1 1 1⎞
b⎜1 0 1 1⎟
c⎝1 1 0 1⎠
```

representation_vectors()

Return a dictionary that associates a column vector with each element of the matroid.

See also

M.representation()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: E = M.groundset_list()
sage: [M.representation_vectors()[e] for e in E]
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 0),
 (1, 1, 1)]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> E = M.groundset_list()
>>> [M.representation_vectors()[e] for e in E]
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 1), (1, 0, 1), (1, 1, 0),
 (1, 1, 1)]
```

class sage.matroids.linear_matroid.**QuaternaryMatroid**

Bases: *LinearMatroid*

Quaternary matroids.

A quaternary matroid is a linear matroid represented over the finite field with four elements. See *LinearMatroid* for a definition.

The simplest way to create a `QuaternaryMatroid` is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object E can be given as a groundset. If E is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a 'reduced' matrix B, to create the matroid induced by $A = [I \ B]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \ B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.
- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) must be a copy of $\mathbf{F}_4$
- `keep_initial_representation` – boolean (default: `True`); decides whether or not an internal copy of the input matrix should be preserved. This can help to see the structure of the matroid (e.g. in the case of graphic matroids), and makes it easier to look at extensions. However, the input matrix may have redundant rows, and sometimes it is desirable to store only a row-reduced copy.
- `basis` – (default: `None`) when provided, this is an ordered subset of `groundset`, such that the submatrix of `matrix` indexed by `basis` is an identity matrix. In this case, no row reduction takes place in the initialization phase.

OUTPUT: a `QuaternaryMatroid` instance based on the data above

Note

The recommended way to generate a quaternary matroid is through the *Matroid()* function. This is usually the preferred way, since it automatically chooses between QuaternaryMatroid and other classes. For direct access to the QuaternaryMatroid constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: GF4 = GF(4, 'x')
sage: x = GF4.gens()[0]
sage: A = Matrix(GF4, 2, 4, [[1, 0, 1, 1], [0, 1, 1, x]])
sage: M = Matroid(A)
sage: M
Quaternary matroid of rank 2 on 4 elements
sage: sorted(M.groundset())
[0, 1, 2, 3]
sage: Matrix(M)
[1 0 1 1]
[0 1 1 x]
sage: M = Matroid(matrix=A, groundset='abcd')
sage: sorted(M.groundset())
['a', 'b', 'c', 'd']
sage: GF4p = GF(4, 'y')
sage: y = GF4p.gens()[0]
sage: B = Matrix(GF4p, 2, 2, [[1, 1], [1, y]])
sage: N = Matroid(reduced_matrix=B, groundset='abcd')
sage: M == N
False
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> GF4 = GF(Integer(4), 'x')
>>> x = GF4.gens()[Integer(0)]
>>> A = Matrix(GF4, Integer(2), Integer(4), [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), x]])
>>> M = Matroid(A)
>>> M
Quaternary matroid of rank 2 on 4 elements
>>> sorted(M.groundset())
[0, 1, 2, 3]
>>> Matrix(M)
[1 0 1 1]
[0 1 1 x]
>>> M = Matroid(matrix=A, groundset='abcd')
>>> sorted(M.groundset())
['a', 'b', 'c', 'd']
```

(continues on next page)

(continued from previous page)

```
>>> GF4p = GF(Integer(4), 'y')
>>> y = GF4p.gens()[Integer(0)]
>>> B = Matrix(GF4p, Integer(2), Integer(2), [[Integer(1), Integer(1)],␣
↪[Integer(1), y]])
>>> N = Matroid(reduced_matrix=B, groundset='abcd')
>>> M == N
False
```

base_ring()

Return the base ring of the matrix representing the matroid, in this case $\mathbf{F}_4$.

EXAMPLES:

```
sage: M = Matroid(ring=GF(4, 'y'), reduced_matrix=[[1, 0, 1],                #␣
↪needs sage.rings.finite_rings
....:                                                 [0, 1, 1]])
sage: M.base_ring()                                                          #␣
↪needs sage.rings.finite_rings
Finite Field in y of size 2^2
```

```
>>> from sage.all import *
>>> M = Matroid(ring=GF(Integer(4), 'y'), reduced_matrix=[[Integer(1),␣
↪Integer(0), Integer(1)],                # needs sage.rings.finite_rings
...                                                 [Integer(0), Integer(1),␣
↪Integer(1)]])
>>> M.base_ring()                                                            #␣
↪needs sage.rings.finite_rings
Finite Field in y of size 2^2
```

bicycle_dimension()

Return the bicycle dimension of the quaternary matroid.

The bicycle dimension of a linear subspace V is $\dim(V \cap V^{\perp})$. We use the inner product $< v, w >= v_1 w_1^* + ... + v_n w_n^*$, where w_i^* is obtained from w_i by applying the unique nontrivial field automorphism of $\mathbf{F}_4$.

The bicycle dimension of a matroid equals the bicycle dimension of its rowspace, and is a matroid invariant. See [Pen2012].

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.Q10()                                             #␣
↪needs sage.rings.finite_rings
sage: M.bicycle_dimension()                                                  #␣
↪needs sage.rings.finite_rings
0
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Q10()                                               #␣
↪needs sage.rings.finite_rings
>>> M.bicycle_dimension()                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings
0
```

characteristic()

Return the characteristic of the base ring of the matrix representing the matroid, in this case 2.

EXAMPLES:

```
sage: M = Matroid(ring=GF(4, 'y'), reduced_matrix=[[1, 0, 1],                #␣
↪needs sage.rings.finite_rings
....:                                                 [0, 1, 1]])
sage: M.characteristic()                                                     #␣
↪needs sage.rings.finite_rings
2
```

```
>>> from sage.all import *
>>> M = Matroid(ring=GF(Integer(4), 'y'), reduced_matrix=[[Integer(1),␣
↪Integer(0), Integer(1)],                # needs sage.rings.finite_rings
...                                                 [Integer(0), Integer(1),␣
↪Integer(1)]])
>>> M.characteristic()                                                       #␣
↪needs sage.rings.finite_rings
2
```

is_valid(*certificate=False*)

Test if the data obey the matroid axioms.

Since this is a linear matroid over the field $\mathbf{F}_4$, this is always the case.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: `True`, or `(True, {})`

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(4, 'x'), [[]]))                                  #␣
↪needs sage.rings.finite_rings
sage: M.is_valid()                                                           #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(4), 'x'), [[]]))                            ␣
↪     # needs sage.rings.finite_rings
>>> M.is_valid()                                                             #␣
↪needs sage.rings.finite_rings
True
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: M = matroids.catalog.RelaxedNonFano("abcdefg")
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
sage: N = M.relabel({'g':'x'})
sage: sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'x']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.RelaxedNonFano("abcdefg")
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> N = M.relabel({'g':'x'})
>>> sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'x']
```

class sage.matroids.linear_matroid.**RegularMatroid**

Bases: *LinearMatroid*

Regular matroids.

A regular matroid is a linear matroid represented over the integers by a totally unimodular matrix.

The simplest way to create a `RegularMatroid` is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object E can be given as a groundset. If E is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a 'reduced' matrix B, to create the matroid induced by $A = [IB]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \quad B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.
- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) ignored
- `keep_initial_representation` – boolean (default: `True`); decides whether or not an internal copy of the input matrix should be preserved. This can help to see the structure of the matroid (e.g. in the case of graphic matroids), and makes it easier to look at extensions. However, the input matrix may have redundant rows, and sometimes it is desirable to store only a row-reduced copy.
- `basis` – (default: `None`) when provided, this is an ordered subset of `groundset`, such that the submatrix of `matrix` indexed by `basis` is an identity matrix. In this case, no row reduction takes place in the initialization phase.

OUTPUT: a `RegularMatroid` instance based on the data above

Note

The recommended way to generate a regular matroid is through the *Matroid()* function. This is usually the preferred way, since it automatically chooses between RegularMatroid and other classes. Moreover, it will test whether the input actually yields a regular matroid, unlike this class. For direct access to the RegularMatroid constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

Warning

No checks are performed to ensure the input data form an actual regular matroid! If not, the behavior is unpredictable, and the internal representation can get corrupted. If in doubt, run *self.is_valid()* to ensure the data are as desired.

EXAMPLES:

```
sage: A = Matrix(ZZ, 2, 4, [[1, 0, 1, 1], [0, 1, 1, 1]])
sage: M = Matroid(A, regular=True); M                                          #␣
↪needs sage.graphs
Regular matroid of rank 2 on 4 elements with 5 bases
sage: sorted(M.groundset())                                                    #␣
↪needs sage.graphs
[0, 1, 2, 3]
sage: Matrix(M)                                                                #␣
↪needs sage.graphs
[1 0 1 1]
[0 1 1 1]
sage: M = Matroid(matrix=A, groundset='abcd', regular=True)                    #␣
↪needs sage.graphs
sage: sorted(M.groundset())                                                    #␣
↪needs sage.graphs
['a', 'b', 'c', 'd']
```

```
>>> from sage.all import *
>>> A = Matrix(ZZ, Integer(2), Integer(4), [[Integer(1), Integer(0), Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(1)]])
>>> M = Matroid(A, regular=True); M                                            #␣
↪needs sage.graphs
Regular matroid of rank 2 on 4 elements with 5 bases
>>> sorted(M.groundset())                                                      #␣
↪needs sage.graphs
[0, 1, 2, 3]
>>> Matrix(M)                                                                  #␣
↪needs sage.graphs
[1 0 1 1]
[0 1 1 1]
>>> M = Matroid(matrix=A, groundset='abcd', regular=True)                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
>>> sorted(M.groundset())                                                    #␣
↪needs sage.graphs
['a', 'b', 'c', 'd']
```

base_ring()

Return the base ring of the matrix representing the matroid, in this case **Z**.

EXAMPLES:

```
sage: M = matroids.catalog.R10()
sage: M.base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R10()
>>> M.base_ring()
Integer Ring
```

bases_count()

Count the number of bases.

EXAMPLES:

```
sage: M = Matroid(graphs.CompleteGraph(5), regular=True)                  #␣
↪needs sage.graphs
sage: M.bases_count()                                                     #␣
↪needs sage.graphs
125
```

```
>>> from sage.all import *
>>> M = Matroid(graphs.CompleteGraph(Integer(5)), regular=True)              ␣
↪     # needs sage.graphs
>>> M.bases_count()                                                       #␣
↪needs sage.graphs
125
```

ALGORITHM:

Since the matroid is regular, we use Kirchhoff's Matrix-Tree Theorem. See also Wikipedia article Kirchhoff%27s_theorem.

binary_matroid(*randomized_tests=1*, *verify=True*)

Return a binary matroid representing `self`.

INPUT:

- `randomized_tests` – ignored
- `verify` – ignored

OUTPUT: a binary matroid

ALGORITHM:

`self` is a regular matroid, so just cast `self` to a BinaryMatroid.

See also

M.binary_matroid()

EXAMPLES:

```
sage: N = matroids.catalog.R10()
sage: N.binary_matroid()
Binary matroid of rank 5 on 10 elements, type (1, None)
```

```
>>> from sage.all import *
>>> N = matroids.catalog.R10()
>>> N.binary_matroid()
Binary matroid of rank 5 on 10 elements, type (1, None)
```

characteristic()

Return the characteristic of the base ring of the matrix representing the matroid, in this case 0.

EXAMPLES:

```
sage: M = matroids.catalog.R10()
sage: M.characteristic()
0
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R10()
>>> M.characteristic()
0
```

has_line_minor(*k*, *hyperlines=None*, *certificate=False*)

Test if the matroid has a $U_{2,k}$-minor.

The matroid $U_{2,k}$ is a matroid on k elements in which every subset of at most 2 elements is independent, and every subset of more than two elements is dependent.

The optional argument `hyperlines` restricts the search space: this method returns `True` if $si(M/F)$ is isomorphic to $U_{2,l}$ with $l \geq k$ for some F in `hyperlines`, and `False` otherwise.

INPUT:

- `k` – the length of the line minor
- `hyperlines` – (default: `None`) a set of flats of codimension 2. Defaults to the set of all flats of codimension 2.
- `certificate` – (default: `False`) if `True` returns `True, F`, where `F` is a flat and `self.minor(contractions=F)` has a $U_{2,k}$ restriction or `False, None`

OUTPUT: boolean or tuple

See also

Matroid.has_minor()

EXAMPLES:

```
sage: M = matroids.catalog.R10()
sage: M.has_line_minor(4)
False
sage: M.has_line_minor(4, certificate=True)
(False, None)
sage: M.has_line_minor(3)
True
sage: M.has_line_minor(3, certificate=True)
(True, frozenset({'a', 'b', 'c', 'g'}))
sage: M.has_line_minor(k=3, hyperlines=[['a', 'b', 'c'],
....:                                   ['a', 'b', 'd' ]])
True
sage: M.has_line_minor(k=3, hyperlines=[['a', 'b', 'c'],
....:                                   ['a', 'b', 'd' ]], certificate=True)
(True, frozenset({'a', 'b', 'c'}))
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R10()
>>> M.has_line_minor(Integer(4))
False
>>> M.has_line_minor(Integer(4), certificate=True)
(False, None)
>>> M.has_line_minor(Integer(3))
True
>>> M.has_line_minor(Integer(3), certificate=True)
(True, frozenset({'a', 'b', 'c', 'g'}))
>>> M.has_line_minor(k=Integer(3), hyperlines=[['a', 'b', 'c'],
...                                   ['a', 'b', 'd' ]])
True
>>> M.has_line_minor(k=Integer(3), hyperlines=[['a', 'b', 'c'],
...                                   ['a', 'b', 'd' ]], certificate=True)
(True, frozenset({'a', 'b', 'c'}))
```

is_binary(*randomized_tests=1*)

Decide if `self` is a binary matroid.

INPUT:

- `randomized_tests` – ignored

OUTPUT: boolean

ALGORITHM:

`self` is a regular matroid, so just return `True`.

See also

M.is_binary()

EXAMPLES:

```
sage: N = matroids.catalog.R10()
sage: N.is_binary()
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> N = matroids.catalog.R10()
>>> N.is_binary()
True
```

is_graphic()

Test if the regular matroid is graphic.

A matroid is *graphic* if there exists a graph whose edge set equals the groundset of the matroid, such that a subset of elements of the matroid is independent if and only if the corresponding subgraph is acyclic.

OUTPUT: boolean

EXAMPLES:

```
sage: M = matroids.catalog.R10()
sage: M.is_graphic()
False
sage: M = Matroid(graphs.CompleteGraph(5), regular=True)                    #␣
↪needs sage.graphs
sage: M.is_graphic()                                                        #␣
↪needs sage.graphs sage.rings.finite_rings
True
sage: M.dual().is_graphic()                                                 #␣
↪needs sage.graphs
False
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R10()
>>> M.is_graphic()
False
>>> M = Matroid(graphs.CompleteGraph(Integer(5)), regular=True)              ␣
↪     # needs sage.graphs
>>> M.is_graphic()                                                          #␣
↪needs sage.graphs sage.rings.finite_rings
True
>>> M.dual().is_graphic()                                                   #␣
↪needs sage.graphs
False
```

ALGORITHM:

In a recent paper, Geelen and Gerards [GG2012] reduced the problem to testing if a system of linear equations has a solution. While not the fastest method, and not necessarily constructive (in the presence of 2-separations especially), it is easy to implement.

is_regular()

Return if `self` is regular.

This is trivially `True` for a *RegularMatroid*.

EXAMPLES:

```
sage: M = matroids.catalog.R10()
sage: M.is_regular()
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R10()
>>> M.is_regular()
True
```

is_ternary(*randomized_tests=1*)

Decide if `self` is a ternary matroid.

INPUT:

- `randomized_tests` – ignored

OUTPUT: boolean

ALGORITHM:

`self` is a regular matroid, so just return `True`.

See also

M.is_ternary()

EXAMPLES:

```
sage: N = matroids.catalog.R10()
sage: N.is_ternary()
True
```

```
>>> from sage.all import *
>>> N = matroids.catalog.R10()
>>> N.is_ternary()
True
```

is_valid(*certificate=False*)

Test if the data obey the matroid axioms.

Since this is a regular matroid, this function tests if the representation matrix is *totally unimodular*, i.e. if all square submatrices have determinant in $\{-1, 0, 1\}$.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: boolean, or (boolean, dictionary)

EXAMPLES:

```
sage: # needs sage.graphs
sage: M = Matroid(Matrix(ZZ, [[1, 0, 0, 1, 1, 0, 1],
....:                         [0, 1, 0, 1, 0, 1, 1],
....:                         [0, 0, 1, 0, 1, 1, 1]]),
....:             regular=True, check=False)
```

(continues on next page)

(continued from previous page)

```
sage: M.is_valid(certificate=True)
(False, {'error': 'the representation matrix is not totally unimodular'})
sage: M = Matroid(graphs.PetersenGraph())
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = Matroid(Matrix(ZZ, [[Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(0), Integer(1)],
...                         [Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(1)],
...                         [Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1), Integer(1)]]),
...             regular=True, check=False)
>>> M.is_valid(certificate=True)
(False, {'error': 'the representation matrix is not totally unimodular'})
>>> M = Matroid(graphs.PetersenGraph())
>>> M.is_valid()
True
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: M = matroids.catalog.R10()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
sage: N = M.relabel({'g': 'x'})
sage: sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i', 'j', 'x']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R10()
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> N = M.relabel({'g': 'x'})
>>> sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i', 'j', 'x']
```

ternary_matroid(*randomized_tests=1*, *verify=True*)

Return a ternary matroid representing `self`.

INPUT:

- `randomized_tests` – ignored
- `verify` – ignored

OUTPUT: a ternary matroid

ALGORITHM:

`self` is a regular matroid, so just cast `self` to a TernaryMatroid.

See also

M.ternary_matroid()

EXAMPLES:

```
sage: N = matroids.catalog.R10()
sage: N.ternary_matroid()
Ternary matroid of rank 5 on 10 elements, type 4+
```

```
>>> from sage.all import *
>>> N = matroids.catalog.R10()
>>> N.ternary_matroid()
Ternary matroid of rank 5 on 10 elements, type 4+
```

class sage.matroids.linear_matroid.**TernaryMatroid**

Bases: *LinearMatroid*

Ternary matroids.

A ternary matroid is a linear matroid represented over the finite field with three elements. See *LinearMatroid* for a definition.

The simplest way to create a `TernaryMatroid` is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object E can be given as a groundset. If E is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a 'reduced' matrix B, to create the matroid induced by $A = [I\ \ B]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I\ \ B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.
- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) ignored
- `keep_initial_representation` – boolean (default: `True`); decides whether or not an internal copy of the input matrix should be preserved. This can help to see the structure of the matroid (e.g. in the case of graphic matroids), and makes it easier to look at extensions. However, the input matrix may have redundant rows, and sometimes it is desirable to store only a row-reduced copy.
- `basis` – (default: `None`) when provided, this is an ordered subset of `groundset`, such that the submatrix of `matrix` indexed by `basis` is an identity matrix. In this case, no row reduction takes place in the initialization phase.

OUTPUT: a `TernaryMatroid` instance based on the data above

Note

The recommended way to generate a ternary matroid is through the *Matroid()* function. This is usually the preferred way, since it automatically chooses between `TernaryMatroid` and other classes. For direct access to the `TernaryMatroid` constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

EXAMPLES:

```
sage: A = Matrix(GF(3), 2, 4, [[1, 0, 1, 1], [0, 1, 1, 1]])
sage: M = Matroid(A); M
Ternary matroid of rank 2 on 4 elements, type 0-
sage: sorted(M.groundset())
[0, 1, 2, 3]
sage: Matrix(M)
[1 0 1 1]
[0 1 1 1]
sage: M = Matroid(matrix=A, groundset='abcd')
sage: sorted(M.groundset())
['a', 'b', 'c', 'd']
sage: B = Matrix(GF(2), 2, 2, [[1, 1], [1, 1]])
sage: N = Matroid(ring=GF(3), reduced_matrix=B, groundset='abcd')                #␣
↪needs sage.rings.finite_rings
sage: M == N                                                                     #␣
↪needs sage.rings.finite_rings
True
```

```
>>> from sage.all import *
>>> A = Matrix(GF(Integer(3)), Integer(2), Integer(4), [[Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(1)]])
>>> M = Matroid(A); M
Ternary matroid of rank 2 on 4 elements, type 0-
>>> sorted(M.groundset())
[0, 1, 2, 3]
>>> Matrix(M)
[1 0 1 1]
[0 1 1 1]
>>> M = Matroid(matrix=A, groundset='abcd')
>>> sorted(M.groundset())
['a', 'b', 'c', 'd']
>>> B = Matrix(GF(Integer(2)), Integer(2), Integer(2), [[Integer(1), Integer(1)],␣
↪[Integer(1), Integer(1)]])
>>> N = Matroid(ring=GF(Integer(3)), reduced_matrix=B, groundset='abcd')          ␣
↪     # needs sage.rings.finite_rings
>>> M == N                                                                       #␣
↪needs sage.rings.finite_rings
True
```

base_ring()

Return the base ring of the matrix representing the matroid, in this case $\mathbf{F}_3$.

EXAMPLES:

```
sage: M = matroids.catalog.NonFano()
sage: M.base_ring()
Finite Field of size 3
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonFano()
>>> M.base_ring()
Finite Field of size 3
```

bicycle_dimension()

Return the bicycle dimension of the ternary matroid.

The bicycle dimension of a linear subspace V is $\dim(V \cap V^{\perp})$. The bicycle dimension of a matroid equals the bicycle dimension of its rowspace, and is a matroid invariant. See [Pen2012].

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.NonFano()
sage: M.bicycle_dimension()
0
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonFano()
>>> M.bicycle_dimension()
0
```

character()

Return the character of the ternary matroid.

For a linear subspace V over $GF(3)$ with orthogonal basis $q_1, \ldots, q_k$ the character equals the product of $|q_i|$ modulo 3, where the product ranges over the i such that $|q_i|$ is not divisible by 3. The character does not depend on the choice of the orthogonal basis. The character of a ternary matroid equals the character of its cocycle-space, and is an invariant for ternary matroids. See [Pen2012].

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.NonFano()
sage: M.character()
2
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonFano()
>>> M.character()
2
```

characteristic()

Return the characteristic of the base ring of the matrix representing the matroid, in this case 3.

EXAMPLES:

```
sage: M = matroids.catalog.NonFano()
sage: M.characteristic()
3
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonFano()
>>> M.characteristic()
3
```

is_ternary(*randomized_tests=1*)

Decide if `self` is a binary matroid.

INPUT:

- `randomized_tests` – ignored

OUTPUT: boolean

ALGORITHM:

`self` is a ternary matroid, so just return `True`.

See also

M.is_ternary()

EXAMPLES:

```
sage: N = matroids.catalog.NonFano()
sage: N.is_ternary()
True
```

```
>>> from sage.all import *
>>> N = matroids.catalog.NonFano()
>>> N.is_ternary()
True
```

is_valid(*certificate=False*)

Test if the data obey the matroid axioms.

Since this is a linear matroid over the field $\mathbf{F}_3$, this is always the case.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: `True`, or `(True, {})`

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(3), [[]]))
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(3)), [[]]))
>>> M.is_valid()
True
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: M = matroids.catalog.NonFano()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
sage: N = M.relabel({'g': 'x'})
sage: sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'x']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.NonFano()
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> N = M.relabel({'g': 'x'})
>>> sorted(N.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'x']
```

ternary_matroid(*randomized_tests=1*, *verify=True*)

Return a ternary matroid representing `self`.

INPUT:

- `randomized_tests` – ignored
- `verify` – ignored

OUTPUT: a binary matroid

ALGORITHM:

`self` is a ternary matroid, so just return `self`.

See also

M.ternary_matroid()

EXAMPLES:

```
sage: N = matroids.catalog.NonFano()
sage: N.ternary_matroid() is N
True
```

```
>>> from sage.all import *
>>> N = matroids.catalog.NonFano()
>>> N.ternary_matroid() is N
True
```

3.8 Rank function matroids

The easiest way to define arbitrary matroids in Sage might be through the class `RankMatroid`. All that is required is a groundset and a function that computes the rank for each given subset.

Of course, since the rank function is used as black box, matroids so defined cannot take advantage of any extra structure your class might have, and rely on default implementations. Besides this, matroids in this class can't be saved.

3.8.1 Constructions

Any function can be used, but no checks are performed, so be careful.

EXAMPLES:

```
sage: def f(X):
....:     return min(len(X), 3)
sage: M = Matroid(groundset=range(6), rank_function=f)
sage: M.is_valid()
True
sage: M.is_isomorphic(matroids.Uniform(3, 6))
True

sage: def g(X):
....:     if len(X) >= 3:
....:         return 1
....:     else:
....:         return 0
sage: N = Matroid(groundset='abc', rank_function=g)
sage: N.is_valid()
False
```

```
>>> from sage.all import *
>>> def f(X):
...     return min(len(X), Integer(3))
>>> M = Matroid(groundset=range(Integer(6)), rank_function=f)
>>> M.is_valid()
True
>>> M.is_isomorphic(matroids.Uniform(Integer(3), Integer(6)))
True

>>> def g(X):
...     if len(X) >= Integer(3):
...         return Integer(1)
```

(continues on next page)

(continued from previous page)

```
...     else:
...         return Integer(0)
>>> N = Matroid(groundset='abc', rank_function=g)
>>> N.is_valid()
False
```

See *below* for more. It is recommended to use the *Matroid()* function for easy construction of a `RankMatroid`. For direct access to the `RankMatroid` constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

3.8.2 Methods

class sage.matroids.rank_matroid.**RankMatroid**(*groundset*, *rank_function*)

Bases: *Matroid*

Matroid specified by its rank function.

INPUT:

- `groundset` – the groundset of a matroid
- `rank_function` – a function mapping subsets of `groundset` to nonnegative integers

OUTPUT: a matroid on `groundset` whose rank function equals `rank_function`

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: def f(X):
....:     return min(len(X), 3)
sage: M = RankMatroid(groundset=range(6), rank_function=f)
sage: M.is_valid()
True
sage: M.is_isomorphic(matroids.Uniform(3, 6))
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> def f(X):
...     return min(len(X), Integer(3))
>>> M = RankMatroid(groundset=range(Integer(6)), rank_function=f)
>>> M.is_valid()
True
>>> M.is_isomorphic(matroids.Uniform(Integer(3), Integer(6)))
True
```

groundset()

Return the groundset of `self`.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = RankMatroid(range(6),
....:                 rank_function=matroids.Uniform(3, 6).rank)
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = RankMatroid(range(Integer(6)),
...                 rank_function=matroids.Uniform(Integer(3), Integer(6)).
↪rank)
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
```

3.9 Transversal matroids

A transversal matroid arises from a groundset E and a collection A of sets over the groundset. This can be modeled as a bipartite graph B, where the vertices on the left are groundset elements, the vertices on the right are the sets, and edges represent containment. Then a set X from the groundset is independent if and only if X has a matching in B.

To construct a transversal matroid, first import *TransversalMatroid* from *sage.matroids.transversal_matroid*. The input should be a set system, formatted as an iterable of iterables:

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: sets = [[3, 4, 5, 6, 7, 8]] * 3
sage: M = TransversalMatroid(sets); M
Transversal matroid of rank 3 on 6 elements, with 3 sets
sage: M.groundset()
frozenset({3, 4, 5, 6, 7, 8})
sage: M.is_isomorphic(matroids.Uniform(3, 6))
True
sage: M = TransversalMatroid([[0, 1], [1, 2, 3], [3, 4, 5]],
....:                        set_labels=['1', '2', '3'])
sage: M.graph().vertices()
['1', '2', '3', 0, 1, 2, 3, 4, 5]
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> sets = [[Integer(3), Integer(4), Integer(5), Integer(6), Integer(7), Integer(8)]]␣
↪* Integer(3)
>>> M = TransversalMatroid(sets); M
Transversal matroid of rank 3 on 6 elements, with 3 sets
>>> M.groundset()
frozenset({3, 4, 5, 6, 7, 8})
>>> M.is_isomorphic(matroids.Uniform(Integer(3), Integer(6)))
True
>>> M = TransversalMatroid([[Integer(0), Integer(1)], [Integer(1), Integer(2),␣
↪Integer(3)], [Integer(3), Integer(4), Integer(5)]],
```

(continues on next page)

(continued from previous page)

```
...                          set_labels=['1', '2', '3'])
>>> M.graph().vertices()
['1', '2', '3', 0, 1, 2, 3, 4, 5]
```

AUTHORS:

- Zachary Gershkoff (2017-08-07): initial version

REFERENCES:

- [Bon2017]

class sage.matroids.transversal_matroid.**TransversalMatroid**

Bases: *BasisExchangeMatroid*

The Transversal Matroid class.

INPUT:

- `sets` – iterable of iterables of elements
- `groundset` – (optional) iterable containing names of groundset elements
- `set_labels` – (optional) list of labels in 1-1 correspondence with the iterables in `sets`
- `matching` – (optional) dictionary specifying a maximal matching between elements and set labels

OUTPUT:

An instance of `TransversalMatroid`. The sets specified in `sets` define the matroid. If `matching` is not specified, the constructor will determine a matching to use for basis exchange.

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: sets = [[0, 1, 2, 3]] * 3
sage: M = TransversalMatroid(sets)
sage: M.full_rank()
3
sage: M.bases_count()
4
sage: sum(1 for b in M.bases())
4
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> sets = [[Integer(0), Integer(1), Integer(2), Integer(3)]] * Integer(3)
>>> M = TransversalMatroid(sets)
>>> M.full_rank()
3
>>> M.bases_count()
4
>>> sum(Integer(1) for b in M.bases())
4
```

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: M = TransversalMatroid(sets=[['a', 'c']], groundset=['a', 'c', 'd'])
sage: M.loops()
```

(continues on next page)

(continued from previous page)

```
frozenset({'d'})
sage: M.full_rank()
1
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> M = TransversalMatroid(sets=[['a', 'c']], groundset=['a', 'c', 'd'])
>>> M.loops()
frozenset({'d'})
>>> M.full_rank()
1
```

graph()

Return a bipartite graph representing the transversal matroid.

A transversal matroid can be represented as a set system, or as a bipartite graph with one color class corresponding to the groundset and the other to the sets of the set system. This method returns that bipartite graph.

OUTPUT: Graph

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: edgedict = {5: [0, 1, 2, 3], 6: [1, 2], 7: [1, 3, 4]}
sage: B = BipartiteGraph(edgedict)
sage: M = TransversalMatroid(edgedict.values(), set_labels=edgedict.keys())
sage: M.graph() == B
True
sage: M2 = TransversalMatroid(edgedict.values())
sage: B2 = M2.graph()
sage: B2 == B
False
sage: B2.is_isomorphic(B)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> edgedict = {Integer(5): [Integer(0), Integer(1), Integer(2), Integer(3)],␣
↪Integer(6): [Integer(1), Integer(2)], Integer(7): [Integer(1), Integer(3),␣
↪Integer(4)]}
>>> B = BipartiteGraph(edgedict)
>>> M = TransversalMatroid(edgedict.values(), set_labels=edgedict.keys())
>>> M.graph() == B
True
>>> M2 = TransversalMatroid(edgedict.values())
>>> B2 = M2.graph()
>>> B2 == B
False
>>> B2.is_isomorphic(B)
True
```

is_valid(*certificate=False*)

Test whether the matching in memory is a valid maximal matching.

The data for a transversal matroid is a set system, which is always valid, but it is possible for a user to provide invalid input with the `matching` parameter. This checks that the matching provided is indeed a matching, fits in the set system, and is maximal.

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import *
sage: sets = [[0, 1, 2, 3], [1, 2], [1, 3, 4]]
sage: set_labels = [5, 6, 7]
sage: M = TransversalMatroid(sets, set_labels=set_labels)
sage: M.is_valid()
True
sage: m = {0: 5, 1: 5, 3: 7}  # not a matching
sage: TransversalMatroid(sets, set_labels=set_labels, matching=m).is_valid()
False
sage: m = {2: 6, 3: 7}  # not maximal
sage: TransversalMatroid(sets, set_labels=set_labels, matching=m).is_valid()
False
sage: m = {0: 6, 1: 5, 3: 7}  # not in the set system
sage: TransversalMatroid(sets, set_labels=set_labels, matching=m).is_valid()
False
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import *
>>> sets = [[Integer(0), Integer(1), Integer(2), Integer(3)], [Integer(1),␣
↪Integer(2)], [Integer(1), Integer(3), Integer(4)]]
>>> set_labels = [Integer(5), Integer(6), Integer(7)]
>>> M = TransversalMatroid(sets, set_labels=set_labels)
>>> M.is_valid()
True
>>> m = {Integer(0): Integer(5), Integer(1): Integer(5), Integer(3):␣
↪Integer(7)}  # not a matching
>>> TransversalMatroid(sets, set_labels=set_labels, matching=m).is_valid()
False
>>> m = {Integer(2): Integer(6), Integer(3): Integer(7)}  # not maximal
>>> TransversalMatroid(sets, set_labels=set_labels, matching=m).is_valid()
False
>>> m = {Integer(0): Integer(6), Integer(1): Integer(5), Integer(3):␣
↪Integer(7)}  # not in the set system
>>> TransversalMatroid(sets, set_labels=set_labels, matching=m).is_valid()
False
```

reduce_presentation()

Return an equal transversal matroid where the number of sets equals the rank.

Every transversal matroid M has a presentation with $r(M)$ sets, and if M has no coloops, then every presentation has $r(M)$ nonempty sets. This method discards extra sets if M has coloops.

OUTPUT: *TransversalMatroid* with a reduced presentation

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: sets = [[0, 1], [2], [2]]
sage: M = TransversalMatroid(sets); M
```

(continues on next page)

(continued from previous page)

```
Transversal matroid of rank 2 on 3 elements, with 3 sets
sage: N = M.reduce_presentation(); N
Transversal matroid of rank 2 on 3 elements, with 2 sets
sage: N.equals(M)
True
sage: N == M
False
sage: sets = [[0, 1], [], [], [2]]
sage: M1 = TransversalMatroid(sets); M1
Transversal matroid of rank 2 on 3 elements, with 4 sets
sage: M1.reduce_presentation()
Transversal matroid of rank 2 on 3 elements, with 2 sets
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> sets = [[Integer(0), Integer(1)], [Integer(2)], [Integer(2)]]
>>> M = TransversalMatroid(sets); M
Transversal matroid of rank 2 on 3 elements, with 3 sets
>>> N = M.reduce_presentation(); N
Transversal matroid of rank 2 on 3 elements, with 2 sets
>>> N.equals(M)
True
>>> N == M
False
>>> sets = [[Integer(0), Integer(1)], [], [], [Integer(2)]]
>>> M1 = TransversalMatroid(sets); M1
Transversal matroid of rank 2 on 3 elements, with 4 sets
>>> M1.reduce_presentation()
Transversal matroid of rank 2 on 3 elements, with 2 sets
```

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: sets = [[0, 1, 2, 3]] * 3
sage: M = TransversalMatroid(sets)
sage: N = M.reduce_presentation()
sage: M == N
True
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> sets = [[Integer(0), Integer(1), Integer(2), Integer(3)]] * Integer(3)
>>> M = TransversalMatroid(sets)
>>> N = M.reduce_presentation()
>>> M == N
True
```

set_labels()

Return the labels used for the transversal sets.

This method will return a list of the labels used of the non-ground set vertices of the bipartite graph used to represent the transversal matroid. This method does not set anything.

OUTPUT: list

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: M = TransversalMatroid([[0, 1], [1, 2, 3], [3, 4, 7]])
sage: M.set_labels()
['s0', 's1', 's2']
sage: M.graph().vertices()
['s0', 's1', 's2', 0, 1, 2, 3, 4, 7]
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> M = TransversalMatroid([[Integer(0), Integer(1)], [Integer(1), Integer(2),
↪ Integer(3)], [Integer(3), Integer(4), Integer(7)]])
>>> M.set_labels()
['s0', 's1', 's2']
>>> M.graph().vertices()
['s0', 's1', 's2', 0, 1, 2, 3, 4, 7]
```

sets()

Return the sets of `self`.

A transversal matroid can be viewed as a groundset with a collection from its powerset. This is represented as a bipartite graph, where an edge represents containment.

OUTPUT: list of lists that correspond to the sets of the transversal matroid

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: sets = [[0, 1, 2, 3], [1, 2], [3, 4]]
sage: set_labels = [5, 6, 7]
sage: M = TransversalMatroid(sets, set_labels=set_labels)
sage: sorted(M.sets()) == sorted(sets)
True
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> sets = [[Integer(0), Integer(1), Integer(2), Integer(3)], [Integer(1),␣
↪Integer(2)], [Integer(3), Integer(4)]]
>>> set_labels = [Integer(5), Integer(6), Integer(7)]
>>> M = TransversalMatroid(sets, set_labels=set_labels)
>>> sorted(M.sets()) == sorted(sets)
True
```

transversal_extension(*element=None*, *newset=False*, *sets=None*)

Return a *TransversalMatroid* extended by an element.

This will modify the presentation of the transversal matroid by adding a new element, and placing this element in the specified sets. It is also possible to use this method to create a new set that will have the new element as its only member, making it a coloop.

INPUT:

- `element` – (optional) the name for the new element
- `newset` – (optional) if specified, the element will be given its own set

- `sets` – iterable of labels (default: `None`) representing the sets in the current presentation that the new element will belong to

OUTPUT:

A `TransversalMatroid` with a groundset element added to specified sets. Note that the `newset` option will make the new element a coloop. If `newset == True`, a name will be generated; otherwise the value of `newset` will be used.

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: M = TransversalMatroid([['a', 'c']], groundset=['a', 'c'], set_labels=[
↪'b'])
sage: M1 = M.transversal_extension(element='d', newset='e')
sage: M2 = M.transversal_extension(element='d', newset=True)
sage: M1.coloops()
frozenset({'d'})
sage: True in M2.graph().vertices()
False
sage: M1.is_isomorphic(M2)
True
sage: M3 = M.transversal_extension('d', sets=['b'])
sage: M3.is_isomorphic(matroids.Uniform(1, 3))
True
sage: M4 = M.transversal_extension('d', sets=['a'])
Traceback (most recent call last):
...
ValueError: sets do not match presentation
sage: M4 = M.transversal_extension('a', sets=['b'])
Traceback (most recent call last):
...
ValueError: cannot extend by element already in groundset
sage: M2.transversal_extension(newset='b')
Traceback (most recent call last):
...
ValueError: newset is already a vertex in the presentation
sage: M5 = M1.transversal_extension()
sage: len(M5.loops())
1
sage: M2.transversal_extension(element='b')
Transversal matroid of rank 2 on 4 elements, with 2 sets
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> M = TransversalMatroid([['a', 'c']], groundset=['a', 'c'], set_labels=['b
↪'])
>>> M1 = M.transversal_extension(element='d', newset='e')
>>> M2 = M.transversal_extension(element='d', newset=True)
>>> M1.coloops()
frozenset({'d'})
>>> True in M2.graph().vertices()
False
>>> M1.is_isomorphic(M2)
True
```

(continues on next page)

(continued from previous page)

```
>>> M3 = M.transversal_extension('d', sets=['b'])
>>> M3.is_isomorphic(matroids.Uniform(Integer(1), Integer(3)))
True
>>> M4 = M.transversal_extension('d', sets=['a'])
Traceback (most recent call last):
...
ValueError: sets do not match presentation
>>> M4 = M.transversal_extension('a', sets=['b'])
Traceback (most recent call last):
...
ValueError: cannot extend by element already in groundset
>>> M2.transversal_extension(newset='b')
Traceback (most recent call last):
...
ValueError: newset is already a vertex in the presentation
>>> M5 = M1.transversal_extension()
>>> len(M5.loops())
1
>>> M2.transversal_extension(element='b')
Transversal matroid of rank 2 on 4 elements, with 2 sets
```

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: sets = [[0, 1, 2, 3], [1, 2], [1, 3, 4]]
sage: M = TransversalMatroid(sets, groundset=range(5), set_labels=[5, 6, 7])
sage: N = M.delete(2)
sage: M1 = N.transversal_extension(element=2, sets=[5, 6])
sage: M1 == M
True
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> sets = [[Integer(0), Integer(1), Integer(2), Integer(3)], [Integer(1),␣
↪Integer(2)], [Integer(1), Integer(3), Integer(4)]]
>>> M = TransversalMatroid(sets, groundset=range(Integer(5)), set_
↪labels=[Integer(5), Integer(6), Integer(7)])
>>> N = M.delete(Integer(2))
>>> M1 = N.transversal_extension(element=Integer(2), sets=[Integer(5),␣
↪Integer(6)])
>>> M1 == M
True
```

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: sets = [[0, 1, 2, 3]] * 3
sage: M = TransversalMatroid(sets, set_labels=[4, 5, 6])
sage: N = M.transversal_extension(element='a', newset=True, sets=[4])
sage: N.graph().degree('a')
2
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> sets = [[Integer(0), Integer(1), Integer(2), Integer(3)]] * Integer(3)
```

(continues on next page)

(continued from previous page)

```
>>> M = TransversalMatroid(sets, set_labels=[Integer(4), Integer(5),␣
↪Integer(6)])
>>> N = M.transversal_extension(element='a', newset=True, sets=[Integer(4)])
>>> N.graph().degree('a')
2
```

transversal_extensions(*element=None*, *sets=None*)

Return an iterator of extensions based on the transversal presentation.

This method will take an extension by adding an element to every possible sub-collection of the collection of desired sets. No checking is done for equal matroids. It is advised to make sure the presentation has as few sets as possible by using *reduce_presentation()*

INPUT:

- `element` – (optional) the name of the new element
- `sets` – (optional) list containing names of sets in the matroid's presentation

OUTPUT: iterator over instances of *TransversalMatroid*

If `sets` is not specified, every set will be used.

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import TransversalMatroid
sage: sets = [[3, 4, 5, 6]] * 3
sage: M = TransversalMatroid(sets, set_labels=[0, 1, 2])
sage: len([N for N in M.transversal_extensions()])
8
sage: len([N for N in M.transversal_extensions(sets=[0, 1])])
4
sage: set(sorted([N.groundset() for N in M.transversal_
↪extensions(element=7)]))
{frozenset({3, 4, 5, 6, 7})}
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import TransversalMatroid
>>> sets = [[Integer(3), Integer(4), Integer(5), Integer(6)]] * Integer(3)
>>> M = TransversalMatroid(sets, set_labels=[Integer(0), Integer(1),␣
↪Integer(2)])
>>> len([N for N in M.transversal_extensions()])
8
>>> len([N for N in M.transversal_extensions(sets=[Integer(0), Integer(1)])])
4
>>> set(sorted([N.groundset() for N in M.transversal_
↪extensions(element=Integer(7))]))
{frozenset({3, 4, 5, 6, 7})}
```

CHAPTER

FOUR

CHOW RINGS OF MATROIDS

4.1 Chow ring ideals of matroids

AUTHORS:

- Shriya M

class sage.matroids.chow_ring_ideal.**AugmentedChowRingIdeal_atom_free**(M, R)

Bases: *ChowRingIdeal*

The augmented Chow ring ideal for a matroid M over ring R in the atom-free presentation.

The augmented Chow ring ideal in the atom-free presentation for a matroid M is defined as the ideal $I_{af}(M)$ of the polynomial ring:

$$R[x_{F_1}, \ldots, x_{F_k}],$$

where $F_1, \ldots, F_k$ are the non-empty flats of M and $I_{af}(M)$ is the ideal generated by

- all quadratic monomials $x_F x_{F'}$ for all incomparable elements F and F' in the lattice of flats,
- for all flats F and $i \in E \setminus F$

$$x_F \sum_{i \in F'} x_{F'}$$

- and for all $i \in E$

$$\sum_{i \in F'} (x_{F'})^2.$$

REFERENCES:

- [MM2022]

INPUT:

- `M` – matroid
- `R` – commutative ring

EXAMPLES:

Augmented Chow ring ideal of Wheel matroid of rank 3:

```
sage: ch = matroids.Wheel(3).chow_ring(QQ, True, 'atom-free')
sage: ch.defining_ideal()
Augmented Chow ring ideal of Wheel(3): Regular matroid of rank 3 on 6
elements with 16 bases in the atom-free presentation
```

```
>>> from sage.all import *
>>> ch = matroids.Wheel(Integer(3)).chow_ring(QQ, True, 'atom-free')
>>> ch.defining_ideal()
Augmented Chow ring ideal of Wheel(3): Regular matroid of rank 3 on 6
elements with 16 bases in the atom-free presentation
```

groebner_basis(*algorithm=''*, **args*, ***kwargs*)

Return the Groebner basis of `self`.

EXAMPLES:

```
sage: M1 = matroids.Uniform(3, 6)
sage: ch = M1.chow_ring(QQ, True, 'atom-free')
sage: ch.defining_ideal().groebner_basis(algorithm='')
Polynomial Sequence with 253 Polynomials in 22 Variables
sage: ch.defining_ideal().groebner_basis(algorithm='').is_groebner()
True
sage: ch.defining_ideal().hilbert_series() == ch.defining_ideal().gens().
↪ideal().hilbert_series()
True
```

```
>>> from sage.all import *
>>> M1 = matroids.Uniform(Integer(3), Integer(6))
>>> ch = M1.chow_ring(QQ, True, 'atom-free')
>>> ch.defining_ideal().groebner_basis(algorithm='')
Polynomial Sequence with 253 Polynomials in 22 Variables
>>> ch.defining_ideal().groebner_basis(algorithm='').is_groebner()
True
>>> ch.defining_ideal().hilbert_series() == ch.defining_ideal().gens().
↪ideal().hilbert_series()
True
```

normal_basis(*algorithm=''*, **args*, ***kwargs*)

Return the monomial basis of the quotient ring of this ideal.

EXAMPLES:

```
sage: ch = Matroid(graphs.CycleGraph(3)).chow_ring(QQ, True, 'atom-free')
sage: I = ch.defining_ideal()
sage: I.normal_basis()
[1, A0, A1, A2, A3, A3^2]
sage: set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
sage: ch = matroids.Wheel(3).chow_ring(QQ, True, 'atom-free')
sage: I = ch.defining_ideal()
sage: I.normal_basis()
Polynomial Sequence with 30 Polynomials in 14 Variables
sage: set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
```

```
>>> from sage.all import *
>>> ch = Matroid(graphs.CycleGraph(Integer(3))).chow_ring(QQ, True, 'atom-free
↪')
```

(continues on next page)

(continued from previous page)

```
>>> I = ch.defining_ideal()
>>> I.normal_basis()
[1, A0, A1, A2, A3, A3^2]
>>> set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
>>> ch = matroids.Wheel(Integer(3)).chow_ring(QQ, True, 'atom-free')
>>> I = ch.defining_ideal()
>>> I.normal_basis()
Polynomial Sequence with 30 Polynomials in 14 Variables
>>> set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
```

class sage.matroids.chow_ring_ideal.**AugmentedChowRingIdeal_fy**(M, R)

Bases: *ChowRingIdeal*

The augmented Chow ring ideal of matroid M over ring R in the Feitchner-Yuzvinsky presentation.

The augmented Chow ring ideal for a matroid M is defined as the ideal $(I_M + J_M)$ of the following polynomial ring

$$R[y_{e_1}, \ldots, y_{e_n}, x_{F_1}, \ldots, x_{F_k}],$$

where

- $F_1, \ldots, F_k$ are the proper flats of M,
- $e_1, \ldots, e_n$ are n elements of groundset of M,
- J_M is the ideal generated by all quadratic monomials $x_F x_{F'}$, where F and F' are incomparable elements in the lattice of flats and $y_i x_F$ for all flats F and $i \in E \setminus F$ and
- I_M is the ideal generated by all linear forms

$$y_i - \sum_{i \notin F} x_F$$

for all $i \in E$.

The augmented Chow ring ideal in the Feitchner-Yuzvinsky presentation for a simple matroid M is defined as the ideal $I_{FY}(M)$ of the following polynomial ring

$$R[y_{e_1}, \ldots, y_{e_n}, y_{F_1 \cup e}, \ldots, y_{F_k \cup e}],$$

where $F_1, \ldots, F_k$ are the flats of M, $e_1, \ldots, e_n$ are n elements of groundset of M, and $I_{FY}(M)$ is the ideal generated by

- all quadratic monomials $y_{F \cup e} y_{F' \cup e}$, for incomparable elements F and F' in the lattice of flats,
- $y_i y_{F \cup e}$ for all flats F and all $i \in E \setminus F$
- for all $i \in E$

$$y_i + \sum_{i \in F} y_{F \cup e}$$

- and

$$\sum_F y_{F \cup e}.$$

Setting $x_F = y_{F \cup e}$ and using the last linear form to eliminate x_E recovers the usual presentation of augmented Chow ring of M.

REFERENCES:

- [MM2022]

INPUT:

- `M` – matroid
- `R` – commutative ring

EXAMPLES:

Augmented Chow ring ideal of Wheel matroid of rank 3:

```
sage: ch = matroids.Wheel(3).chow_ring(QQ, True, 'fy')
sage: ch.defining_ideal()
Augmented Chow ring ideal of Wheel(3): Regular matroid of rank 3 on 6
elements with 16 bases of Feitchner-Yuzvinsky presentation
```

```
>>> from sage.all import *
>>> ch = matroids.Wheel(Integer(3)).chow_ring(QQ, True, 'fy')
>>> ch.defining_ideal()
Augmented Chow ring ideal of Wheel(3): Regular matroid of rank 3 on 6
elements with 16 bases of Feitchner-Yuzvinsky presentation
```

groebner_basis(*algorithm=''*, **args*, ***kwargs*)

Return the Groebner basis of `self`.

EXAMPLES:

```
sage: ch = matroids.catalog.NonFano().chow_ring(QQ, True, 'fy')
sage: ch.defining_ideal().groebner_basis(algorithm='')
Polynomial Sequence with 178 Polynomials in 25 Variables
sage: ch.defining_ideal().groebner_basis(algorithm='').is_groebner()
True
sage: ch.defining_ideal().hilbert_series() == ch.defining_ideal().gens().
↪ideal().hilbert_series()
True
```

```
>>> from sage.all import *
>>> ch = matroids.catalog.NonFano().chow_ring(QQ, True, 'fy')
>>> ch.defining_ideal().groebner_basis(algorithm='')
Polynomial Sequence with 178 Polynomials in 25 Variables
>>> ch.defining_ideal().groebner_basis(algorithm='').is_groebner()
True
>>> ch.defining_ideal().hilbert_series() == ch.defining_ideal().gens().
↪ideal().hilbert_series()
True
```

normal_basis(*algorithm=''*, **args*, ***kwargs*)

Return the monomial basis of the quotient ring of this ideal.

EXAMPLES:

```
sage: ch = matroids.Uniform(2, 5).chow_ring(QQ, True, 'fy')
sage: I = ch.defining_ideal()
sage: I.normal_basis()
[1, B0, B1, B2, B3, B4, B01234, B01234^2]
sage: set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
sage: ch = matroids.catalog.Fano().chow_ring(QQ, True, 'fy')
sage: I = ch.defining_ideal()
sage: I.normal_basis()
Polynomial Sequence with 32 Polynomials in 15 Variables
sage: set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(2), Integer(5)).chow_ring(QQ, True, 'fy')
>>> I = ch.defining_ideal()
>>> I.normal_basis()
[1, B0, B1, B2, B3, B4, B01234, B01234^2]
>>> set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
>>> ch = matroids.catalog.Fano().chow_ring(QQ, True, 'fy')
>>> I = ch.defining_ideal()
>>> I.normal_basis()
Polynomial Sequence with 32 Polynomials in 15 Variables
>>> set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
```

class sage.matroids.chow_ring_ideal.**ChowRingIdeal**(*ring*, *gens*, *coerce=True*)

Bases: MPolynomialIdeal

flats_to_generator_dict()

Return the corresponding generators of flats/groundset elements of Chow ring ideal.

EXAMPLES:

```
sage: ch = matroids.Uniform(4, 6).chow_ring(QQ, True, 'atom-free')
sage: ch.defining_ideal().flats_to_generator_dict()
{frozenset({0}): A0, frozenset({1}): A1, frozenset({2}): A2,
 frozenset({3}): A3, frozenset({4}): A4, frozenset({5}): A5,
 frozenset({0, 1}): A01, frozenset({0, 2}): A02,
 frozenset({0, 3}): A03, frozenset({0, 4}): A04,
 frozenset({0, 5}): A05, frozenset({1, 2}): A12,
 frozenset({1, 3}): A13, frozenset({1, 4}): A14,
 frozenset({1, 5}): A15, frozenset({2, 3}): A23,
 frozenset({2, 4}): A24, frozenset({2, 5}): A25,
 frozenset({3, 4}): A34, frozenset({3, 5}): A35,
 frozenset({4, 5}): A45, frozenset({0, 1, 2}): A012,
 frozenset({0, 1, 3}): A013, frozenset({0, 1, 4}): A014,
 frozenset({0, 1, 5}): A015, frozenset({0, 2, 3}): A023,
 frozenset({0, 2, 4}): A024, frozenset({0, 2, 5}): A025,
 frozenset({0, 3, 4}): A034, frozenset({0, 3, 5}): A035,
 frozenset({0, 4, 5}): A045, frozenset({1, 2, 3}): A123,
 frozenset({1, 2, 4}): A124, frozenset({1, 2, 5}): A125,
```

(continues on next page)

(continued from previous page)

```
 frozenset({1, 3, 4}): A134, frozenset({1, 3, 5}): A135,
 frozenset({1, 4, 5}): A145, frozenset({2, 3, 4}): A234,
 frozenset({2, 3, 5}): A235, frozenset({2, 4, 5}): A245,
 frozenset({3, 4, 5}): A345,
 frozenset({0, 1, 2, 3, 4, 5}): A012345}
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(4), Integer(6)).chow_ring(QQ, True, 'atom-
↪free')
>>> ch.defining_ideal().flats_to_generator_dict()
{frozenset({0}): A0, frozenset({1}): A1, frozenset({2}): A2,
 frozenset({3}): A3, frozenset({4}): A4, frozenset({5}): A5,
 frozenset({0, 1}): A01, frozenset({0, 2}): A02,
 frozenset({0, 3}): A03, frozenset({0, 4}): A04,
 frozenset({0, 5}): A05, frozenset({1, 2}): A12,
 frozenset({1, 3}): A13, frozenset({1, 4}): A14,
 frozenset({1, 5}): A15, frozenset({2, 3}): A23,
 frozenset({2, 4}): A24, frozenset({2, 5}): A25,
 frozenset({3, 4}): A34, frozenset({3, 5}): A35,
 frozenset({4, 5}): A45, frozenset({0, 1, 2}): A012,
 frozenset({0, 1, 3}): A013, frozenset({0, 1, 4}): A014,
 frozenset({0, 1, 5}): A015, frozenset({0, 2, 3}): A023,
 frozenset({0, 2, 4}): A024, frozenset({0, 2, 5}): A025,
 frozenset({0, 3, 4}): A034, frozenset({0, 3, 5}): A035,
 frozenset({0, 4, 5}): A045, frozenset({1, 2, 3}): A123,
 frozenset({1, 2, 4}): A124, frozenset({1, 2, 5}): A125,
 frozenset({1, 3, 4}): A134, frozenset({1, 3, 5}): A135,
 frozenset({1, 4, 5}): A145, frozenset({2, 3, 4}): A234,
 frozenset({2, 3, 5}): A235, frozenset({2, 4, 5}): A245,
 frozenset({3, 4, 5}): A345,
 frozenset({0, 1, 2, 3, 4, 5}): A012345}
```

matroid()

Return the matroid of the given Chow ring ideal.

EXAMPLES:

```
sage: ch = matroids.Uniform(3, 6).chow_ring(QQ, False)
sage: ch.defining_ideal().matroid()
U(3, 6): Matroid of rank 3 on 6 elements with circuit-closures
{3: {{0, 1, 2, 3, 4, 5}}}
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(3), Integer(6)).chow_ring(QQ, False)
>>> ch.defining_ideal().matroid()
U(3, 6): Matroid of rank 3 on 6 elements with circuit-closures
{3: {{0, 1, 2, 3, 4, 5}}}
```

class sage.matroids.chow_ring_ideal.**ChowRingIdeal_nonaug**(M, R)

Bases: *ChowRingIdeal*

The Chow ring ideal of a matroid M.

The *Chow ring ideal* for a matroid M is defined as the ideal $(I_M + J_M)$ of the polynomial ring

$$R[x_{F_1}, \ldots, x_{F_k}],$$

where

- $F_1, \ldots, F_k$ are the non-empty flats of M,
- I_M is the Stanley-Reisner ideal, i.e., it is generated by products $x_{F_1}, \ldots, x_{F_t}$ for subsets $\{F_1, \ldots, F_t\}$ of flats that are not chains, and
- J_M is the ideal generated by all linear forms

$$\sum_{a \in F} x_F$$

for all atoms a in the lattice of flats.

INPUT:

- `M` – matroid
- `R` – commutative ring

REFERENCES:

- [ANR2023]

EXAMPLES:

Chow ring ideal of uniform matroid of rank 3 on 6 elements:

```
sage: ch = matroids.Uniform(3, 6).chow_ring(QQ, False)
sage: ch.defining_ideal()
Chow ring ideal of U(3, 6): Matroid of rank 3 on 6 elements with
 circuit-closures {3: {{0, 1, 2, 3, 4, 5}}} - non augmented
sage: ch = matroids.catalog.Fano().chow_ring(QQ, False)
sage: ch.defining_ideal()
Chow ring ideal of Fano: Binary matroid of rank 3 on 7 elements,
 type (3, 0) - non augmented
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(3), Integer(6)).chow_ring(QQ, False)
>>> ch.defining_ideal()
Chow ring ideal of U(3, 6): Matroid of rank 3 on 6 elements with
 circuit-closures {3: {{0, 1, 2, 3, 4, 5}}} - non augmented
>>> ch = matroids.catalog.Fano().chow_ring(QQ, False)
>>> ch.defining_ideal()
Chow ring ideal of Fano: Binary matroid of rank 3 on 7 elements,
 type (3, 0) - non augmented
```

groebner_basis(*algorithm=''*, **args*, ***kwargs*)

Return a Groebner basis of `self`.

EXAMPLES:

```
sage: ch = Matroid(groundset='abc', bases=['ab', 'ac']).chow_ring(QQ, False)
sage: ch.defining_ideal().groebner_basis()
[Aa*Abc, Aa + Aabc, Abc + Aabc, Aa*Aabc, Abc*Aabc, Aabc^2]
```

(continues on next page)

(continued from previous page)

```
sage: ch.defining_ideal().groebner_basis().is_groebner()
True
sage: ch.defining_ideal().hilbert_series() == ch.defining_ideal().gens().
↪ideal().hilbert_series()
True
```

```
>>> from sage.all import *
>>> ch = Matroid(groundset='abc', bases=['ab', 'ac']).chow_ring(QQ, False)
>>> ch.defining_ideal().groebner_basis()
[Aa*Abc, Aa + Aabc, Abc + Aabc, Aa*Aabc, Abc*Aabc, Aabc^2]
>>> ch.defining_ideal().groebner_basis().is_groebner()
True
>>> ch.defining_ideal().hilbert_series() == ch.defining_ideal().gens().
↪ideal().hilbert_series()
True
```

Another example would be the Groebner basis of the Chow ring ideal of the matroid of the length 3 cycle graph:

```
sage: ch = Matroid(graphs.CycleGraph(3)).chow_ring(QQ, False)
sage: ch.defining_ideal().groebner_basis()
[A0*A1, A0*A2, A1*A2, A0 + A3, A1 + A3, A2 + A3, A0*A3, A1*A3, A2*A3, A3^2]
sage: ch.defining_ideal().groebner_basis().is_groebner()
True
sage: ch.defining_ideal().hilbert_series() == ch.defining_ideal().gens().
↪ideal().hilbert_series()
True
```

```
>>> from sage.all import *
>>> ch = Matroid(graphs.CycleGraph(Integer(3))).chow_ring(QQ, False)
>>> ch.defining_ideal().groebner_basis()
[A0*A1, A0*A2, A1*A2, A0 + A3, A1 + A3, A2 + A3, A0*A3, A1*A3, A2*A3, A3^2]
>>> ch.defining_ideal().groebner_basis().is_groebner()
True
>>> ch.defining_ideal().hilbert_series() == ch.defining_ideal().gens().
↪ideal().hilbert_series()
True
```

normal_basis(*algorithm=''*, **args*, ***kwargs*)

Return the monomial basis of the quotient ring of this ideal.

EXAMPLES:

```
sage: ch = matroids.Z(3).chow_ring(QQ, False)
sage: I = ch.defining_ideal()
sage: I.normal_basis()
[1, Ax2x3y1, Ax1x3y2, Ay1y2y3, Ax1x2y3, Atx3y3, Atx2y2, Atx1y1,
 Atx1x2x3y1y2y3, Atx1x2x3y1y2y3^2]
sage: set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
sage: ch = matroids.AG(2,3).chow_ring(QQ, False)
sage: I = ch.defining_ideal()
```

(continues on next page)

(continued from previous page)

```
sage: I.normal_basis()
[1, A012, A345, A236, A156, A046, A247, A137, A057, A678, A258,
 A148, A038, A012345678, A012345678^2]
sage: set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
```

```
>>> from sage.all import *
>>> ch = matroids.Z(Integer(3)).chow_ring(QQ, False)
>>> I = ch.defining_ideal()
>>> I.normal_basis()
[1, Ax2x3y1, Ax1x3y2, Ay1y2y3, Ax1x2y3, Atx3y3, Atx2y2, Atx1y1,
 Atx1x2x3y1y2y3, Atx1x2x3y1y2y3^2]
>>> set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
>>> ch = matroids.AG(Integer(2),Integer(3)).chow_ring(QQ, False)
>>> I = ch.defining_ideal()
>>> I.normal_basis()
[1, A012, A345, A236, A156, A046, A247, A137, A057, A678, A258,
 A148, A038, A012345678, A012345678^2]
>>> set(I.gens().ideal().normal_basis()) == set(I.normal_basis())
True
```

4.2 Chow rings of matroids

AUTHORS:

- Shriya M

class sage.matroids.chow_ring.**ChowRing**(*R*, *M*, *augmented*, *presentation=None*)

Bases: QuotientRing_generic

The Chow ring of a matroid.

The *Chow ring of the matroid* M is defined as the quotient ring

$$A^*(M)_R := R[x_{F_1}, \ldots, x_{F_k}]/(I_M + J_M),$$

where $(I_M + J_M)$ is the *Chow ring ideal* of matroid M.

The *augmented Chow ring of matroid* M has two different presentations as quotient rings:

The *Feitchner-Yuzvinsky presentation* is the quotient ring

$$A(M)_R := R[y_{e_1}, \ldots, y_{e_n}, x_{F_1}, \ldots, x_{F_k}]/I_{FY}(M),$$

where $I_{FY}(M)$ is the *Feitchner-Yuzvinsky augmented Chow ring ideal* of matroid M.

The *atom-free presentation* is the quotient ring

$$A(M)_R := R[x_{F_1}, \ldots, x_{F_k}]/I_{af}(M),$$

where $I_{af}(M)$ is the *atom-free augmented Chow ring ideal* of matroid M.

See also

sage.matroids.chow_ring_ideal

Warning

Different presentations of Chow rings of non-simple matroids may not be isomorphic to one another.

INPUT:

- `M` – matroid
- `R` – commutative ring
- `augmented` – boolean; when `True`, this is the augmented Chow ring and if `False`, this is the non-augmented Chow ring
- `presentation` – string (default: `None`); one of the following (ignored if `augmented=False`)
  - `"fy"` - the Feitchner-Yuzvinsky presentation
  - `"atom-free"` - the atom-free presentation

REFERENCES:

- [FY2004]
- [AHK2015]

EXAMPLES:

```
sage: M1 = matroids.catalog.P8pp()
sage: ch = M1.chow_ring(QQ, False)
sage: ch
Chow ring of P8'': Matroid of rank 4 on 8 elements with 8 nonspanning circuits
over Rational Field
```

```
>>> from sage.all import *
>>> M1 = matroids.catalog.P8pp()
>>> ch = M1.chow_ring(QQ, False)
>>> ch
Chow ring of P8'': Matroid of rank 4 on 8 elements with 8 nonspanning circuits
over Rational Field
```

class Element(*parent*, *rep*, *reduce=True*)

Bases: `QuotientRingElement`

degree()

Return the degree of `self`.

EXAMPLES:

```
sage: ch = matroids.Uniform(3, 6).chow_ring(QQ, False)
sage: for b in ch.basis():
....:     print(b, b.degree())
1 0
```

(continues on next page)

(continued from previous page)

```
A01 1
A12 1
A02 1
A23 1
A13 1
A03 1
A34 1
A24 1
A14 1
A04 1
A45 1
A35 1
A25 1
A15 1
A05 1
A012345 1
A012345^2 2
sage: v = sum(ch.basis())
sage: v.degree()
2
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(3), Integer(6)).chow_ring(QQ, False)
>>> for b in ch.basis():
...     print(b, b.degree())
1 0
A01 1
A12 1
A02 1
A23 1
A13 1
A03 1
A34 1
A24 1
A14 1
A04 1
A45 1
A35 1
A25 1
A15 1
A05 1
A012345 1
A012345^2 2
>>> v = sum(ch.basis())
>>> v.degree()
2
```

homogeneous_degree()

Return the (homogeneous) degree of `self` if homogeneous otherwise raise an error.

EXAMPLES:

```
sage: ch = matroids.catalog.Fano().chow_ring(QQ, True, 'fy')
sage: for b in ch.basis():
....:     print(b, b.homogeneous_degree())
1 0
Ba 1
Ba*Babcdefg 2
Bb 1
Bb*Babcdefg 2
Bc 1
Bc*Babcdefg 2
Bd 1
Bd*Babcdefg 2
Bbcd 1
Bbcd^2 2
Be 1
Be*Babcdefg 2
Bace 1
Bace^2 2
Bf 1
Bf*Babcdefg 2
Bdef 1
Bdef^2 2
Babf 1
Babf^2 2
Bg 1
Bg*Babcdefg 2
Bcfg 1
Bcfg^2 2
Bbeg 1
Bbeg^2 2
Badg 1
Badg^2 2
Babcdefg 1
Babcdefg^2 2
Babcdefg^3 3
sage: v = sum(ch.basis()); v
Babcdefg^3 + Babf^2 + Bace^2 + Badg^2 + Bbcd^2 + Bbeg^2 +
Bcfg^2 + Bdef^2 + Ba*Babcdefg + Bb*Babcdefg + Bc*Babcdefg +
Bd*Babcdefg + Be*Babcdefg + Bf*Babcdefg + Bg*Babcdefg +
Babcdefg^2 + Ba + Bb + Bc + Bd + Be + Bf + Bg + Babf + Bace +
Badg + Bbcd + Bbeg + Bcfg + Bdef + Babcdefg + 1
sage: v.homogeneous_degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
```

```
>>> from sage.all import *
>>> ch = matroids.catalog.Fano().chow_ring(QQ, True, 'fy')
>>> for b in ch.basis():
...     print(b, b.homogeneous_degree())
1 0
Ba 1
```

(continues on next page)

(continued from previous page)

```
Ba*Babcdefg 2
Bb 1
Bb*Babcdefg 2
Bc 1
Bc*Babcdefg 2
Bd 1
Bd*Babcdefg 2
Bbcd 1
Bbcd^2 2
Be 1
Be*Babcdefg 2
Bace 1
Bace^2 2
Bf 1
Bf*Babcdefg 2
Bdef 1
Bdef^2 2
Babf 1
Babf^2 2
Bg 1
Bg*Babcdefg 2
Bcfg 1
Bcfg^2 2
Bbeg 1
Bbeg^2 2
Badg 1
Badg^2 2
Babcdefg 1
Babcdefg^2 2
Babcdefg^3 3
>>> v = sum(ch.basis()); v
Babcdefg^3 + Babf^2 + Bace^2 + Badg^2 + Bbcd^2 + Bbeg^2 +
Bcfg^2 + Bdef^2 + Ba*Babcdefg + Bb*Babcdefg + Bc*Babcdefg +
Bd*Babcdefg + Be*Babcdefg + Bf*Babcdefg + Bg*Babcdefg +
Babcdefg^2 + Ba + Bb + Bc + Bd + Be + Bf + Bg + Babf + Bace +
Badg + Bbcd + Bbeg + Bcfg + Bdef + Babcdefg + 1
>>> v.homogeneous_degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
```

monomial_coefficients(*copy=None*)

Return the monomial coefficients of `self`.

EXAMPLES:

```
sage: ch = matroids.catalog.NonFano().chow_ring(QQ, True, 'atom-free')
sage: v = ch.an_element(); v
Aa
sage: v.monomial_coefficients()
{0: 0, 1: 1, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0,
 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 0, 16: 0, 17: 0,
```

(continues on next page)

(continued from previous page)

```
 18: 0, 19: 0, 20: 0, 21: 0, 22: 0, 23: 0, 24: 0, 25: 0,
 26: 0, 27: 0, 28: 0, 29: 0, 30: 0, 31: 0, 32: 0, 33: 0,
 34: 0, 35: 0}
```

```
>>> from sage.all import *
>>> ch = matroids.catalog.NonFano().chow_ring(QQ, True, 'atom-free')
>>> v = ch.an_element(); v
Aa
>>> v.monomial_coefficients()
{0: 0, 1: 1, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0,
 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 0, 16: 0, 17: 0,
 18: 0, 19: 0, 20: 0, 21: 0, 22: 0, 23: 0, 24: 0, 25: 0,
 26: 0, 27: 0, 28: 0, 29: 0, 30: 0, 31: 0, 32: 0, 33: 0,
 34: 0, 35: 0}
```

to_vector(*order=None*)

Return `self` as a (dense) free module vector.

EXAMPLES:

```
sage: ch = matroids.Uniform(3, 6).chow_ring(QQ, False)
sage: v = ch.an_element(); v
-A01 - A02 - A03 - A04 - A05 - A012345
sage: v.to_vector()
(0, -1, 0, -1, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, 0, -1, -1, 0)
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(3), Integer(6)).chow_ring(QQ, False)
>>> v = ch.an_element(); v
-A01 - A02 - A03 - A04 - A05 - A012345
>>> v.to_vector()
(0, -1, 0, -1, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, 0, -1, -1, 0)
```

basis()

Return the monomial basis of the given Chow ring.

EXAMPLES:

```
sage: ch = matroids.Uniform(3, 6).chow_ring(QQ, True, 'fy')
sage: ch.basis()
Family (1, B0, B0*B012345, B1, B1*B012345, B01, B01^2, B2,
 B2*B012345, B12, B12^2, B02, B02^2, B3, B3*B012345, B23, B23^2,
 B13, B13^2, B03, B03^2, B4, B4*B012345, B34, B34^2, B24, B24^2,
 B14, B14^2, B04, B04^2, B5, B5*B012345, B45, B45^2, B35, B35^2,
 B25, B25^2, B15, B15^2, B05, B05^2, B012345, B012345^2, B012345^3)
sage: set(ch.defining_ideal().normal_basis()) == set(ch.basis())
True
sage: ch = matroids.catalog.Fano().chow_ring(QQ, False)
sage: ch.basis()
Family (1, Abcd, Aace, Adef, Aabf, Acfg, Abeg, Aadg, Aabcdefg,
 Aabcdefg^2)
sage: set(ch.defining_ideal().normal_basis()) == set(ch.basis())
```

(continues on next page)

(continued from previous page)

```
True
sage: ch = matroids.Wheel(3).chow_ring(QQ, True, 'atom-free')
sage: ch.basis()
Family (1, A0, A0*A012345, A1, A1*A012345, A2, A2*A012345, A3,
 A3*A012345, A23, A23^2, A013, A013^2, A4, A4*A012345, A124, A124^2,
 A04, A04^2, A5, A5*A012345, A345, A345^2, A15, A15^2, A025, A025^2,
 A012345, A012345^2, A012345^3)
sage: set(ch.defining_ideal().normal_basis()) == set(ch.basis())
True
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(3), Integer(6)).chow_ring(QQ, True, 'fy')
>>> ch.basis()
Family (1, B0, B0*B012345, B1, B1*B012345, B01, B01^2, B2,
 B2*B012345, B12, B12^2, B02, B02^2, B3, B3*B012345, B23, B23^2,
 B13, B13^2, B03, B03^2, B4, B4*B012345, B34, B34^2, B24, B24^2,
 B14, B14^2, B04, B04^2, B5, B5*B012345, B45, B45^2, B35, B35^2,
 B25, B25^2, B15, B15^2, B05, B05^2, B012345, B012345^2, B012345^3)
>>> set(ch.defining_ideal().normal_basis()) == set(ch.basis())
True
>>> ch = matroids.catalog.Fano().chow_ring(QQ, False)
>>> ch.basis()
Family (1, Abcd, Aace, Adef, Aabf, Acfg, Abeg, Aadg, Aabcdefg,
 Aabcdefg^2)
>>> set(ch.defining_ideal().normal_basis()) == set(ch.basis())
True
>>> ch = matroids.Wheel(Integer(3)).chow_ring(QQ, True, 'atom-free')
>>> ch.basis()
Family (1, A0, A0*A012345, A1, A1*A012345, A2, A2*A012345, A3,
 A3*A012345, A23, A23^2, A013, A013^2, A4, A4*A012345, A124, A124^2,
 A04, A04^2, A5, A5*A012345, A345, A345^2, A15, A15^2, A025, A025^2,
 A012345, A012345^2, A012345^3)
>>> set(ch.defining_ideal().normal_basis()) == set(ch.basis())
True
```

lefschetz_element()

Return one Lefschetz element of the given Chow ring.

EXAMPLES:

```
sage: ch = matroids.catalog.P8pp().chow_ring(QQ, False)
sage: ch.lefschetz_element()
-2*Aab - 2*Aac - 2*Aad - 2*Aae - 2*Aaf - 2*Aag - 2*Aah - 2*Abc
- 2*Abd - 2*Abe - 2*Abf - 2*Abg - 2*Abh - 2*Acd - 2*Ace - 2*Acf
- 2*Acg - 2*Ach - 2*Ade - 2*Adf - 2*Adg - 2*Adh - 2*Aef - 2*Aeg
- 2*Aeh - 2*Afg - 2*Afh - 2*Agh - 6*Aabc - 6*Aabd - 6*Aabe
- 12*Aabfh - 6*Aabg - 6*Aacd - 12*Aacef - 12*Aacgh - 12*Aadeg
- 6*Aadf - 6*Aadh - 6*Aaeh - 6*Aafg - 6*Abcd - 12*Abceg
- 6*Abcf - 6*Abch - 12*Abdeh - 12*Abdfg - 6*Abef - 6*Abgh
- 6*Acde - 12*Acdfh - 6*Acdg - 6*Aceh - 6*Acfg - 6*Adef
- 6*Adgh - 6*Aefg - 6*Aefh - 6*Aegh - 6*Afgh - 56*Aabcdefgh
```

```
>>> from sage.all import *
>>> ch = matroids.catalog.P8pp().chow_ring(QQ, False)
>>> ch.lefschetz_element()
-2*Aab - 2*Aac - 2*Aad - 2*Aae - 2*Aaf - 2*Aag - 2*Aah - 2*Abc
- 2*Abd - 2*Abe - 2*Abf - 2*Abg - 2*Abh - 2*Acd - 2*Ace - 2*Acf
- 2*Acg - 2*Ach - 2*Ade - 2*Adf - 2*Adg - 2*Adh - 2*Aef - 2*Aeg
- 2*Aeh - 2*Afg - 2*Afh - 2*Agh - 6*Aabc - 6*Aabd - 6*Aabe
- 12*Aabfh - 6*Aabg - 6*Aacd - 12*Aacef - 12*Aacgh - 12*Aadeg
- 6*Aadf - 6*Aadh - 6*Aaeh - 6*Aafg - 6*Abcd - 12*Abceg
- 6*Abcf - 6*Abch - 12*Abdeh - 12*Abdfg - 6*Abef - 6*Abgh
- 6*Acde - 12*Acdfh - 6*Acdg - 6*Aceh - 6*Acfg - 6*Adef
- 6*Adgh - 6*Aefg - 6*Aefh - 6*Aegh - 6*Afgh - 56*Aabcdefgh
```

The following example finds the Lefschetz element of the Chow ring of the uniform matroid of rank 4 on 5 elements (non-augmented). It is then multiplied with the elements of FY-monomial bases of different degrees:

```
sage: ch = matroids.Uniform(4, 5).chow_ring(QQ, False)
sage: basis_deg = {}
sage: for b in ch.basis():
....:     deg = b.homogeneous_degree()
....:     if deg not in basis_deg:
....:         basis_deg[deg] = []
....:     basis_deg[deg].append(b)
....:
sage: basis_deg
{0: [1],
 1: [A01, A12, A02, A012, A23, A13, A123, A03, A013, A023, A34, A24,
     A234, A14, A124, A134, A04, A014, A024, A034, A01234],
 2: [A01*A01234, A12*A01234, A02*A01234, A012^2, A23*A01234,
     A13*A01234, A123^2, A03*A01234, A013^2, A023^2, A34*A01234,
     A24*A01234, A234^2, A14*A01234, A124^2, A134^2, A04*A01234,
     A014^2, A024^2, A034^2, A01234^2],
 3: [A01234^3]}
sage: g_eq_maps = {}
sage: lefschetz_el = ch.lefschetz_element(); lefschetz_el
-2*A01 - 2*A02 - 2*A03 - 2*A04 - 2*A12 - 2*A13 - 2*A14 - 2*A23
- 2*A24 - 2*A34 - 6*A012 - 6*A013 - 6*A014 - 6*A023 - 6*A024
- 6*A034 - 6*A123 - 6*A124 - 6*A134 - 6*A234 - 20*A01234
sage: for deg in basis_deg:
....:     if deg not in g_eq_maps:
....:         g_eq_maps[deg] = []
....:     g_eq_maps[deg].extend([i*lefschetz_el for i in basis_deg[deg]])
....:
sage: g_eq_maps
{0: [-2*A01 - 2*A02 - 2*A03 - 2*A04 - 2*A12 - 2*A13 - 2*A14 - 2*A23
     - 2*A24 - 2*A34 - 6*A012 - 6*A013 - 6*A014 - 6*A023 - 6*A024
     - 6*A034 - 6*A123 - 6*A124 - 6*A134 - 6*A234 - 20*A01234],
 1: [2*A012^2 + 2*A013^2 + 2*A014^2 - 10*A01*A01234 + 2*A01234^2,
     2*A012^2 + 2*A123^2 + 2*A124^2 - 10*A12*A01234 + 2*A01234^2,
     2*A012^2 + 2*A023^2 + 2*A024^2 - 10*A02*A01234 + 2*A01234^2,
     -6*A012^2 + 2*A01*A01234 + 2*A02*A01234 + 2*A12*A01234,
     2*A023^2 + 2*A123^2 + 2*A234^2 - 10*A23*A01234 + 2*A01234^2,
```

(continues on next page)

(continued from previous page)

```
     2*A013^2 + 2*A123^2 + 2*A134^2 - 10*A13*A01234 + 2*A01234^2,
     -6*A123^2 + 2*A12*A01234 + 2*A13*A01234 + 2*A23*A01234,
     2*A013^2 + 2*A023^2 + 2*A034^2 - 10*A03*A01234 + 2*A01234^2,
     -6*A013^2 + 2*A01*A01234 + 2*A03*A01234 + 2*A13*A01234,
     -6*A023^2 + 2*A02*A01234 + 2*A03*A01234 + 2*A23*A01234,
     2*A034^2 + 2*A134^2 + 2*A234^2 - 10*A34*A01234 + 2*A01234^2,
     2*A024^2 + 2*A124^2 + 2*A234^2 - 10*A24*A01234 + 2*A01234^2,
     -6*A234^2 + 2*A23*A01234 + 2*A24*A01234 + 2*A34*A01234,
     2*A014^2 + 2*A124^2 + 2*A134^2 - 10*A14*A01234 + 2*A01234^2,
     -6*A124^2 + 2*A12*A01234 + 2*A14*A01234 + 2*A24*A01234,
     -6*A134^2 + 2*A13*A01234 + 2*A14*A01234 + 2*A34*A01234,
     2*A014^2 + 2*A024^2 + 2*A034^2 - 10*A04*A01234 + 2*A01234^2,
     -6*A014^2 + 2*A01*A01234 + 2*A04*A01234 + 2*A14*A01234,
     -6*A024^2 + 2*A02*A01234 + 2*A04*A01234 + 2*A24*A01234,
     -6*A034^2 + 2*A03*A01234 + 2*A04*A01234 + 2*A34*A01234,
     -2*A01*A01234 - 2*A02*A01234 - 2*A03*A01234 - 2*A04*A01234
     - 2*A12*A01234 - 2*A13*A01234 - 2*A14*A01234 - 2*A23*A01234
     - 2*A24*A01234 - 2*A34*A01234 - 20*A01234^2],
 2: [2*A01234^3, 2*A01234^3, 2*A01234^3, 6*A01234^3, 2*A01234^3,
     2*A01234^3, 6*A01234^3, 2*A01234^3, 6*A01234^3, 6*A01234^3,
     2*A01234^3, 2*A01234^3, 6*A01234^3, 2*A01234^3, 6*A01234^3,
     6*A01234^3, 2*A01234^3, 6*A01234^3, 6*A01234^3, 6*A01234^3,
     -20*A01234^3],
 3: [0]}
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(4), Integer(5)).chow_ring(QQ, False)
>>> basis_deg = {}
>>> for b in ch.basis():
...     deg = b.homogeneous_degree()
...     if deg not in basis_deg:
...         basis_deg[deg] = []
...     basis_deg[deg].append(b)
....:
>>> basis_deg
{0: [1],
 1: [A01, A12, A02, A012, A23, A13, A123, A03, A013, A023, A34, A24,
     A234, A14, A124, A134, A04, A014, A024, A034, A01234],
 2: [A01*A01234, A12*A01234, A02*A01234, A012^2, A23*A01234,
     A13*A01234, A123^2, A03*A01234, A013^2, A023^2, A34*A01234,
     A24*A01234, A234^2, A14*A01234, A124^2, A134^2, A04*A01234,
     A014^2, A024^2, A034^2, A01234^2],
 3: [A01234^3]}
>>> g_eq_maps = {}
>>> lefschetz_el = ch.lefschetz_element(); lefschetz_el
-2*A01 - 2*A02 - 2*A03 - 2*A04 - 2*A12 - 2*A13 - 2*A14 - 2*A23
- 2*A24 - 2*A34 - 6*A012 - 6*A013 - 6*A014 - 6*A023 - 6*A024
- 6*A034 - 6*A123 - 6*A124 - 6*A134 - 6*A234 - 20*A01234
>>> for deg in basis_deg:
...     if deg not in g_eq_maps:
...         g_eq_maps[deg] = []
...     g_eq_maps[deg].extend([i*lefschetz_el for i in basis_deg[deg]])
```

(continues on next page)

(continued from previous page)

```
....:
>>> g_eq_maps
{0: [-2*A01 - 2*A02 - 2*A03 - 2*A04 - 2*A12 - 2*A13 - 2*A14 - 2*A23
     - 2*A24 - 2*A34 - 6*A012 - 6*A013 - 6*A014 - 6*A023 - 6*A024
     - 6*A034 - 6*A123 - 6*A124 - 6*A134 - 6*A234 - 20*A01234],
 1: [2*A012^2 + 2*A013^2 + 2*A014^2 - 10*A01*A01234 + 2*A01234^2,
     2*A012^2 + 2*A123^2 + 2*A124^2 - 10*A12*A01234 + 2*A01234^2,
     2*A012^2 + 2*A023^2 + 2*A024^2 - 10*A02*A01234 + 2*A01234^2,
     -6*A012^2 + 2*A01*A01234 + 2*A02*A01234 + 2*A12*A01234,
     2*A023^2 + 2*A123^2 + 2*A234^2 - 10*A23*A01234 + 2*A01234^2,
     2*A013^2 + 2*A123^2 + 2*A134^2 - 10*A13*A01234 + 2*A01234^2,
     -6*A123^2 + 2*A12*A01234 + 2*A13*A01234 + 2*A23*A01234,
     2*A013^2 + 2*A023^2 + 2*A034^2 - 10*A03*A01234 + 2*A01234^2,
     -6*A013^2 + 2*A01*A01234 + 2*A03*A01234 + 2*A13*A01234,
     -6*A023^2 + 2*A02*A01234 + 2*A03*A01234 + 2*A23*A01234,
     2*A034^2 + 2*A134^2 + 2*A234^2 - 10*A34*A01234 + 2*A01234^2,
     2*A024^2 + 2*A124^2 + 2*A234^2 - 10*A24*A01234 + 2*A01234^2,
     -6*A234^2 + 2*A23*A01234 + 2*A24*A01234 + 2*A34*A01234,
     2*A014^2 + 2*A124^2 + 2*A134^2 - 10*A14*A01234 + 2*A01234^2,
     -6*A124^2 + 2*A12*A01234 + 2*A14*A01234 + 2*A24*A01234,
     -6*A134^2 + 2*A13*A01234 + 2*A14*A01234 + 2*A34*A01234,
     2*A014^2 + 2*A024^2 + 2*A034^2 - 10*A04*A01234 + 2*A01234^2,
     -6*A014^2 + 2*A01*A01234 + 2*A04*A01234 + 2*A14*A01234,
     -6*A024^2 + 2*A02*A01234 + 2*A04*A01234 + 2*A24*A01234,
     -6*A034^2 + 2*A03*A01234 + 2*A04*A01234 + 2*A34*A01234,
     -2*A01*A01234 - 2*A02*A01234 - 2*A03*A01234 - 2*A04*A01234
     - 2*A12*A01234 - 2*A13*A01234 - 2*A14*A01234 - 2*A23*A01234
     - 2*A24*A01234 - 2*A34*A01234 - 20*A01234^2],
 2: [2*A01234^3, 2*A01234^3, 2*A01234^3, 6*A01234^3, 2*A01234^3,
     2*A01234^3, 6*A01234^3, 2*A01234^3, 6*A01234^3, 6*A01234^3,
     2*A01234^3, 2*A01234^3, 6*A01234^3, 2*A01234^3, 6*A01234^3,
     6*A01234^3, 2*A01234^3, 6*A01234^3, 6*A01234^3, 6*A01234^3,
     -20*A01234^3],
 3: [0]}
```

matroid()

Return the matroid of `self`.

EXAMPLES:

```
sage: ch = matroids.Uniform(3, 6).chow_ring(QQ, True, 'fy')
sage: ch.matroid()
U(3, 6): Matroid of rank 3 on 6 elements with circuit-closures
{3: {{0, 1, 2, 3, 4, 5}}}
```

```
>>> from sage.all import *
>>> ch = matroids.Uniform(Integer(3), Integer(6)).chow_ring(QQ, True, 'fy')
>>> ch.matroid()
U(3, 6): Matroid of rank 3 on 6 elements with circuit-closures
{3: {{0, 1, 2, 3, 4, 5}}}
```

poincare_pairing(*el1*, *el2*)

Return the Poincaré pairing of any two elements of the Chow ring.

EXAMPLES:

```
sage: ch = matroids.Wheel(3).chow_ring(QQ, True, 'atom-free')
sage: A0, A1, A2, A3, A4, A5, A013, A025, A04, A124, A15, A23, A345, A012345␣
↪= ch.gens()
sage: u = ch(-1/6*A2*A012345 + 41/48*A012345^2); u
-1/6*A2*A012345 + 41/48*A012345^2
sage: v = ch(-A345^2 - 1/4*A345); v
-A345^2 - 1/4*A345
sage: ch.poincare_pairing(v, u)
3
```

```
>>> from sage.all import *
>>> ch = matroids.Wheel(Integer(3)).chow_ring(QQ, True, 'atom-free')
>>> A0, A1, A2, A3, A4, A5, A013, A025, A04, A124, A15, A23, A345, A012345 =␣
↪ch.gens()
>>> u = ch(-Integer(1)/Integer(6)*A2*A012345 + Integer(41)/
↪Integer(48)*A012345**Integer(2)); u
-1/6*A2*A012345 + 41/48*A012345^2
>>> v = ch(-A345**Integer(2) - Integer(1)/Integer(4)*A345); v
-A345^2 - 1/4*A345
>>> ch.poincare_pairing(v, u)
3
```

CHAPTER

FIVE

ABSTRACT MATROID CLASSES

5.1 Basis exchange matroids

BasisExchangeMatroid is an abstract class implementing default matroid functionality in terms of basis exchange. Several concrete matroid classes are subclasses of this. They have the following methods in addition to the ones provided by the parent class *Matroid*.

- *bases_count()*
- *groundset_list()*

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

class sage.matroids.basis_exchange_matroid.**BasisExchangeMatroid**

Bases: *Matroid*

Class BasisExchangeMatroid is a virtual class that derives from Matroid. It implements each of the elementary matroid methods (*rank()*, *max_independent()*, *circuit()*, *closure()* etc.), essentially by crawling the base exchange graph of the matroid. This is the graph whose vertices are the bases of the matroid, two bases being adjacent in the graph if their symmetric difference has 2 members.

This base exchange graph is not stored as such, but should be provided implicitly by the child class in the form of two methods `_is_exchange_pair(x, y)` and `_exchange(x, y)`, as well as an initial basis. At any moment, BasisExchangeMatroid keeps a current basis B. The method `_is_exchange_pair(x, y)` should return a boolean indicating whether $B - x + y$ is a basis. The method `_exchange(x, y)` is called when the current basis B is replaced by said $B - x + y$. It is up to the child class to update its internal data structure to make information relative to the new basis more accessible. For instance, a linear matroid would perform a row reduction to make the column labeled by y a standard basis vector (and therefore the columns indexed by $B - x + y$ would form an identity matrix).

Each of the elementary matroid methods has a straightforward greedy-type implementation in terms of these two methods. For example, given a subset F of the groundset, one can step to a basis B over the edges of the base exchange graph which has maximal intersection with F, in each step increasing the intersection of the current B with F. Then one computes the rank of F as the cardinality of the intersection of F and B.

The following matroid classes can/will implement their oracle efficiently by deriving from `BasisExchangeMatroid`:

- *BasisMatroid*: keeps a list of all bases.
  - `_is_exchange_pair(x, y)` reduces to a query whether $B - x + y$ is a basis.
  - `_exchange(x, y)` has no work to do.
- *LinearMatroid*: keeps a matrix representation A of the matroid so that $A[B] = I$.

  - `_is_exchange_pair(x, y)` reduces to testing whether $A[r, y]$ is nonzero, where $A[r, x] = 1$.
  - `_exchange(x, y)` should modify the matrix so that $A[B - x + y]$ becomes I, which means pivoting on $A[r, y]$.

- `TransversalMatroid` (not yet implemented): If A is a set of subsets of E, then I is independent if it is a system of distinct representatives of A, i.e. if I is covered by a matching of an appropriate bipartite graph G, with color classes A and E and an edge (A_i, e) if e is in the subset A_i. At any time you keep a maximum matching M of G covering the current basis B.
  - `_is_exchange_pair(x, y)` checks for the existence of an M-alternating path P from y to x.
  - `_exchange(x, y)` replaces M by the symmetric difference of M and $E(P)$.
- `AlgebraicMatroid` (not yet implemented): keeps a list of polynomials in variables $E - B + e$ for each variable e in B.
  - `_is_exchange_pair(x, y)` checks whether the polynomial that relates y to $E - B$ uses x.
  - `_exchange(x, y)` make new list of polynomials by computing resultants.

All but the first of the above matroids are algebraic, and all implementations specializations of the algebraic one.

BasisExchangeMatroid internally renders subsets of the groundset as bitsets. It provides optimized methods for enumerating bases, nonbases, flats, circuits, etc.

bases_count()

Return the number of bases of the matroid.

A *basis* is an inclusionwise maximal independent set.

See also

M.basis().

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.N1()
sage: M.bases_count()
184
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N1()
>>> M.bases_count()
184
```

basis()

Return an arbitrary basis of the matroid.

A *basis* is an inclusionwise maximal independent set.

Note

The output of this method can change in between calls. It reflects the internal state of the matroid. This state is updated by lots of methods, including the method `M._move_current_basis()`.

OUTPUT: set of elements

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted(M.basis())
['a', 'b', 'c']
sage: M.rank('cd')
2
sage: sorted(M.basis())
['a', 'c', 'd']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted(M.basis())
['a', 'b', 'c']
>>> M.rank('cd')
2
>>> sorted(M.basis())
['a', 'c', 'd']
```

circuits(*k=None*)

Return the circuits of the matroid.

OUTPUT: iterable containing all circuits

See also

M.circuit()

EXAMPLES:

```
sage: M = Matroid(matroids.catalog.NonFano().bases())
sage: sorted([sorted(C) for C in M.circuits()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'b', 'f'],
 ['a', 'c', 'd', 'f'], ['a', 'c', 'e'], ['a', 'd', 'e', 'f'],
 ['a', 'd', 'g'], ['a', 'e', 'f', 'g'], ['b', 'c', 'd'],
 ['b', 'c', 'e', 'f'], ['b', 'd', 'e', 'f'], ['b', 'd', 'f', 'g'],
 ['b', 'e', 'g'], ['c', 'd', 'e', 'f'], ['c', 'd', 'e', 'g'],
 ['c', 'f', 'g'], ['d', 'e', 'f', 'g']]
```

```
>>> from sage.all import *
>>> M = Matroid(matroids.catalog.NonFano().bases())
>>> sorted([sorted(C) for C in M.circuits()])
[['a', 'b', 'c', 'g'], ['a', 'b', 'd', 'e'], ['a', 'b', 'f'],
 ['a', 'c', 'd', 'f'], ['a', 'c', 'e'], ['a', 'd', 'e', 'f'],
 ['a', 'd', 'g'], ['a', 'e', 'f', 'g'], ['b', 'c', 'd'],
 ['b', 'c', 'e', 'f'], ['b', 'd', 'e', 'f'], ['b', 'd', 'f', 'g'],
 ['b', 'e', 'g'], ['c', 'd', 'e', 'f'], ['c', 'd', 'e', 'g'],
 ['c', 'f', 'g'], ['d', 'e', 'f', 'g']]
```

cocircuits()

Return the cocircuits of the matroid.

OUTPUT: iterable containing all cocircuits

See also

Matroid.cocircuit()

EXAMPLES:

```
sage: M = Matroid(bases=matroids.catalog.NonFano().bases())
sage: sorted([sorted(C) for C in M.cocircuits()])
[['a', 'b', 'c', 'd', 'g'], ['a', 'b', 'c', 'e', 'g'],
 ['a', 'b', 'c', 'f', 'g'], ['a', 'b', 'd', 'e'],
 ['a', 'c', 'd', 'f'], ['a', 'e', 'f', 'g'], ['b', 'c', 'e', 'f'],
 ['b', 'd', 'f', 'g'], ['c', 'd', 'e', 'g']]
```

```
>>> from sage.all import *
>>> M = Matroid(bases=matroids.catalog.NonFano().bases())
>>> sorted([sorted(C) for C in M.cocircuits()])
[['a', 'b', 'c', 'd', 'g'], ['a', 'b', 'c', 'e', 'g'],
 ['a', 'b', 'c', 'f', 'g'], ['a', 'b', 'd', 'e'],
 ['a', 'c', 'd', 'f'], ['a', 'e', 'f', 'g'], ['b', 'c', 'e', 'f'],
 ['b', 'd', 'f', 'g'], ['c', 'd', 'e', 'g']]
```

coflats(k)

Return the collection of coflats of the matroid of specified corank.

A *coflat* is a coclosed set.

INPUT:

- `k` – integer

OUTPUT: iterable containing all coflats of corank k

See also

Matroid.coclosure()

EXAMPLES:

```
sage: M = matroids.catalog.S8().dual()
sage: M.whitney_numbers2()
[1, 8, 22, 14, 1]
sage: len(M.coflats(2))
22
sage: len(M.coflats(8))
0
sage: len(M.coflats(4))
1
```

```
>>> from sage.all import *
>>> M = matroids.catalog.S8().dual()
>>> M.whitney_numbers2()
```

(continues on next page)

(continued from previous page)

```
[1, 8, 22, 14, 1]
>>> len(M.coflats(Integer(2)))
22
>>> len(M.coflats(Integer(8)))
0
>>> len(M.coflats(Integer(4)))
1
```

components()

Return an iterable containing the components of the matroid.

A *component* is an inclusionwise maximal connected subset of the matroid. A subset is *connected* if the matroid resulting from deleting the complement of that subset is *connected*.

OUTPUT: list of subsets

See also

M.is_connected(), *M.delete()*

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = Matroid(ring=QQ, matrix=[[1, 0, 0, 1, 1, 0],
....:                              [0, 1, 0, 1, 2, 0],
....:                              [0, 0, 1, 0, 0, 1]])
sage: setprint(M.components())
[{0, 1, 3, 4}, {2, 5}]
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = Matroid(ring=QQ, matrix=[[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1), Integer(0)],
...                              [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(2), Integer(0)],
...                              [Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)]])
>>> setprint(M.components())
[{0, 1, 3, 4}, {2, 5}]
```

dependent_sets(k)

Return the dependent sets of fixed size.

INPUT:

- `k` – integer

OUTPUT: iterable containing all dependent sets of size `k`

EXAMPLES:

```
sage: M = matroids.catalog.N1()
sage: len(M.nonbases())
68
```

(continues on next page)

(continued from previous page)

```
sage: [len(M.dependent_sets(k)) for k in range(M.full_rank() + 1)]
[0, 0, 0, 0, 9, 68]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N1()
>>> len(M.nonbases())
68
>>> [len(M.dependent_sets(k)) for k in range(M.full_rank() + Integer(1))]
[0, 0, 0, 0, 9, 68]
```

flats(*k*)

Return the collection of flats of the matroid of specified rank.

A *flat* is a closed set.

INPUT:

- `k` – integer

OUTPUT: `SetSystem`

See also

Matroid.closure()

EXAMPLES:

```
sage: M = matroids.catalog.S8()
sage: M.whitney_numbers2()
[1, 8, 22, 14, 1]
sage: len(M.flats(2))
22
sage: len(M.flats(8))
0
sage: len(M.flats(4))
1
```

```
>>> from sage.all import *
>>> M = matroids.catalog.S8()
>>> M.whitney_numbers2()
[1, 8, 22, 14, 1]
>>> len(M.flats(Integer(2)))
22
>>> len(M.flats(Integer(8)))
0
>>> len(M.flats(Integer(4)))
1
```

full_corank()

Return the corank of the matroid.

The *corank* of the matroid equals the rank of the dual matroid. It is given by `M.size() - M.full_rank()`.

OUTPUT: integer

See also

M.dual(), *M.full_rank()*

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.full_corank()
4
sage: M.dual().full_corank()
3
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.full_corank()
4
>>> M.dual().full_corank()
3
```

full_rank()

Return the rank of the matroid.

The *rank* of the matroid is the size of the largest independent subset of the groundset.

OUTPUT: integer

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.full_rank()
3
sage: M.dual().full_rank()
4
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.full_rank()
3
>>> M.dual().full_rank()
4
```

groundset()

Return the groundset of the matroid.

The groundset is the set of elements that comprise the matroid.

OUTPUT: set

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

groundset_list()

Return a list of elements of the groundset of the matroid.

The order of the list does not change between calls.

OUTPUT: list

See also

M.groundset()

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: type(M.groundset())
<... 'frozenset'>
sage: type(M.groundset_list())
<... 'list'>
sage: sorted(M.groundset_list())
['a', 'b', 'c', 'd', 'e', 'f', 'g']

sage: E = M.groundset_list()
sage: E.remove('a')
sage: sorted(M.groundset_list())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> type(M.groundset())
<... 'frozenset'>
>>> type(M.groundset_list())
<... 'list'>
>>> sorted(M.groundset_list())
['a', 'b', 'c', 'd', 'e', 'f', 'g']

>>> E = M.groundset_list()
>>> E.remove('a')
>>> sorted(M.groundset_list())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

independent_sets(*k=-1*)

Return the independent sets of the matroid.

INPUT:

- k – integer (optional); if specified, return the size-k independent sets of the matroid

OUTPUT: SetSystem

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: I = M.independent_sets()
sage: len(I)
57
sage: M = matroids.catalog.N1()
sage: M.bases_count()
184
sage: [len(M.independent_sets(k)) for k in range(M.full_rank() + 1)]
[1, 10, 45, 120, 201, 184]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> I = M.independent_sets()
>>> len(I)
57
>>> M = matroids.catalog.N1()
>>> M.bases_count()
184
>>> [len(M.independent_sets(k)) for k in range(M.full_rank() + Integer(1))]
[1, 10, 45, 120, 201, 184]
```

is_valid(*certificate=False*)

Test if the data obey the matroid axioms.

This method checks the basis axioms for the class. If B is the set of bases of a matroid M, then

- B is nonempty
- if X and Y are in B, and x is in $X - Y$, then there is a y in $Y - X$ such that $(X - x) + y$ is again a member of B.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: boolean, or (boolean, dictionary)

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid(matroids.catalog.Fano())
sage: M.is_valid()
True
sage: M = Matroid(groundset='abcd', bases=['ab', 'cd'])
sage: M.is_valid(certificate=True)
(False, {'error': 'exchange axiom failed'})
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = BasisMatroid(matroids.catalog.Fano())
>>> M.is_valid()
True
>>> M = Matroid(groundset='abcd', bases=['ab', 'cd'])
>>> M.is_valid(certificate=True)
(False, {'error': 'exchange axiom failed'})
```

noncospanning_cocircuits()

Return the noncospanning cocircuits of the matroid.

A *noncospanning cocircuit* is a cocircuit whose corank is strictly smaller than the corank of the matroid.

OUTPUT: iterable containing all nonspanning circuits

See also

Matroid.cocircuit(), *Matroid.corank()*

EXAMPLES:

```
sage: M = matroids.catalog.N1()
sage: len(M.noncospanning_cocircuits())
23
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N1()
>>> len(M.noncospanning_cocircuits())
23
```

nonspanning_circuits()

Return the nonspanning circuits of the matroid.

A *nonspanning circuit* is a circuit whose rank is strictly smaller than the rank of the matroid.

OUTPUT: iterable containing all nonspanning circuits

See also

Matroid.circuit(), *Matroid.rank()*

EXAMPLES:

```
sage: M = matroids.catalog.N1()
sage: len(M.nonspanning_circuits())
23
```

```
>>> from sage.all import *
>>> M = matroids.catalog.N1()
>>> len(M.nonspanning_circuits())
23
```

whitney_numbers2()

Return the Whitney numbers of the second kind of the matroid.

The Whitney numbers of the second kind are here encoded as a vector $(W_0, \ldots, W_r)$, where W_i is the number of flats of rank i, and r is the rank of the matroid.

OUTPUT: list of integers

EXAMPLES:

```
sage: M = matroids.catalog.S8()
sage: M.whitney_numbers2()
[1, 8, 22, 14, 1]
```

```
>>> from sage.all import *
>>> M = matroids.catalog.S8()
>>> M.whitney_numbers2()
[1, 8, 22, 14, 1]
```

5.2 Dual matroids

5.2.1 Theory

Let M be a matroid with groundset E. If B is the set of bases of M, then the set $\{E - b : b \in B\}$ is the set of bases of another matroid, the dual of M.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: N = M.dual()
sage: M.is_basis('abc')
True
sage: N.is_basis('defg')
True
sage: M.dual().dual() == M
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> N = M.dual()
>>> M.is_basis('abc')
True
>>> N.is_basis('defg')
True
>>> M.dual().dual() == M
True
```

5.2.2 Implementation

The class `DualMatroid` wraps around a matroid instance to represent its dual. Only useful for classes that don't have an explicit construction of the dual (such as *RankMatroid* and *CircuitClosuresMatroid*). It is also used as the default implementation of the method *M.dual()*. For direct access to the `DualMatroid` constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See also *sage.matroids.advanced*.

AUTHORS:

- Rudi Pendavingh, Michael Welsh, Stefan van Zwam (2013-04-01): initial version

class sage.matroids.dual_matroid.**DualMatroid**(*matroid*)

Bases: *Matroid*

Dual of a matroid.

For some matroid representations it can be computationally expensive to derive an explicit representation of the dual. This class wraps around any matroid to provide an abstract dual. It also serves as the default implementation of the dual.

INPUT:

- `matroid` – matroid

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.catalog.Vamos()
sage: Md = DualMatroid(M)  # indirect doctest
sage: Md.rank('abd') == M.corank('abd')
True
sage: Md
Dual of 'Vamos: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
     {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}'
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.Vamos()
>>> Md = DualMatroid(M)  # indirect doctest
>>> Md.rank('abd') == M.corank('abd')
True
>>> Md
Dual of 'Vamos: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
     {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}'
```

dual()

Return the dual of the matroid.

Let M be a matroid with groundset E. If B is the set of bases of M, then the set $\{E - b : b \in B\}$ is the set of bases of another matroid, the *dual* of M. Note that the dual of the dual of M equals M, so if this is the *DualMatroid* instance wrapping M then the returned matroid is M.

OUTPUT: the dual matroid

EXAMPLES:

```
sage: M = matroids.catalog.Pappus().dual()
sage: N = M.dual()
sage: N.rank()
3
sage: N
Pappus: Matroid of rank 3 on 9 elements with 9 nonspanning circuits
```

```
sage: from sage.all import *
sage: M = matroids.catalog.Pappus().dual()
sage: N = M.dual()
sage: N.rank()
3
sage: N
Pappus: Matroid of rank 3 on 9 elements with 9 nonspanning circuits
```

groundset()

Return the groundset of the matroid.

The groundset is the set of elements that comprise the matroid.

OUTPUT: set

EXAMPLES:

```
sage: M = matroids.catalog.Pappus().dual()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

```
sage: from sage.all import *
sage: M = matroids.catalog.Pappus().dual()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

is_valid(*certificate=False*)

Test if `self` obeys the matroid axioms.

For a *DualMatroid*, we check its dual.

INPUT:

- `certificate` – boolean (default: `False`)

OUTPUT: boolean, or (boolean, dictionary)

EXAMPLES:

```
sage: M = matroids.catalog.K5dual()
sage: type(M)
<class 'sage.matroids.dual_matroid.DualMatroid'>
sage: M.is_valid()
True
sage: M = Matroid([[0, 1], [2, 3]])
sage: M.dual().is_valid()
False
```

```
sage: from sage.all import *
sage: M = matroids.catalog.K5dual()
sage: type(M)
<class 'sage.matroids.dual_matroid.DualMatroid'>
sage: M.is_valid()
True
sage: M = Matroid([[Integer(0), Integer(1)], [Integer(2), Integer(3)]])
```

(continues on next page)

(continued from previous page)

```
>>> M.dual().is_valid()
False
```

relabel(*mapping*)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element e by `mapping[e]`, where `mapping` is a given injective map. If `mapping[e]` is not defined, then the identity map is assumed.

INPUT:

- `mapping` – a Python object such that `mapping[e]` is the new label of e

OUTPUT: matroid

EXAMPLES:

```
sage: M = matroids.catalog.K5dual(range(10))
sage: type(M)
<class 'sage.matroids.dual_matroid.DualMatroid'>
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: N = M.dual().relabel({0:10})
sage: sorted(N.groundset())
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: N.is_isomorphic(matroids.catalog.K5())
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.K5dual(range(Integer(10)))
>>> type(M)
<class 'sage.matroids.dual_matroid.DualMatroid'>
>>> sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> N = M.dual().relabel({Integer(0):Integer(10)})
>>> sorted(N.groundset())
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> N.is_isomorphic(matroids.catalog.K5())
True
```

5.3 Minors of matroids

5.3.1 Theory

Let M be a matroid with groundset E. There are two standard ways to remove an element from E so that the result is again a matroid, *deletion* and *contraction*. Deletion is simply omitting the elements from a set D from E and keeping all remaining independent sets. This is denoted `M \ D` (this also works in Sage). Contraction is the dual operation: `M / C == (M.dual() \ C).dual()`.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: M.delete(['a', 'c' ]) == M.delete(['a', 'c'])
True
```

(continues on next page)

(continued from previous page)

```
sage: M / 'a' == M.contract('a')
True
sage: (M / 'c').delete('ab') == M.minor(contractions='c', deletions='ab')
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.delete(['a', 'c' ]) == M.delete(['a', 'c'])
True
>>> M / 'a' == M.contract('a')
True
>>> (M / 'c').delete('ab') == M.minor(contractions='c', deletions='ab')
True
```

If a contraction set is not independent (or a deletion set not coindependent), this is taken care of:

```
sage: M = matroids.catalog.Fano()
sage: M.rank('abf')
2
sage: M / 'abf' == (M / 'ab').delete('f')
True
sage: M / 'abf' == (M / 'af').delete('b')
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.rank('abf')
2
>>> M / 'abf' == (M / 'ab').delete('f')
True
>>> M / 'abf' == (M / 'af').delete('b')
True
```

See also

M.delete(), *M.contract()*, *M.minor()*,

5.3.2 Implementation

The class *MinorMatroid* wraps around a matroid instance to represent a minor. Only useful for classes that don't have an explicit construction of minors (such as *RankMatroid* and *CircuitClosuresMatroid*). It is also used as default implementation of the minor methods *M.minor(C, D)*, *M.delete(D)*, *M.contract(C)*. For direct access to the `DualMatroid` constructor, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See also *sage.matroids.advanced*.

AUTHORS:

- Rudi Pendavingh, Michael Welsh, Stefan van Zwam (2013-04-01): initial version

5.3.3 Methods

class sage.matroids.minor_matroid.**MinorMatroid**(*matroid*, *contractions=None*, *deletions=None*)

Bases: *Matroid*

Minor of a matroid.

For some matroid representations, it can be computationally expensive to derive an explicit representation of a minor. This class wraps around any matroid to provide an abstract minor. It also serves as default implementation.

Return a minor.

INPUT:

- `matroid` – matroid
- `contractions` – an object with Python's `frozenset` interface containing a subset of `self.groundset()`.
- `deletions` – an object with Python's `frozenset` interface containing a subset of `self.groundset()`

OUTPUT:

A `MinorMatroid` instance representing `matroid / contractions \ deletions`.

Warning

This class does NOT do any checks. Besides the assumptions above, we assume the following:

- `contractions` is independent
- `deletions` is coindependent
- `contractions` and `deletions` are disjoint.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.catalog.Vamos()
sage: N = MinorMatroid(matroid=M, contractions=set(['a']),
....:                  deletions=set())
sage: N._minor(contractions=set(), deletions=set(['b', 'c']))
M / {'a'} \ {'b', 'c'}, where M is Vamos:
Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
     {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.Vamos()
>>> N = MinorMatroid(matroid=M, contractions=set(['a']),
...                  deletions=set())
>>> N._minor(contractions=set(), deletions=set(['b', 'c']))
M / {'a'} \ {'b', 'c'}, where M is Vamos:
Matroid of rank 4 on 8 elements with circuit-closures
```

(continues on next page)

(continued from previous page)

```
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
     {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
```

groundset()

Return the groundset of the matroid.

EXAMPLES:

```
sage: M = matroids.catalog.Pappus().contract(['c'])
sage: sorted(M.groundset())
['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i']
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Pappus().contract(['c'])
>>> sorted(M.groundset())
['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i']
```

CHAPTER

SIX

ADVANCED FUNCTIONALITY

6.1 Advanced matroid functionality

This module collects a number of advanced functions which are not directly available to the end user by default. To import them into the main namespace, type:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

This adds the following to the main namespace:

- Matroid classes:
  - *BasisMatroid*
  - *CircuitsMatroid*
  - *CircuitClosuresMatroid*
  - *DualMatroid*
  - *FlatsMatroid*
  - *GraphicMatroid*
  - *LinearMatroid*
  - *RegularMatroid*
  - *BinaryMatroid*
  - *TernaryMatroid*
  - *QuaternaryMatroid*
  - *MinorMatroid*
  - *RankMatroid*

  Note that you can construct all of these through the *Matroid()* function, which is available on startup. Using the classes directly can sometimes be useful for faster code (e.g. if your code calls `Matroid()` frequently).

- Other classes:
  - *LinearSubclasses*
  - *MatroidExtensions*

Instances of these classes are returned by the methods *Matroid.linear_subclasses()* and *Matroid.extensions()*.

- Useful functions:
  - *setprint()*
  - *newlabel()*
  - *get_nonisomorphic_matroids()*
  - *lift_cross_ratios()*
  - *lift_map()*
  - *cmp_elements_key()*

AUTHORS:

- Stefan van Zwam (2013-04-01): initial version

6.2 Iterators for linear subclasses

The classes below are iterators returned by the functions *M.linear_subclasses()* and *M.extensions()*. See the documentation of these methods for more detail. For direct access to these classes, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See also *sage.matroids.advanced*.

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

6.2.1 Methods

class sage.matroids.extension.**CutNode**

Bases: object

An internal class used for creating linear subclasses of a matroids in a depth-first manner.

A linear subclass is a set of hyperplanes mc with the property that certain sets of hyperplanes must either be fully contained in mc or intersect mc in at most 1 element. The way we generate them is by a depth-first search. This class represents a node in the search tree.

It contains the set of hyperplanes selected so far, as well as a collection of hyperplanes whose insertion has been explored elsewhere in the search tree.

The class has methods for selecting a hyperplane to insert, for inserting hyperplanes and closing the set to become a linear subclass again, and for adding a hyperplane to the set of *forbidden* hyperplanes, and similarly closing that set.

class sage.matroids.extension.**LinearSubclasses**

Bases: object

An iterable set of linear subclasses of a matroid.

Enumerate linear subclasses of a given matroid. A *linear subclass* is a collection of hyperplanes (flats of rank $r-1$ where r is the rank of the matroid) with the property that no modular triple of hyperplanes has exactly two members in the linear subclass. A triple of hyperplanes in a matroid of rank r is *modular* if its intersection has rank $r-2$.

INPUT:

- `M` – matroid
- `line_length` – integer (default: `None`)
- `subsets` – (default: `None`) a set of subsets of the groundset of `M`
- `splice` – (default: `None`) a matroid N such that for some $e \in E(N)$ and some $f \in E(M)$, we have $N \setminus e = M \setminus f$

OUTPUT: an enumerator for the linear subclasses of M

If `line_length` is not `None`, the enumeration is restricted to linear subclasses `mc` so containing at least one of each set of `line_length` hyperplanes which have a common intersection of rank $r-2$.

If `subsets` is not `None`, the enumeration is restricted to linear subclasses `mc` containing all hyperplanes which fully contain some set from `subsets`.

If `splice` is not `None`, then the enumeration is restricted to linear subclasses mc such that if M' is the extension of M by e that arises from mc, then $M' \setminus f = N$.

EXAMPLES:

```
sage: from sage.matroids.extension import LinearSubclasses
sage: M = matroids.Uniform(3, 6)
sage: len([mc for mc in LinearSubclasses(M)])
83
sage: len([mc for mc in LinearSubclasses(M, line_length=5)])
22
sage: for mc in LinearSubclasses(M, subsets=[[0, 1], [2, 3], [4, 5]]):
....:     print(len(mc))
3
15
```

```
>>> from sage.all import *
>>> from sage.matroids.extension import LinearSubclasses
>>> M = matroids.Uniform(Integer(3), Integer(6))
>>> len([mc for mc in LinearSubclasses(M)])
83
>>> len([mc for mc in LinearSubclasses(M, line_length=Integer(5))])
22
>>> for mc in LinearSubclasses(M, subsets=[[Integer(0), Integer(1)], [Integer(2),␣
↪Integer(3)], [Integer(4), Integer(5)]]):
...     print(len(mc))
3
15
```

Note that this class is intended for runtime, internal use, so no loads/dumps mechanism was implemented.

class sage.matroids.extension.**LinearSubclassesIter**

Bases: `object`

An iterator for a set of linear subclass.

class sage.matroids.extension.**MatroidExtensions**

Bases: *LinearSubclasses*

An iterable set of single-element extensions of a given matroid.

INPUT:

- `M` – matroid
- `e` – an element
- `line_length` – integer (default: `None`)
- `subsets` – (default: `None`) a set of subsets of the groundset of `M`
- `splice` – a matroid N such that for some $f \in E(M)$, we have $N \setminus e = M \setminus f$

OUTPUT:

An enumerator for the extensions of `M` to a matroid `N` so that $N \setminus e = M$. If `line_length` is not `None`, the enumeration is restricted to extensions N without $U(2, k)$-minors, where `k > line_length`.

If `subsets` is not `None`, the enumeration is restricted to extensions N of M by element e so that all hyperplanes of M which fully contain some set from `subsets`, will also span e.

If `splice` is not `None`, then the enumeration is restricted to extensions M' such that $M' \setminus f = N$, where $E(M) \setminus E(N) = \{f\}$.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.Uniform(3, 6)
sage: len([N for N in MatroidExtensions(M, 'x')])
83
sage: len([N for N in MatroidExtensions(M, 'x', line_length=5)])
22
sage: for N in MatroidExtensions(M, 'x', subsets=[[0, 1], [2, 3],
....:                                             [4, 5]]): print(N)
Matroid of rank 3 on 7 elements with 32 bases
Matroid of rank 3 on 7 elements with 20 bases
sage: M = BasisMatroid(matroids.catalog.BetsyRoss()); M
Matroid of rank 3 on 11 elements with 140 bases
sage: e = 'k'; f = 'h'; Me = M.delete(e); Mf=M.delete(f)
sage: for N in MatroidExtensions(Mf, f, splice=Me): print(N)
Matroid of rank 3 on 11 elements with 141 bases
Matroid of rank 3 on 11 elements with 140 bases
sage: for N in MatroidExtensions(Me, e, splice=Mf): print(N)
Matroid of rank 3 on 11 elements with 141 bases
Matroid of rank 3 on 11 elements with 140 bases
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = matroids.Uniform(Integer(3), Integer(6))
>>> len([N for N in MatroidExtensions(M, 'x')])
83
>>> len([N for N in MatroidExtensions(M, 'x', line_length=Integer(5))])
22
>>> for N in MatroidExtensions(M, 'x', subsets=[[Integer(0), Integer(1)],␣
↪[Integer(2), Integer(3)],
```

(continues on next page)

(continued from previous page)

```
...                                                 [Integer(4), Integer(5)]]):␣
↪print(N)
Matroid of rank 3 on 7 elements with 32 bases
Matroid of rank 3 on 7 elements with 20 bases
>>> M = BasisMatroid(matroids.catalog.BetsyRoss()); M
Matroid of rank 3 on 11 elements with 140 bases
>>> e = 'k'; f = 'h'; Me = M.delete(e); Mf=M.delete(f)
>>> for N in MatroidExtensions(Mf, f, splice=Me): print(N)
Matroid of rank 3 on 11 elements with 141 bases
Matroid of rank 3 on 11 elements with 140 bases
>>> for N in MatroidExtensions(Me, e, splice=Mf): print(N)
Matroid of rank 3 on 11 elements with 141 bases
Matroid of rank 3 on 11 elements with 140 bases
```

Note that this class is intended for runtime, internal use, so no loads/dumps mechanism was implemented.

6.3 Some useful functions for the matroid class.

For direct access to the methods *newlabel()*, *setprint()* and *get_nonisomorphic_matroids()*, type:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

See also *sage.matroids.advanced*.

AUTHORS:

- Stefan van Zwam (2011-06-24): initial version

sage.matroids.utilities.**cmp_elements_key**(*x*)

A helper function to compare elements which may be integers or strings.

EXAMPLES:

```
sage: from sage.matroids.utilities import cmp_elements_key
sage: l = ['a', 'b', 1, 3, 2, 10, 111, 100, 'c', 'aa']
sage: sorted(l, key=cmp_elements_key)
[1, 2, 3, 10, 100, 111, 'a', 'aa', 'b', 'c']
```

```
>>> from sage.all import *
>>> from sage.matroids.utilities import cmp_elements_key
>>> l = ['a', 'b', Integer(1), Integer(3), Integer(2), Integer(10), Integer(111),␣
↪Integer(100), 'c', 'aa']
>>> sorted(l, key=cmp_elements_key)
[1, 2, 3, 10, 100, 111, 'a', 'aa', 'b', 'c']
```

sage.matroids.utilities.**get_nonisomorphic_matroids**(*MSet*)

Return non-isomorphic members of the matroids in set `MSet`.

For direct access to `get_nonisomorphic_matroids`, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

INPUT:

- `MSet` – an iterable whose members are matroids

OUTPUT:

A list containing one representative of each isomorphism class of members of `MSet`.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: L = matroids.Uniform(3, 5).extensions()
sage: len(list(L))
32
sage: len(get_nonisomorphic_matroids(L))
5
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> L = matroids.Uniform(Integer(3), Integer(5)).extensions()
>>> len(list(L))
32
>>> len(get_nonisomorphic_matroids(L))
5
```

`sage.matroids.utilities.`**`lift_cross_ratios`**(*A*, *lift_map=None*)

Return a matrix which arises from the given matrix by lifting cross ratios.

INPUT:

- `A` – a matrix over a ring `source_ring`
- `lift_map` – a Python dictionary, mapping each cross ratio of `A` to some element of a target ring, and such that `lift_map[source_ring(1)] = target_ring(1)`

OUTPUT: `Z` – a matrix over the ring `target_ring`

The intended use of this method is to create a (reduced) matrix representation of a matroid `M` over a ring `target_ring`, given a (reduced) matrix representation of `A` of `M` over a ring `source_ring` and a map `lift_map` from `source_ring` to `target_ring`.

This method will create a unique candidate representation `Z`, but will not verify if `Z` is indeed a representation of `M`. However, this is guaranteed if the conditions of the lift theorem (see [PvZ2010]) hold for the lift map in combination with the matrix `A`.

For a lift map f and a matrix A these conditions are as follows. First of all $f : S \to T$, where S is a set of invertible elements of the source ring and T is a set of invertible elements of the target ring. The matrix A has entries from the source ring, and each cross ratio of A is contained in S. Moreover:

- $1 \in S, 1 \in T$;
- for all $x \in S$: $f(x) = 1$ if and only if $x = 1$;
- for all $x, y \in S$: if $x + y = 0$ then $f(x) + f(y) = 0$;

- for all $x, y \in S$: if $x + y = 1$ then $f(x) + f(y) = 1$;
- for all $x, y, z \in S$: if $xy = z$ then $f(x)f(y) = f(z)$.

Any ring homomorphism $h : P \to R$ induces a lift map from the set of units S of P to the set of units T of R. There exist lift maps which do not arise in this manner. Several such maps can be created by the function *lift_map()*.

See also

lift_map()

EXAMPLES:

```
sage: # needs sage.graphs
sage: from sage.matroids.advanced import lift_cross_ratios, lift_map,␣
↪LinearMatroid
sage: R = GF(7)
sage: to_sixth_root_of_unity = lift_map('sru')                                  #␣
↪needs sage.rings.number_field
sage: A = Matrix(R, [[1, 0, 6, 1, 2],[6, 1, 0, 0, 1],[0, 6, 3, 6, 0]])
sage: A
[1 0 6 1 2]
[6 1 0 0 1]
[0 6 3 6 0]
sage: Z = lift_cross_ratios(A, to_sixth_root_of_unity)                          #␣
↪needs sage.rings.finite_rings sage.rings.number_field
sage: Z                                                                         #␣
↪needs sage.rings.finite_rings sage.rings.number_field
[     1      0      1      1      1]
[-z + 1      1      0      0      1]
[     0     -1      1 -z + 1      0]
sage: M = LinearMatroid(reduced_matrix=A)
sage: sorted(M.cross_ratios())
[3, 5]
sage: N = LinearMatroid(reduced_matrix=Z)                                       #␣
↪needs sage.rings.finite_rings sage.rings.number_field
sage: sorted(N.cross_ratios())                                                  #␣
↪needs sage.rings.finite_rings sage.rings.number_field
[-z + 1, z]
sage: M.is_isomorphism(N, {e:e for e in M.groundset()})                         #␣
↪needs sage.rings.finite_rings sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> from sage.matroids.advanced import lift_cross_ratios, lift_map, LinearMatroid
>>> R = GF(Integer(7))
>>> to_sixth_root_of_unity = lift_map('sru')                                    #␣
↪needs sage.rings.number_field
>>> A = Matrix(R, [[Integer(1), Integer(0), Integer(6), Integer(1), Integer(2)],
↪[Integer(6), Integer(1), Integer(0), Integer(0), Integer(1)],[Integer(0),␣
↪Integer(6), Integer(3), Integer(6), Integer(0)]])
```

(continues on next page)

(continued from previous page)

```
>>> A
[1 0 6 1 2]
[6 1 0 0 1]
[0 6 3 6 0]
>>> Z = lift_cross_ratios(A, to_sixth_root_of_unity)                    #␣
↪needs sage.rings.finite_rings sage.rings.number_field
>>> Z                                                                   #␣
↪needs sage.rings.finite_rings sage.rings.number_field
[     1      0      1      1      1]
[-z + 1      1      0      0      1]
[     0     -1      1 -z + 1      0]
>>> M = LinearMatroid(reduced_matrix=A)
>>> sorted(M.cross_ratios())
[3, 5]
>>> N = LinearMatroid(reduced_matrix=Z)                                 #␣
↪needs sage.rings.finite_rings sage.rings.number_field
>>> sorted(N.cross_ratios())                                           #␣
↪needs sage.rings.finite_rings sage.rings.number_field
[-z + 1, z]
>>> M.is_isomorphism(N, {e:e for e in M.groundset()})                   #␣
↪needs sage.rings.finite_rings sage.rings.number_field
True
```

sage.matroids.utilities.**lift_map**(*target*)

Create a lift map, to be used for lifting the cross ratios of a matroid representation.

See also

lift_cross_ratios()

INPUT:

- `target` – string describing the target (partial) field

OUTPUT: dictionary

Depending on the value of `target`, the following lift maps will be created:

- "reg": a lift map from $\mathbf{F}_3$ to the regular partial field $(\mathbf{Z}, < -1 >)$.
- "sru": a lift map from $\mathbf{F}_7$ to the sixth-root-of-unity partial field $(\mathbf{Q}(z), < z >)$, where z is a sixth root of unity. The map sends 3 to z.
- "dyadic": a lift map from $\mathbf{F}_{11}$ to the dyadic partial field $(\mathbf{Q}, < -1, 2 >)$.
- "gm": a lift map from $\mathbf{F}_{19}$ to the golden mean partial field $(\mathbf{Q}(t), < -1, t >)$, where t is a root of $t^2 - t - 1$. The map sends 5 to t.

The example below shows that the latter map satisfies three necessary conditions stated in *lift_cross_ratios()*

EXAMPLES:

```
sage: from sage.matroids.utilities import lift_map
sage: lm = lift_map('gm')                                               #␣
↪needs sage.rings.finite_rings sage.rings.number_field
sage: for x in lm:                                                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings sage.rings.number_field
....:     if (x == 1) is not (lm[x] == 1):
....:         print('not a proper lift map')
....:     for y in lm:
....:         if (x+y == 0) and not (lm[x]+lm[y] == 0):
....:             print('not a proper lift map')
....:         if (x+y == 1) and not (lm[x]+lm[y] == 1):
....:             print('not a proper lift map')
....:         for z in lm:
....:             if (x*y==z) and not (lm[x]*lm[y]==lm[z]):
....:                 print('not a proper lift map')
```

```
>>> from sage.all import *
>>> from sage.matroids.utilities import lift_map
>>> lm = lift_map('gm')                                                    #␣
↪needs sage.rings.finite_rings sage.rings.number_field
>>> for x in lm:                                                           #␣
↪needs sage.rings.finite_rings sage.rings.number_field
...     if (x == Integer(1)) is not (lm[x] == Integer(1)):
...         print('not a proper lift map')
...     for y in lm:
...         if (x+y == Integer(0)) and not (lm[x]+lm[y] == Integer(0)):
...             print('not a proper lift map')
...         if (x+y == Integer(1)) and not (lm[x]+lm[y] == Integer(1)):
...             print('not a proper lift map')
...         for z in lm:
...             if (x*y==z) and not (lm[x]*lm[y]==lm[z]):
...                 print('not a proper lift map')
```

sage.matroids.utilities.**make_regular_matroid_from_matroid**(*matroid*)

Attempt to construct a regular representation of a matroid.

INPUT:

- `matroid` – matroid

OUTPUT:

Return a $(0, 1, -1)$-matrix over the integers such that, if the input is a regular matroid, then the output is a totally unimodular matrix representing that matroid.

EXAMPLES:

```
sage: from sage.matroids.utilities import make_regular_matroid_from_matroid
sage: make_regular_matroid_from_matroid(                                   #␣
↪needs sage.graphs
....:               matroids.CompleteGraphic(6)).is_isomorphic(
....:                                   matroids.CompleteGraphic(6))
True
```

```
>>> from sage.all import *
>>> from sage.matroids.utilities import make_regular_matroid_from_matroid
>>> make_regular_matroid_from_matroid(                                     #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
...                matroids.CompleteGraphic(Integer(6))).is_isomorphic(
...                                      matroids.CompleteGraphic(Integer(6)))
True
```

sage.matroids.utilities.**newlabel**(*groundset*)

Create a new element label different from the labels in `groundset`.

INPUT:

- `groundset` – set of objects

OUTPUT: string not in the set `groundset`

For direct access to `newlabel`, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

ALGORITHM:

1. Create a set of all one-character alphanumeric strings.
2. Remove the string representation of each existing element from this list.
3. If the list is nonempty, return any element.
4. Otherwise, return the concatenation of the strings of each existing element, preceded by 'e'.

EXAMPLES:

```
sage: from sage.matroids.advanced import newlabel
sage: S = set(['a', 42, 'b'])
sage: newlabel(S) in S
False

sage: S = set('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789')
sage: t = newlabel(S)
sage: len(t)
63
sage: t[0]
'e'
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import newlabel
>>> S = set(['a', Integer(42), 'b'])
>>> newlabel(S) in S
False

>>> S = set('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789')
>>> t = newlabel(S)
>>> len(t)
63
>>> t[Integer(0)]
'e'
```

sage.matroids.utilities.**sanitize_contractions_deletions**(*matroid*, *contractions*, *deletions*)

Return a fixed version of sets `contractions` and `deletions`.

INPUT:

- `matroid` – a *Matroid* instance
- `contractions` – a subset of the groundset
- `deletions` – a subset of the groundset

OUTPUT: an independent set `C` and a coindependent set `D` such that

```
matroid / contractions \ deletions == matroid / C \ D
```

Raise an error if either is not a subset of the groundset of `matroid` or if they are not disjoint.

This function is used by the *Matroid.minor()* method.

EXAMPLES:

```
sage: from sage.matroids.utilities import setprint
sage: from sage.matroids.utilities import sanitize_contractions_deletions
sage: M = matroids.catalog.Fano()
sage: setprint(sanitize_contractions_deletions(M, 'abc', 'defg'))
[{'a', 'b', 'c'}, {'d', 'e', 'f', 'g'}]
sage: setprint(sanitize_contractions_deletions(M, 'defg', 'abc'))
[{'a', 'b', 'c', 'f'}, {'d', 'e', 'g'}]
sage: setprint(sanitize_contractions_deletions(M, [1, 2, 3], 'efg'))
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not a subset of the groundset
sage: setprint(sanitize_contractions_deletions(M, 'efg', [1, 2, 3]))
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not a subset of the groundset
sage: setprint(sanitize_contractions_deletions(M, 'ade', 'efg'))
Traceback (most recent call last):
...
ValueError: contraction and deletion sets are not disjoint.
```

```
>>> from sage.all import *
>>> from sage.matroids.utilities import setprint
>>> from sage.matroids.utilities import sanitize_contractions_deletions
>>> M = matroids.catalog.Fano()
>>> setprint(sanitize_contractions_deletions(M, 'abc', 'defg'))
[{'a', 'b', 'c'}, {'d', 'e', 'f', 'g'}]
>>> setprint(sanitize_contractions_deletions(M, 'defg', 'abc'))
[{'a', 'b', 'c', 'f'}, {'d', 'e', 'g'}]
>>> setprint(sanitize_contractions_deletions(M, [Integer(1), Integer(2),␣
↪Integer(3)], 'efg'))
Traceback (most recent call last):
...
ValueError: [1, 2, 3] is not a subset of the groundset
>>> setprint(sanitize_contractions_deletions(M, 'efg', [Integer(1), Integer(2),␣
↪Integer(3)]))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: [1, 2, 3] is not a subset of the groundset
>>> setprint(sanitize_contractions_deletions(M, 'ade', 'efg'))
Traceback (most recent call last):
...
ValueError: contraction and deletion sets are not disjoint.
```

sage.matroids.utilities.**setprint**(*X*)

Print nested data structures nicely.

Python's data structures `set` and `frozenset` do not print nicely. This function can be used as replacement for `print` to overcome this. For direct access to `setprint`, run:

```
sage: from sage.matroids.advanced import *
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
```

Note

This function will be redundant when Sage moves to Python 3, since the default `print` will suffice then.

INPUT:

- `X` – any Python object

OUTPUT: none; however, the function prints a nice representation of `X`

EXAMPLES:

Output looks much better:

```
sage: from sage.matroids.advanced import setprint
sage: L = [{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {4, 1, 3}]
sage: print(L)
[{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {1, 3, 4}]
sage: setprint(L)
[{1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}]
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> L = [{Integer(1), Integer(2), Integer(3)}, {Integer(1), Integer(2),␣
↪Integer(4)}, {Integer(2), Integer(3), Integer(4)}, {Integer(4), Integer(1),␣
↪Integer(3)}]
>>> print(L)
[{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {1, 3, 4}]
>>> setprint(L)
[{1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}]
```

Note that for iterables, the effect can be undesirable:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Fano().delete('efg')
```

(continues on next page)

(continued from previous page)

```
sage: M.bases()
SetSystem of 3 sets over 4 elements
sage: setprint(M.bases())
[{'a', 'b', 'c'}, {'a', 'b', 'd'}, {'a', 'c', 'd'}]
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> M = matroids.catalog.Fano().delete('efg')
>>> M.bases()
SetSystem of 3 sets over 4 elements
>>> setprint(M.bases())
[{'a', 'b', 'c'}, {'a', 'b', 'd'}, {'a', 'c', 'd'}]
```

An exception was made for subclasses of SageObject:

```
sage: from sage.matroids.advanced import setprint
sage: G = graphs.PetersenGraph()                                        #␣
↪needs sage.graphs
sage: list(G)                                                           #␣
↪needs sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: setprint(G)                                                       #␣
↪needs sage.graphs
Petersen graph: Graph on 10 vertices
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import setprint
>>> G = graphs.PetersenGraph()                                          #␣
↪needs sage.graphs
>>> list(G)                                                             #␣
↪needs sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> setprint(G)                                                         #␣
↪needs sage.graphs
Petersen graph: Graph on 10 vertices
```

sage.matroids.utilities.**setprint_s**(*X*, *toplevel=False*)

Create the string for use by `setprint()`.

INPUT:

- `X` – any Python object
- `toplevel` – boolean (default: `False`); indicates whether this is a recursion or not

OUTPUT:

A string representation of the object, with nice notation for sets and frozensets.

EXAMPLES:

```
sage: from sage.matroids.utilities import setprint_s
sage: L = [{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {4, 1, 3}]
sage: setprint_s(L)
'[{1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}]'
```

```
>>> from sage.all import *
>>> from sage.matroids.utilities import setprint_s
>>> L = [{Integer(1), Integer(2), Integer(3)}, {Integer(1), Integer(2),␣
↪Integer(4)}, {Integer(2), Integer(3), Integer(4)}, {Integer(4), Integer(1),␣
↪Integer(3)}]
>>> setprint_s(L)
'[{1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}]'
```

The `toplevel` argument only affects strings, to mimic `print`'s behavior:

```
sage: X = 'abcd'
sage: setprint_s(X)
"'abcd'"
sage: setprint_s(X, toplevel=True)
'abcd'
```

```
>>> from sage.all import *
>>> X = 'abcd'
>>> setprint_s(X)
"'abcd'"
>>> setprint_s(X, toplevel=True)
'abcd'
```

sage.matroids.utilities.**spanning_forest**(*M*)

Return a list of edges of a spanning forest of the bipartite graph defined by M

INPUT:

- `M` – a matrix defining a bipartite graph G. The vertices are the rows and columns, if $M[i, j]$ is nonzero, then there is an edge between row i and column j.

OUTPUT:

A list of tuples (r_i, c_i) representing edges between row r_i and column c_i.

EXAMPLES:

```
sage: from sage.matroids.utilities import spanning_forest
sage: len(spanning_forest(matrix([[1,1,1],[1,1,1],[1,1,1]])))                  #␣
↪needs sage.graphs
5
sage: len(spanning_forest(matrix([[0,0,1],[0,1,0],[0,1,0]])))                  #␣
↪needs sage.graphs
3
```

```
>>> from sage.all import *
>>> from sage.matroids.utilities import spanning_forest
>>> len(spanning_forest(matrix([[Integer(1),Integer(1),Integer(1)],[Integer(1),
↪Integer(1),Integer(1)],[Integer(1),Integer(1),Integer(1)]])))
↪# needs sage.graphs
5
>>> len(spanning_forest(matrix([[Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(1),Integer(0)]])))
↪# needs sage.graphs
3
```

sage.matroids.utilities.**spanning_stars**(*M*)

Return the edges of a connected subgraph that is a union of all edges incident some subset of vertices.

INPUT:

- M – a matrix defining a bipartite graph G. The vertices are the rows and columns, if $M[i, j]$ is nonzero, then there is an edge between row i and column 0.

OUTPUT:

A list of tuples $(row, column)$ in a spanning forest of the bipartite graph defined by M.

EXAMPLES:

```
sage: from sage.matroids.utilities import spanning_stars
sage: edges = spanning_stars(matrix([[1,1,1],[1,1,1],[1,1,1]]))                #␣
↪needs sage.graphs
sage: Graph([(x+3, y) for x,y in edges]).is_connected()                        #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> from sage.matroids.utilities import spanning_stars
>>> edges = spanning_stars(matrix([[Integer(1),Integer(1),Integer(1)],[Integer(1),
↪Integer(1),Integer(1)],[Integer(1),Integer(1),Integer(1)]]))                  #␣
↪needs sage.graphs
>>> Graph([(x+Integer(3), y) for x,y in edges]).is_connected()                  ␣
↪     # needs sage.graphs
True
```

sage.matroids.utilities.**split_vertex**(*G*, *u*, *v=None*, *edges=None*)

Split a vertex in a graph.

If an edge is inserted between the vertices after splitting, this corresponds to a graphic coextension of a matroid.

INPUT:

- G – a SageMath Graph
- u – a vertex in G
- v – (optional) the name of the new vertex after the splitting. If v is specified and already in the graph, it must be an isolated vertex.
- edges – (optional) iterable container of edges on u that move to v after the splitting. If None, v will be an isolated vertex. The edge labels must be specified.

EXAMPLES:

```
sage: # needs sage.graphs
sage: from sage.matroids.utilities import split_vertex
sage: G = graphs.BullGraph()
sage: split_vertex(G, u=1, v=55, edges=[(1, 3)])
Traceback (most recent call last):
...
ValueError: the edges are not all incident with u
sage: split_vertex(G, u=1, v=55, edges=[(1, 3, None)])
sage: list(G.edges(sort=True))
[(0, 1, None), (0, 2, None), (1, 2, None), (2, 4, None), (3, 55, None)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> from sage.matroids.utilities import split_vertex
>>> G = graphs.BullGraph()
>>> split_vertex(G, u=Integer(1), v=Integer(55), edges=[(Integer(1), Integer(3))])
Traceback (most recent call last):
...
ValueError: the edges are not all incident with u
>>> split_vertex(G, u=Integer(1), v=Integer(55), edges=[(Integer(1), Integer(3),␣
↪None)])
>>> list(G.edges(sort=True))
[(0, 1, None), (0, 2, None), (1, 2, None), (2, 4, None), (3, 55, None)]
```

CHAPTER

SEVEN

INTERNALS

7.1 Lean matrices

Internal data structures for the `LinearMatroid` class and some subclasses. Note that many of the methods are `cdef`, and therefore only available from Cython code.

Warning

Intended for internal use by the `LinearMatroid` classes only. End users should work with Sage matrices instead. Methods that are used outside lean_matrix.pyx and have no equivalent in Sage's `Matrix` have been flagged in the code, as well as where they are used, by `# Not a Sage matrix operation` or `# Deprecated Sage matrix operation`.

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

class sage.matroids.lean_matrix.**BinaryMatrix**

Bases: *LeanMatrix*

Binary matrix class. Entries are stored bit-packed into integers.

INPUT:

- `m` – number of rows
- `n` – number of columns
- `M` – (default: `None`) `Matrix` or `BinaryMatrix` instance. Assumption: dimensions of `M` are at most `m` by `n`.
- `ring` – (default: `None`) ignored

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(2, 2, Matrix(GF(7), [[0, 0], [0, 0]]))
sage: B = BinaryMatrix(2, 2, ring=GF(5))
sage: C = BinaryMatrix(2, 2)
sage: A == B and A == C
True
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = BinaryMatrix(Integer(2), Integer(2), Matrix(GF(Integer(7)), [[Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)], [Integer(0), Integer(0)]]))
>>> B = BinaryMatrix(Integer(2), Integer(2), ring=GF(Integer(5)))
>>> C = BinaryMatrix(Integer(2), Integer(2))
>>> A == B and A == C
True
```

base_ring()

Return $GF(2)$.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(4, 4)
sage: A.base_ring()
Finite Field of size 2
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = BinaryMatrix(Integer(4), Integer(4))
>>> A.base_ring()
Finite Field of size 2
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(3, 4)
sage: A.characteristic()
2
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = BinaryMatrix(Integer(3), Integer(4))
>>> A.characteristic()
2
```

class sage.matroids.lean_matrix.**GenericMatrix**

Bases: *LeanMatrix*

Matrix over arbitrary Sage ring.

INPUT:

- `nrows` – number of rows
- `ncols` – number of columns
- `M` – (default: `None`) a `Matrix` or `GenericMatrix` of dimensions at most `m*n`
- `ring` – (default: `None`) a Sage ring

Note

This class is intended for internal use by the LinearMatroid class only. Hence it does not derive from `SageObject`. If `A` is a LeanMatrix instance, and you need access from other parts of Sage, use `Matrix(A)` instead.

If the constructor is fed a GenericMatrix instance, the `ring` argument is ignored. Otherwise, the matrix entries will be converted to the appropriate ring.

EXAMPLES:

```
sage: M = Matroid(ring=GF(5), matrix=[[1, 0, 1, 1, 1], [0, 1, 1, 2, 3]])  #␣
↪indirect doctest
sage: M.is_isomorphic(matroids.Uniform(2, 5))
True
```

```
>>> from sage.all import *
>>> M = Matroid(ring=GF(Integer(5)), matrix=[[Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2),␣
↪Integer(3)]])  # indirect doctest
>>> M.is_isomorphic(matroids.Uniform(Integer(2), Integer(5)))
True
```

base_ring()

Return the base ring of `self`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import GenericMatrix
sage: A = GenericMatrix(3, 4, ring=GF(5))
sage: A.base_ring()
Finite Field of size 5
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import GenericMatrix
>>> A = GenericMatrix(Integer(3), Integer(4), ring=GF(Integer(5)))
>>> A.base_ring()
Finite Field of size 5
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import GenericMatrix
sage: A = GenericMatrix(3, 4, ring=GF(5))
sage: A.characteristic()
5
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import GenericMatrix
>>> A = GenericMatrix(Integer(3), Integer(4), ring=GF(Integer(5)))
>>> A.characteristic()
5
```

class sage.matroids.lean_matrix.**LeanMatrix**

Bases: object

Lean matrices

Sage's matrix classes are powerful, versatile, and often very fast. However, their performance with regard to pivoting (pretty much the only task performed on them within the context of matroids) leaves something to be desired. The LeanMatrix classes provide the LinearMatroid classes with a custom, light-weight data structure to store and manipulate matrices.

This is the abstract base class. Most methods are not implemented; this is only to fix the interface.

Note

This class is intended for internal use by the LinearMatroid class only. Hence it does not derive from SageObject. If A is a LeanMatrix instance, and you need access from other parts of Sage, use Matrix(A) instead.

EXAMPLES:

```
sage: M = Matroid(ring=GF(5), matrix=[[1, 0, 1, 1, 1], [0, 1, 1, 2, 3]])  #␣
↪indirect doctest
sage: M.is_isomorphic(matroids.Uniform(2, 5))
True
```

```
>>> from sage.all import *
>>> M = Matroid(ring=GF(Integer(5)), matrix=[[Integer(1), Integer(0), Integer(1),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(1), Integer(2),␣
↪Integer(3)]])  # indirect doctest
>>> M.is_isomorphic(matroids.Uniform(Integer(2), Integer(5)))
True
```

base_ring()

Return the base ring.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import LeanMatrix
sage: A = LeanMatrix(3, 4)
sage: A.base_ring()
Traceback (most recent call last):
...
NotImplementedError: subclasses need to implement this.
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import LeanMatrix
>>> A = LeanMatrix(Integer(3), Integer(4))
>>> A.base_ring()
Traceback (most recent call last):
...
NotImplementedError: subclasses need to implement this.
```

characteristic()

Return the characteristic of self.base_ring().

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import GenericMatrix
sage: A = GenericMatrix(3, 4, ring=GF(5))
sage: A.characteristic()
5
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import GenericMatrix
>>> A = GenericMatrix(Integer(3), Integer(4), ring=GF(Integer(5)))
>>> A.characteristic()
5
```

ncols()

Return the number of columns.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import LeanMatrix
sage: A = LeanMatrix(3, 4)
sage: A.ncols()
4
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import LeanMatrix
>>> A = LeanMatrix(Integer(3), Integer(4))
>>> A.ncols()
4
```

nrows()

Return the number of rows.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import LeanMatrix
sage: A = LeanMatrix(3, 4)
sage: A.nrows()
3
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import LeanMatrix
>>> A = LeanMatrix(Integer(3), Integer(4))
>>> A.nrows()
3
```

class sage.matroids.lean_matrix.**PlusMinusOneMatrix**

Bases: *LeanMatrix*

Matrix with nonzero entries of ± 1.

INPUT:

- `nrows` – number of rows
- `ncols` – number of columns
- `M` – (default: `None`) a `Matrix` or `GenericMatrix` of dimensions at most `m*n`

Note

This class is intended for internal use by the *LinearMatroid* class only. Hence it does not derive from `SageObject`. If `A` is a *LeanMatrix* instance, and you need access from other parts of Sage, use `Matrix(A)` instead.

This class is mainly intended for use with the *RegularMatroid* class, so entries are assumed to be ± 1 or 0. No overflow checking takes place!

EXAMPLES:

```
sage: M = Matroid(graphs.CompleteGraph(4).incidence_matrix(oriented=True),  #␣
↪indirect doctest
....:             regular=True)
sage: M.is_isomorphic(matroids.Wheel(3))
True
```

```
>>> from sage.all import *
>>> M = Matroid(graphs.CompleteGraph(Integer(4)).incidence_matrix(oriented=True),␣
↪ # indirect doctest
...              regular=True)
>>> M.is_isomorphic(matroids.Wheel(Integer(3)))
True
```

base_ring()

Return the base ring of `self`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import PlusMinusOneMatrix
sage: A = PlusMinusOneMatrix(3, 4)
sage: A.base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import PlusMinusOneMatrix
>>> A = PlusMinusOneMatrix(Integer(3), Integer(4))
>>> A.base_ring()
Integer Ring
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import PlusMinusOneMatrix
sage: A = PlusMinusOneMatrix(3, 4)
sage: A.characteristic()
0
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import PlusMinusOneMatrix
>>> A = PlusMinusOneMatrix(Integer(3), Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> A.characteristic()
0
```

class sage.matroids.lean_matrix.**QuaternaryMatrix**

Bases: *LeanMatrix*

Matrices over GF(4).

INPUT:

- `m` – number of rows
- `n` – number of columns
- `M` – (default: `None`) `QuaternaryMatrix` or `LeanMatrix` or (Sage) `Matrix` instance. If not given, new matrix will be filled with zeroes. Assumption: `M` has dimensions at most `m` times `n`.
- `ring` – (default: `None`) a copy of GF(4); useful for specifying generator name

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = QuaternaryMatrix(2, 2, Matrix(GF(4, 'x'), [[0, 0], [0, 0]]))
sage: B = QuaternaryMatrix(2, 2, GenericMatrix(2, 2, ring=GF(4, 'x')))
sage: C = QuaternaryMatrix(2, 2, ring=GF(4, 'x'))
sage: A == B and A == C
True
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = QuaternaryMatrix(Integer(2), Integer(2), Matrix(GF(Integer(4), 'x'),␣
↪[[Integer(0), Integer(0)], [Integer(0), Integer(0)]]))
>>> B = QuaternaryMatrix(Integer(2), Integer(2), GenericMatrix(Integer(2),␣
↪Integer(2), ring=GF(Integer(4), 'x')))
>>> C = QuaternaryMatrix(Integer(2), Integer(2), ring=GF(Integer(4), 'x'))
>>> A == B and A == C
True
```

base_ring()

Return copy of $GF(4)$ with appropriate generator.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = QuaternaryMatrix(2, 2, ring=GF(4, 'f'))
sage: A.base_ring()
Finite Field in f of size 2^2
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = QuaternaryMatrix(Integer(2), Integer(2), ring=GF(Integer(4), 'f'))
>>> A.base_ring()
Finite Field in f of size 2^2
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = QuaternaryMatrix(200, 5000, ring=GF(4, 'x'))
sage: A.characteristic()
2
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = QuaternaryMatrix(Integer(200), Integer(5000), ring=GF(Integer(4), 'x
↪'))
>>> A.characteristic()
2
```

class sage.matroids.lean_matrix.**RationalMatrix**

Bases: *LeanMatrix*

Matrix over the rationals.

INPUT:

- `nrows` – number of rows
- `ncols` – number of columns
- `M` – (default: `None`) a `Matrix` or `GenericMatrix` of dimensions at most `m * n`

EXAMPLES:

```
sage: M = Matroid(graphs.CompleteGraph(4).incidence_matrix(oriented=True))  #␣
↪indirect doctest
sage: M.is_isomorphic(matroids.Wheel(3))
True
```

```
>>> from sage.all import *
>>> M = Matroid(graphs.CompleteGraph(Integer(4)).incidence_matrix(oriented=True))␣
↪ # indirect doctest
>>> M.is_isomorphic(matroids.Wheel(Integer(3)))
True
```

base_ring()

Return the base ring of `self`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import RationalMatrix
sage: A = RationalMatrix(3, 4)
sage: A.base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import RationalMatrix
>>> A = RationalMatrix(Integer(3), Integer(4))
>>> A.base_ring()
Rational Field
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import RationalMatrix
sage: A = RationalMatrix(3, 4)
sage: A.characteristic()
0
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import RationalMatrix
>>> A = RationalMatrix(Integer(3), Integer(4))
>>> A.characteristic()
0
```

class sage.matroids.lean_matrix.**TernaryMatrix**

Bases: *LeanMatrix*

Ternary matrix class. Entries are stored bit-packed into integers.

INPUT:

- `m` – number of rows
- `n` – number of columns
- `M` – (default: `None`) `Matrix` or `TernaryMatrix` instance. Assumption: dimensions of `M` are at most `m` by `n`.
- `ring` – (default: `None`) ignored

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = TernaryMatrix(2, 2, Matrix(GF(7), [[0, 0], [0, 0]]))
sage: B = TernaryMatrix(2, 2, ring=GF(5))
sage: C = TernaryMatrix(2, 2)
sage: A == B and A == C
True
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = TernaryMatrix(Integer(2), Integer(2), Matrix(GF(Integer(7)), [[Integer(0),
↪ Integer(0)], [Integer(0), Integer(0)]]))
>>> B = TernaryMatrix(Integer(2), Integer(2), ring=GF(Integer(5)))
>>> C = TernaryMatrix(Integer(2), Integer(2))
>>> A == B and A == C
True
```

base_ring()

Return GF(3).

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = TernaryMatrix(3, 3)
sage: A.base_ring()
Finite Field of size 3
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = TernaryMatrix(Integer(3), Integer(3))
>>> A.base_ring()
Finite Field of size 3
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = TernaryMatrix(3, 4)
sage: A.characteristic()
3
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = TernaryMatrix(Integer(3), Integer(4))
>>> A.characteristic()
3
```

sage.matroids.lean_matrix.**generic_identity**(*n*, *ring*)

Return a GenericMatrix instance containing the $nimesn$ identity matrix over `ring`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = generic_identity(2, QQ)
sage: Matrix(A)
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = generic_identity(Integer(2), QQ)
>>> Matrix(A)
[1 0]
[0 1]
```

7.2 Helper functions for plotting the geometric representation of matroids

AUTHORS:

- Jayant Apte (2014-06-06): initial version

Note

This file provides functions that are called by `show()` and `plot()` methods of abstract matroids class. The basic idea is to first decide the placement of points in $mathbb{R}^2$ and then draw lines in geometric representation through these points. Point placement procedures such as `addtripts`, `addnontripts` together produce `(x,y)`

tuples corresponding to ground set of the matroid in a dictionary. These methods provide simple but rigid point placement algorithm. Alternatively, one can build the point placement dictionary manually or via an optimization that gives aesthetically pleasing point placement (in some sense. This is not yet implemented). One can then use `createline` function to produce sequence of `100` points on a smooth curve containing the points in the specified line which in turn uses `scipy.interpolate.splprep()` and `scipy.interpolate.splev()`. Then one can use sage's graphics primitives `line`, `point`, `text` and `points` to produce graphics object containing points (ground set elements) and lines (for a rank 3 matroid, these are flats of rank 2 of size greater than equal to 3) of the geometric representation of the matroid. Loops and parallel elements are added as per conventions in [Oxl2011] using function `addlp`. The priority order for point placement methods used inside plot() and show() is as follows:

1. User Specified points dictionary and lineorders
2. cached point placement dictionary and line orders (a list of ordered lists) in M._cached_info (a dictionary)
3. Internal point placement and orders deciding heuristics If a custom point placement and/or line orders is desired, then user can simply specify the custom points dictionary as:

```
M.cached_info = {'plot_positions':<dictionary_of _points>,
                 'plot_lineorders':<list of lists>}
```

7.2.1 REFERENCES

- [Oxl2011] James Oxley, "Matroid Theory, Second Edition". Oxford University Press, 2011.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: M1 = Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1,0,1],
....:                                  [0, 1, 0, 1, 0, 1, 1,0,0,1,0],
....:                                  [0, 0, 1, 1, 1, 0, 1,0,0,0,0]])
sage: pos_dict = {0: (0, 0),  1: (2, 0),  2: (1, 2),  3: (1.5, 1.0),
....:             4: (0.5, 1.0),  5: (1.0, 0.0), 6: (1.0, 0.666666666666667),
....:             7: (3,3), 8: (4,0), 9: (-1,1), 10: (-2,-2)}
sage: M1._cached_info = {'plot_positions': pos_dict, 'plot_lineorders': None}
sage: matroids_plot_helpers.geomrep(M1, sp=True)                                #␣
↪needs sage.plot sage.rings.finite_rings
Graphics object consisting of 22 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> M1 = Matroid(ring=GF(Integer(2)), matrix=[[Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(1),Integer(0),Integer(1),Integer(0),
↪Integer(1)],
...                                   [Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(1), Integer(1),Integer(0),Integer(0),Integer(1),Integer(0)],
...                                   [Integer(0), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)]])
>>> pos_dict = {Integer(0): (Integer(0), Integer(0)),  Integer(1): (Integer(2),␣
↪Integer(0)),  Integer(2): (Integer(1), Integer(2)),  Integer(3): (RealNumber('1.5'),
↪ RealNumber('1.0')),
...              Integer(4): (RealNumber('0.5'), RealNumber('1.0')),  Integer(5):␣
↪(RealNumber('1.0'), RealNumber('0.0')), Integer(6): (RealNumber('1.0'), RealNumber(
↪'0.666666666666667')),
...              Integer(7): (Integer(3),Integer(3)), Integer(8): (Integer(4),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)), Integer(9): (-Integer(1),Integer(1)), Integer(10): (-Integer(2),-
↪Integer(2))}
>>> M1._cached_info = {'plot_positions': pos_dict, 'plot_lineorders': None}
>>> matroids_plot_helpers.geomrep(M1, sp=True)                                #␣
↪needs sage.plot sage.rings.finite_rings
Graphics object consisting of 22 graphics primitives
```

sage.matroids.matroids_plot_helpers.**addlp**(*M*, *M1*, *L*, *P*, *ptsdict*, *G=None*, *limits=None*)

Return a graphics object containing loops (in inset) and parallel elements of matroid.

INPUT:

- `M` – matroid
- `M1` – a simple matroid corresponding to `M`
- `L` – list of elements in `M.groundset()` that are loops of matroid `M`
- `P` – list of elements in `M.groundset()` not in `M.simplify.groundset()` or `L`
- `ptsdict` – dictionary containing elements in `M.groundset()` not necessarily containing elements of `L`
- `G` – (optional) a sage graphics object to which loops and parallel elements of matroid M added
- `limits` – (optional) current axes limits [xmin,xmax,ymin,ymax]

OUTPUT: a 2-tuple containing:

1. A sage graphics object containing loops and parallel elements of matroid `M`
2. axes limits array

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: M = Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1],
....:                                 [0, 1, 0, 1, 0, 1, 1,0,0],
....:                                 [0, 0, 1, 1, 1, 0, 1,0,0]])
sage: [M1,L,P] = matroids_plot_helpers.slp(M)                                #␣
↪needs sage.rings.finite_rings
sage: G, lims = matroids_plot_helpers.addlp(M,M1,L,P,{0:(0,0)})              #␣
↪needs sage.plot sage.rings.finite_rings
sage: G.show(axes=False)                                                     #␣
↪needs sage.plot sage.rings.finite_rings
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> M = Matroid(ring=GF(Integer(2)), matrix=[[Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(1),Integer(0),Integer(1)],
...                                 [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1),Integer(0),Integer(0)],
...                                 [Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(1),Integer(0),Integer(0)]])
>>> [M1,L,P] = matroids_plot_helpers.slp(M)                                  #␣
↪needs sage.rings.finite_rings
>>> G, lims = matroids_plot_helpers.addlp(M,M1,L,P,{Integer(0):(Integer(0),
↪Integer(0))})                  # needs sage.plot sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
>>> G.show(axes=False)                                                    #␣
↪needs sage.plot sage.rings.finite_rings
```

Note

This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**addnontripts**(*tripts_labels*, *nontripts_labels*, *ptsdict*)

Return modified `ptsdict` with additional keys and values corresponding to `nontripts`.

INPUT:

- `tripts` – list of 3 ground set elements that are to be placed on vertices of the triangle
- `ptsdict` – dictionary (at least) containing ground set elements in `tripts` as keys and their (x,y) position as values
- `nontripts` – list of ground set elements whose corresponding points are to be placed inside the triangle

OUTPUT:

A dictionary containing ground set elements in `tripts` as keys and their (x,y) position as values along with all keys and respective values in `ptsdict`.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: from sage.matroids.advanced import setprint
sage: ptsdict={'a':(0,0),'b':(1,2),'c':(2,0)}
sage: ptsdict_1=matroids_plot_helpers.addnontripts(['a','b','c'],
....:         ['d','e','f'],ptsdict)
sage: setprint(ptsdict_1)
{'a': [0, 0], 'b': [1, 2], 'c': [0, 2], 'd': [0.6666666666666666, 1.0],
'e': [0.6666666666666666, 0.8888888888888888],
'f': [0.8888888888888888, 1.3333333333333333]}
sage: ptsdict_2=matroids_plot_helpers.addnontripts(['a','b','c'],
....:         ['d','e','f','g','h'],ptsdict)
sage: setprint(ptsdict_2)
{'a': [0, 0], 'b': [1, 2], 'c': [0, 2], 'd': [0.6666666666666666, 1.0],
'e': [0.6666666666666666, 0.8888888888888888],
'f': [0.8888888888888888, 1.3333333333333333],
'g': [0.2222222222222222, 1.0],
'h': [0.5185185185185185, 0.5555555555555555]}
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> from sage.matroids.advanced import setprint
>>> ptsdict={'a':(Integer(0),Integer(0)),'b':(Integer(1),Integer(2)),'c
↪':(Integer(2),Integer(0))}
>>> ptsdict_1=matroids_plot_helpers.addnontripts(['a','b','c'],
...         ['d','e','f'],ptsdict)
>>> setprint(ptsdict_1)
{'a': [0, 0], 'b': [1, 2], 'c': [0, 2], 'd': [0.6666666666666666, 1.0],
'e': [0.6666666666666666, 0.8888888888888888],
```

(continues on next page)

(continued from previous page)

```
'f': [0.8888888888888888, 1.3333333333333333]}
>>> ptsdict_2=matroids_plot_helpers.addnontripts(['a','b','c'],
...         ['d','e','f','g','h'],ptsdict)
>>> setprint(ptsdict_2)
{'a': [0, 0], 'b': [1, 2], 'c': [0, 2], 'd': [0.6666666666666666, 1.0],
'e': [0.6666666666666666, 0.8888888888888888],
'f': [0.8888888888888888, 1.3333333333333333],
'g': [0.2222222222222222, 1.0],
'h': [0.5185185185185185, 0.5555555555555555]}
```

Note

This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**createline**(*ptsdict*, *ll*, *lineorders2=None*)

Return ordered lists of coordinates of points to be traversed to draw a 2D line.

INPUT:

- `ptsdict` – dictionary containing keys and their (x,y) position as values
- `ll` – list of keys in `ptsdict` through which a line is to be drawn
- `lineorders2` – (optional) list of ordered lists of keys in `ptsdict` such that if ll is setwise same as any of these then points corresponding to values of the keys will be traversed in that order thus overriding internal order deciding heuristic

OUTPUT: a tuple containing 4 elements in this order:

1. Ordered list of x-coordinates of values of keys in `ll` specified in ptsdict.
2. Ordered list of y-coordinates of values of keys in `ll` specified in ptsdict.
3. Ordered list of interpolated x-coordinates of points through which a line can be drawn.
4. Ordered list of interpolated y-coordinates of points through which a line can be drawn.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: ptsdict = {'a':(1,3),'b':(2,1),'c':(4,5),'d':(5,2)}
sage: x,y,x_i,y_i = matroids_plot_helpers.createline(ptsdict,
....: ['a','b','c','d'])
sage: [len(x), len(y), len(x_i), len(y_i)]
[4, 4, 100, 100]
sage: G = line(zip(x_i, y_i), color='black', thickness=3, zorder=1)             #␣
↪needs sage.plot
sage: G += points(zip(x, y), color='black', size=300, zorder=2)                 #␣
↪needs sage.plot
sage: G.show()                                                                  #␣
↪needs sage.plot
sage: x,y,x_i,y_i = matroids_plot_helpers.createline(ptsdict,
....: ['a','b','c','d'],lineorders2=[['b','a','c','d'],
....: ['p','q','r','s']])
sage: [len(x), len(y), len(x_i), len(y_i)]
```

(continues on next page)

(continued from previous page)

```
[4, 4, 100, 100]
sage: G = line(zip(x_i, y_i), color='black', thickness=3, zorder=1)             #␣
↪needs sage.plot
sage: G += points(zip(x, y), color='black', size=300, zorder=2)                 #␣
↪needs sage.plot
sage: G.show()                                                                  #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> ptsdict = {'a':(Integer(1),Integer(3)),'b':(Integer(2),Integer(1)),'c
↪':(Integer(4),Integer(5)),'d':(Integer(5),Integer(2))}
>>> x,y,x_i,y_i = matroids_plot_helpers.createline(ptsdict,
... ['a','b','c','d'])
>>> [len(x), len(y), len(x_i), len(y_i)]
[4, 4, 100, 100]
>>> G = line(zip(x_i, y_i), color='black', thickness=Integer(3),␣
↪zorder=Integer(1))             # needs sage.plot
>>> G += points(zip(x, y), color='black', size=Integer(300), zorder=Integer(2))  ␣
↪              # needs sage.plot
>>> G.show()                                                                    #␣
↪needs sage.plot
>>> x,y,x_i,y_i = matroids_plot_helpers.createline(ptsdict,
... ['a','b','c','d'],lineorders2=[['b','a','c','d'],
... ['p','q','r','s']])
>>> [len(x), len(y), len(x_i), len(y_i)]
[4, 4, 100, 100]
>>> G = line(zip(x_i, y_i), color='black', thickness=Integer(3),␣
↪zorder=Integer(1))             # needs sage.plot
>>> G += points(zip(x, y), color='black', size=Integer(300), zorder=Integer(2))  ␣
↪              # needs sage.plot
>>> G.show()                                                                    #␣
↪needs sage.plot
```

Note

This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**geomrep**(*M1*, *B1=None*, *lineorders1=None*, *pd=None*, *sp=False*)

Return a sage graphics object containing geometric representation of matroid M1.

INPUT:

- `M1` – matroid
- `B1` – (optional) list of elements in `M1.groundset()` that correspond to a basis of `M1` and will be placed as vertices of the triangle in the geometric representation of `M1`
- `lineorders1` – (optional) list of ordered lists of elements of `M1.groundset()` such that if a line in geometric representation is setwise same as any of these then points contained will be traversed in that order thus overriding internal order deciding heuristic
- `pd` – (optional) dictionary mapping ground set elements to their (x,y) positions

- `sp` – (optional) if `True`, a positioning dictionary and line orders will be placed in `M._cached_info`

OUTPUT:

A sage graphics object of type <class 'sage.plot.graphics.Graphics'> that corresponds to the geometric representation of the matroid.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: M = matroids.catalog.P7()
sage: G = matroids_plot_helpers.geomrep(M)                                  #␣
↪needs sage.plot
sage: G.show(xmin=-2, xmax=3, ymin=-2, ymax=3)                              #␣
↪needs sage.plot
sage: M = matroids.catalog.P7()
sage: G = matroids_plot_helpers.geomrep(M, lineorders1=[['f','e','d']])     #␣
↪needs sage.plot
sage: G.show(xmin=-2, xmax=3, ymin=-2, ymax=3)                              #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> M = matroids.catalog.P7()
>>> G = matroids_plot_helpers.geomrep(M)                                    #␣
↪needs sage.plot
>>> G.show(xmin=-Integer(2), xmax=Integer(3), ymin=-Integer(2), ymax=Integer(3)) ␣
↪                                  # needs sage.plot
>>> M = matroids.catalog.P7()
>>> G = matroids_plot_helpers.geomrep(M, lineorders1=[['f','e','d']])       #␣
↪needs sage.plot
>>> G.show(xmin=-Integer(2), xmax=Integer(3), ymin=-Integer(2), ymax=Integer(3)) ␣
↪                                  # needs sage.plot
```

Note

This method does NOT do any checks.

`sage.matroids.matroids_plot_helpers.`**`it`**(*M*, *B1*, *nB1*, *lps*)

Return points on and off the triangle and lines to be drawn for a rank 3 matroid.

INPUT:

- `M` – matroid
- `B1` – list of groundset elements of `M` that corresponds to a basis of matroid `M`
- `nB1` – list of elements in the ground set of M that corresponds to `M.simplify.groundset() \ B1`
- `lps` – list of elements in the ground set of matroid M that are loops

OUTPUT: a tuple containing 4 elements in this order:

1. A dictionary containing 2-tuple (x,y) coordinates with `M.simplify.groundset()` elements that can be placed on the sides of the triangle as keys.
2. A list of 3 lists of elements of `M.simplify.groundset()` that can be placed on the 3 sides of the triangle.

3. A list of elements of `M.simplify.groundset()` that cane be placed inside the triangle in the geometric representation.
4. A list of lists of elements of `M.simplify.groundset()` that correspond to lines in the geometric representation other than the sides of the triangle.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers as mph
sage: M = Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0],
....: [0, 1, 0, 1, 0, 1, 1,0],[0, 0, 1, 1, 1, 0, 1,0]])
sage: N = M.simplify()
sage: B1 = list(N.basis())
sage: nB1 = list(set(M.simplify().groundset())-set(B1))
sage: pts,trilines,nontripts,curvedlines = mph.it(M,
....: B1,nB1,M.loops())
sage: pts
{1: (1.0, 0.0), 2: (1.5, 1.0), 3: (0.5, 1.0),
 4: (0, 0), 5: (1, 2), 6: (2, 0)}
sage: trilines
[[3, 4, 5], [2, 5, 6], [1, 4, 6]]
sage: nontripts
[0]
sage: curvedlines
[[0, 1, 5], [0, 2, 4], [0, 3, 6], [1, 2, 3], [1, 4, 6], [2, 5, 6],
 [3, 4, 5]]
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers as mph
>>> M = Matroid(ring=GF(Integer(2)), matrix=[[Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(1), Integer(1), Integer(1),Integer(0)],
... [Integer(0), Integer(1), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(1),Integer(0)],[Integer(0), Integer(0), Integer(1), Integer(1),␣
↪Integer(1), Integer(0), Integer(1),Integer(0)]])
>>> N = M.simplify()
>>> B1 = list(N.basis())
>>> nB1 = list(set(M.simplify().groundset())-set(B1))
>>> pts,trilines,nontripts,curvedlines = mph.it(M,
... B1,nB1,M.loops())
>>> pts
{1: (1.0, 0.0), 2: (1.5, 1.0), 3: (0.5, 1.0),
 4: (0, 0), 5: (1, 2), 6: (2, 0)}
>>> trilines
[[3, 4, 5], [2, 5, 6], [1, 4, 6]]
>>> nontripts
[0]
>>> curvedlines
[[0, 1, 5], [0, 2, 4], [0, 3, 6], [1, 2, 3], [1, 4, 6], [2, 5, 6],
 [3, 4, 5]]
```

Note

This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**line_hasorder**(*l*, *lodrs=None*)

Determine if an order is specified for a line.

INPUT:

- `l` – a line specified as a list of ground set elements
- `lordrs` – (optional) list of lists each specifying an order on a subset of ground set elements that may or may not correspond to a line in geometric representation

OUTPUT: a tuple containing 2 elements in this order:

1. A boolean indicating whether there is any list in `lordrs` that is setwise equal to `l`.
2. A list specifying an order on `set(l)` if 1. is True, otherwise an empty list.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: matroids_plot_helpers.line_hasorder(['a','b','c','d'],
....: [['a','c','d','b'],['p','q','r']])
(True, ['a', 'c', 'd', 'b'])
sage: matroids_plot_helpers.line_hasorder(['a','b','c','d'],
....: [['p','q','r'],['l','m','n','o']])
(False, [])
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> matroids_plot_helpers.line_hasorder(['a','b','c','d'],
... [['a','c','d','b'],['p','q','r']])
(True, ['a', 'c', 'd', 'b'])
>>> matroids_plot_helpers.line_hasorder(['a','b','c','d'],
... [['p','q','r'],['l','m','n','o']])
(False, [])
```

Note

This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**lineorders_union**(*lineorders1*, *lineorders2*)

Return a list of ordered lists of ground set elements that corresponds to union of two sets of ordered lists of ground set elements in a sense.

INPUT:

- `lineorders1` – list of ordered lists specifying orders on subsets of ground set
- `lineorders2` – list of ordered lists specifying orders subsets of ground set

OUTPUT:

A list of ordered lists of ground set elements that are (setwise) in only one of `lineorders1` or `lineorders2` along with the ones in lineorder2 that are setwise equal to any list in lineorders1.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: matroids_plot_helpers.lineorders_union([['a','b','c'],
....: ['p','q','r'],['i','j','k','l']],[['r','p','q']])
[['a', 'b', 'c'], ['p', 'q', 'r'], ['i', 'j', 'k', 'l']]
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> matroids_plot_helpers.lineorders_union([['a','b','c'],
... ['p','q','r'],['i','j','k','l']],[['r','p','q']])
[['a', 'b', 'c'], ['p', 'q', 'r'], ['i', 'j', 'k', 'l']]
```

sage.matroids.matroids_plot_helpers.**posdict_is_sane**(*M1*, *pos_dict*)

Return a boolean establishing sanity of `posdict` wrt matroid `M`.

INPUT:

- `M1` – matroid
- `posdict` – dictionary mapping ground set elements to (x,y) positions

OUTPUT:

A boolean that is `True` if posdict indeed has all the required elements to plot the geometric elements, otherwise `False`.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: M1 = Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1,0,1],
....:                                  [0, 1, 0, 1, 0, 1, 1,0,0,1,0],
....:                                  [0, 0, 1, 1, 1, 0, 1,0,0,0,0]])
sage: pos_dict = {0: (0, 0),  1: (2, 0),  2: (1, 2),  3: (1.5, 1.0),
....: 4: (0.5, 1.0),  5: (1.0, 0.0), 6: (1.0, 0.6666666666666666)}
sage: matroids_plot_helpers.posdict_is_sane(M1,pos_dict)                    #␣
↪needs sage.rings.finite_rings
True
sage: pos_dict = {1: (2, 0),  2: (1, 2),  3: (1.5, 1.0),
....:             4: (0.5, 1.0), 5: (1.0, 0.0), 6: (1.0, 0.6666666666666666)}
sage: matroids_plot_helpers.posdict_is_sane(M1,pos_dict)                    #␣
↪needs sage.rings.finite_rings
False
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> M1 = Matroid(ring=GF(Integer(2)), matrix=[[Integer(1), Integer(0), Integer(0),
↪ Integer(0), Integer(1), Integer(1), Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1)],
...                                  [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)],
...                                  [Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(1),Integer(0),Integer(0),Integer(0),
↪Integer(0)]])
>>> pos_dict = {Integer(0): (Integer(0), Integer(0)),  Integer(1): (Integer(2),␣
↪Integer(0)),  Integer(2): (Integer(1), Integer(2)),  Integer(3): (RealNumber('1.
```

(continues on next page)

(continued from previous page)

```
↪5'), RealNumber('1.0')),
... Integer(4): (RealNumber('0.5'), RealNumber('1.0')),  Integer(5): (RealNumber(
↪'1.0'), RealNumber('0.0')), Integer(6): (RealNumber('1.0'), RealNumber('0.
↪6666666666666666'))}
>>> matroids_plot_helpers.posdict_is_sane(M1,pos_dict)                          #␣
↪needs sage.rings.finite_rings
True
>>> pos_dict = {Integer(1): (Integer(2), Integer(0)),  Integer(2): (Integer(1),␣
↪Integer(2)),  Integer(3): (RealNumber('1.5'), RealNumber('1.0')),
...             Integer(4): (RealNumber('0.5'), RealNumber('1.0')), Integer(5):␣
↪(RealNumber('1.0'), RealNumber('0.0')), Integer(6): (RealNumber('1.0'),␣
↪RealNumber('0.6666666666666666'))}
>>> matroids_plot_helpers.posdict_is_sane(M1,pos_dict)                          #␣
↪needs sage.rings.finite_rings
False
```

Note

This method does NOT do any checks. `M1` is assumed to be a matroid and `posdict` is assumed to be a dictionary.

`sage.matroids.matroids_plot_helpers.`**`slp`**(*M1*, *pos_dict=None*, *B=None*)

Return simple matroid, loops and parallel elements of given matroid.

INPUT:

- `M1` – matroid
- `pos_dict` – (optional) dictionary containing non loopy elements of `M` as keys and their (x,y) positions as keys. While simplifying the matroid, all except one element in a parallel class that is also specified in `pos_dict` will be retained.
- `B` – (optional) a basis of M1 that has been chosen for placement on vertices of triangle

OUTPUT: a tuple containing 3 elements in this order:

1. Simple matroid corresponding to `M1`.
2. Loops of matroid `M1`.
3. Elements that are in $M1.groundset()$ but not in ground set of 1 or in 2

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: from sage.matroids.advanced import setprint
sage: M1 = Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1,0,1],
....:                                  [0, 1, 0, 1, 0, 1, 1,0,0,1,0],
....:                                  [0, 0, 1, 1, 1, 0, 1,0,0,0,0]])
sage: [M,L,P] = matroids_plot_helpers.slp(M1)                                  #␣
↪needs sage.rings.finite_rings
sage: M.is_simple()                                                            #␣
↪needs sage.rings.finite_rings
True
sage: setprint([L,P])                                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings
[{10, 8, 9}, {7}]
sage: M1 = Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1,0,1],
....:                                  [0, 1, 0, 1, 0, 1, 1,0,0,1,0],
....:                                  [0, 0, 1, 1, 1, 0, 1,0,0,0,0]])
sage: posdict = {8: (0, 0),  1: (2, 0),  2: (1, 2),  3: (1.5, 1.0),
....:            4: (0.5, 1.0),  5: (1.0, 0.0), 6: (1.0, 0.6666666666666666)}
sage: [M,L,P] = matroids_plot_helpers.slp(M1, pos_dict=posdict)                 #␣
↪needs sage.rings.finite_rings
sage: M.is_simple()                                                             #␣
↪needs sage.rings.finite_rings
True
sage: setprint([L,P])                                                           #␣
↪needs sage.rings.finite_rings
[{0, 10, 9}, {7}]
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> from sage.matroids.advanced import setprint
>>> M1 = Matroid(ring=GF(Integer(2)), matrix=[[Integer(1), Integer(0), Integer(0),
↪ Integer(0), Integer(1), Integer(1), Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1)],
...                                  [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)],
...                                  [Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(1),Integer(0),Integer(0),Integer(0),
↪Integer(0)]])
>>> [M,L,P] = matroids_plot_helpers.slp(M1)                                     #␣
↪needs sage.rings.finite_rings
>>> M.is_simple()                                                              #␣
↪needs sage.rings.finite_rings
True
>>> setprint([L,P])                                                            #␣
↪needs sage.rings.finite_rings
[{10, 8, 9}, {7}]
>>> M1 = Matroid(ring=GF(Integer(2)), matrix=[[Integer(1), Integer(0), Integer(0),
↪ Integer(0), Integer(1), Integer(1), Integer(1),Integer(0),Integer(1),
↪Integer(0),Integer(1)],
...                                  [Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(0), Integer(1), Integer(1),Integer(0),Integer(0),Integer(1),
↪Integer(0)],
...                                  [Integer(0), Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(0), Integer(1),Integer(0),Integer(0),Integer(0),
↪Integer(0)]])
>>> posdict = {Integer(8): (Integer(0), Integer(0)),  Integer(1): (Integer(2),␣
↪Integer(0)),  Integer(2): (Integer(1), Integer(2)),  Integer(3): (RealNumber('1.
↪5'), RealNumber('1.0')),
...             Integer(4): (RealNumber('0.5'), RealNumber('1.0')),  Integer(5):␣
↪(RealNumber('1.0'), RealNumber('0.0')), Integer(6): (RealNumber('1.0'),␣
↪RealNumber('0.6666666666666666'))}
>>> [M,L,P] = matroids_plot_helpers.slp(M1, pos_dict=posdict)                   #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings
>>> M.is_simple()                                                    #␣
↪needs sage.rings.finite_rings
True
>>> setprint([L,P])                                                  #␣
↪needs sage.rings.finite_rings
[{0, 10, 9}, {7}]
```

Note

This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**tracklims**(*lims*, *x_i=[]*, *y_i=[]*)

Return modified limits list.

INPUT:

- `lims` – list with 4 elements `[xmin,xmax,ymin,ymax]`
- `x_i` – new x values to track
- `y_i` – new y values to track

OUTPUT: list with 4 elements `[xmin,xmax,ymin,ymax]`

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: matroids_plot_helpers.tracklims([0,5,-1,7],[1,2,3,6,-1],
....: [-1,2,3,6])
[-1, 6, -1, 7]
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> matroids_plot_helpers.tracklims([Integer(0),Integer(5),-Integer(1),
↪Integer(7)],[Integer(1),Integer(2),Integer(3),Integer(6),-Integer(1)],
... [-Integer(1),Integer(2),Integer(3),Integer(6)])
[-1, 6, -1, 7]
```

Note

This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**trigrid**(*tripts*)

Return a grid of 4 points inside given 3 points as a list.

INPUT:

- `tripts` – list of 3 lists of the form [x,y] where x and y are the Cartesian coordinates of a point

OUTPUT: list of lists containing 4 points in following order:

- 1. Barycenter of 3 input points.
- 2,3,4. Barycenters of 1. with 3 different 2-subsets of input points respectively.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: points = matroids_plot_helpers.trigrid([[2,1],[4,5],[5,2]])
sage: points
[[3.666666666666665, 2.666666666666665],
 [3.222222222222222, 2.888888888888889],
 [4.222222222222222, 3.222222222222222],
 [3.5555555555555554, 1.8888888888888886]]
```

```
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> points = matroids_plot_helpers.trigrid([[Integer(2),Integer(1)],[Integer(4),
↪Integer(5)],[Integer(5),Integer(2)]])
>>> points
[[3.666666666666665, 2.666666666666665],
 [3.222222222222222, 2.888888888888889],
 [4.222222222222222, 3.222222222222222],
 [3.5555555555555554, 1.8888888888888886]]
```

Note

This method does NOT do any checks.

7.3 Set systems

Many matroid methods return a collection of subsets. In this module a class *SetSystem* is defined to do just this. The class is intended for internal use, so all you can do as a user is iterate over its members.

The class is equipped with partition refinement methods to help with matroid isomorphism testing.

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

class sage.matroids.set_system.**SetSystem**

Bases: object

A SetSystem is an enumerator of a collection of subsets of a given fixed and finite groundset. It offers the possibility to enumerate its contents. One is most likely to encounter these as output from some Matroid methods:

```
sage: M = matroids.catalog.Fano()
sage: M.circuits()
SetSystem of 14 sets over 7 elements
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.circuits()
SetSystem of 14 sets over 7 elements
```

To access the sets in this structure, simply iterate over them. The simplest way must be:

```
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
sage: T = list(S)
```

```
>>> from sage.all import *
>>> from sage.matroids.set_system import SetSystem
>>> S = SetSystem([Integer(1), Integer(2), Integer(3), Integer(4)], [[Integer(1),␣
↪Integer(2)], [Integer(3), Integer(4)], [Integer(1), Integer(2), Integer(4)]])
>>> T = list(S)
```

Or immediately use it to iterate:

```
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
sage: [min(X) for X in S]
[1, 3, 1]
```

```
>>> from sage.all import *
>>> from sage.matroids.set_system import SetSystem
>>> S = SetSystem([Integer(1), Integer(2), Integer(3), Integer(4)], [[Integer(1),␣
↪Integer(2)], [Integer(3), Integer(4)], [Integer(1), Integer(2), Integer(4)]])
>>> [min(X) for X in S]
[1, 3, 1]
```

Note that this class is intended for runtime, so no loads/dumps mechanism was implemented.

Warning

The only guaranteed behavior of this class is that it is iterable. It is expected that M.circuits(), M.bases(), and so on will in the near future return actual iterators. All other methods (which are already hidden by default) are only for internal use by the Sage matroid code.

is_connected()

Test if the *SetSystem* is connected.

A *SetSystem* is connected if there is no nonempty proper subset X of the groundset so the each subset is either contained in X or disjoint from X.

EXAMPLES:

```
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
sage: S.is_connected()
True
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4]])
sage: S.is_connected()
False
sage: S = SetSystem([1], [])
sage: S.is_connected()
True
```

```
>>> from sage.all import *
>>> from sage.matroids.set_system import SetSystem
>>> S = SetSystem([Integer(1), Integer(2), Integer(3), Integer(4)],␣
↪[[Integer(1), Integer(2)], [Integer(3), Integer(4)], [Integer(1),␣
↪Integer(2), Integer(4)]])
>>> S.is_connected()
True
>>> S = SetSystem([Integer(1), Integer(2), Integer(3), Integer(4)],␣
↪[[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> S.is_connected()
False
>>> S = SetSystem([Integer(1)], [])
>>> S.is_connected()
True
```

class sage.matroids.set_system.**SetSystemIterator**

Bases: `object`

Create an iterator for a SetSystem.

Called internally when iterating over the contents of a SetSystem.

EXAMPLES:

```
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
sage: type(S.__iter__())
<... 'sage.matroids.set_system.SetSystemIterator'>
```

```
>>> from sage.all import *
>>> from sage.matroids.set_system import SetSystem
>>> S = SetSystem([Integer(1), Integer(2), Integer(3), Integer(4)], [[Integer(1),␣
↪Integer(2)], [Integer(3), Integer(4)], [Integer(1), Integer(2), Integer(4)]])
>>> type(S.__iter__())
<... 'sage.matroids.set_system.SetSystemIterator'>
```

7.4 Unpickling methods

Python saves objects by providing a pair `(f, data)` such that `f(data)` reconstructs the object. This module collects the loading (_unpickling_ in Python terminology) functions for Sage's matroids.

Note

The reason this code was separated out from the classes was to make it play nice with lazy importing of the `Matroid()` and `matroids` keywords.

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-07-01): initial version
- Giorgos Mousa (2024-01-01): add CircuitsMatroid and FlatsMatroid

sage.matroids.unpickling.**unpickle_basis_matroid**(*version*, *data*)

Unpickle a BasisMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer; expected to be 0
- `data` – tuple `(E, R, name, BB)` in which `E` is the groundset of the matroid, `R` is the rank, `name` is a custom name, and `BB` is the bitpacked list of bases, as pickled by Sage's `bitset_pickle`.

OUTPUT: matroid

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid(matroids.catalog.Vamos())
sage: M == loads(dumps(M))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = BasisMatroid(matroids.catalog.Vamos())
>>> M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_binary_matrix**(*version*, *data*)

Reconstruct a `BinaryMatrix` object (internal Sage data structure).

Warning

Users should not call this method directly.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(2, 5)
sage: A == loads(dumps(A))  # indirect doctest
True
sage: C = BinaryMatrix(2, 2, Matrix(GF(2), [[1, 1], [0, 1]]))
sage: C == loads(dumps(C))
True
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = BinaryMatrix(Integer(2), Integer(5))
```

(continues on next page)

(continued from previous page)

```
>>> A == loads(dumps(A))  # indirect doctest
True
>>> C = BinaryMatrix(Integer(2), Integer(2), Matrix(GF(Integer(2)), [[Integer(1),␣
↪Integer(1)], [Integer(0), Integer(1)]]))
>>> C == loads(dumps(C))
True
```

sage.matroids.unpickling.**unpickle_binary_matroid**(*version*, *data*)

Unpickle a BinaryMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer (currently 0)
- `data` – tuple `(A, E, B, name)` where `A` is the representation matrix, `E` is the groundset of the matroid, `B` is the currently displayed basis, and `name` is a custom name.

  OUTPUT: `BinaryMatroid`

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(2), [[1, 0, 0, 1], [0, 1, 0, 1],
....:                            [0, 0, 1, 1]]))
sage: M == loads(dumps(M))  # indirect doctest
True
sage: M.rename('U34')
sage: loads(dumps(M))
U34
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(2)), [[Integer(1), Integer(0), Integer(0),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(0), Integer(1)],
...                            [Integer(0), Integer(0), Integer(1), Integer(1)]]))
>>> M == loads(dumps(M))  # indirect doctest
True
>>> M.rename('U34')
>>> loads(dumps(M))
U34
```

sage.matroids.unpickling.**unpickle_circuit_closures_matroid**(*version*, *data*)

Unpickle a CircuitClosuresMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer; expected to be 0
- `data` – tuple `(E, CC, name)` in which `E` is the groundset of the matroid, `CC` is the dictionary of circuit closures, and `name` is a custom name.

OUTPUT: matroid

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: M = matroids.catalog.Vamos()
sage: M == loads(dumps(M))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M == loads(dumps(M))  # indirect doctest
True
```

`sage.matroids.unpickling.`**`unpickle_circuits_matroid`**(*version*, *data*)

Unpickle a CircuitsMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer; expected to be 0
- `data` – tuple `(E, C, name)` in which `E` is the groundset of the matroid, `C` is the list of circuits , and `name` is a custom name.

OUTPUT: matroid

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: M = matroids.Theta(5)
sage: M == loads(dumps(M))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> M = matroids.Theta(Integer(5))
>>> M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_dual_matroid**(*version*, *data*)

Unpickle a DualMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer; expected to be 0
- `data` – tuple `(M, name)` in which `M` is the internal matroid, and `name` is a custom name

OUTPUT: matroid

Warning

Users should not call this function directly. Instead, use load/save.

EXAMPLES:

```
sage: M = matroids.catalog.Vamos().dual()
sage: M == loads(dumps(M))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos().dual()
>>> M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_flats_matroid**(*version*, *data*)

Unpickle a `FlatsMatroid`.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer; expected to be 0
- `data` – tuple `(E, F, name)` in which `E` is the groundset of the matroid, `F` is the dictionary of flats, and `name` is a custom name.

OUTPUT: matroid

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.catalog.Vamos())
sage: M == loads(dumps(M))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.matroids.flats_matroid import FlatsMatroid
>>> M = FlatsMatroid(matroids.catalog.Vamos())
>>> M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_gammoid**(*version*, *data*)

Unpickle a *Gammoid*.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer; expected to be 0
- `data` – tuple `(D, roots, E, name)` in which `D` is a loopless DiGraph representing the gammoid, `roots` is a subset of the vertices, `E` is the groundset of the matroid, and `name` is a custom name.

OUTPUT: matroid

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: from sage.matroids.gammoid import Gammoid
sage: M = Gammoid(digraphs.TransitiveTournament(5), roots=[3, 4])
sage: M == loads(dumps(M))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.matroids.gammoid import Gammoid
>>> M = Gammoid(digraphs.TransitiveTournament(Integer(5)), roots=[Integer(3),␣
↪Integer(4)])
>>> M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_generic_matrix**(*version*, *data*)

Reconstruct a `GenericMatrix` object (internal Sage data structure).

Warning

Users should not call this method directly.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = GenericMatrix(2, 5, ring=QQ)
sage: A == loads(dumps(A))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = GenericMatrix(Integer(2), Integer(5), ring=QQ)
>>> A == loads(dumps(A))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_graphic_matroid**(*version*, *data*)

Unpickle a GraphicMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer (currently 0)
- `data` – tuple consisting of a SageMath graph and a name

OUTPUT: `GraphicMatroid`

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: M = Matroid(graphs.DiamondGraph())                                    #␣
↪needs sage.graphs
sage: M == loads(dumps(M))                                                  #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> M = Matroid(graphs.DiamondGraph())                                      #␣
↪needs sage.graphs
>>> M == loads(dumps(M))                                                    #␣
↪needs sage.graphs
True
```

sage.matroids.unpickling.**unpickle_linear_matroid**(*version*, *data*)

Unpickle a LinearMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer (currently 0)
- `data` – tuple `(A, E, reduced, name)` where `A` is the representation matrix, `E` is the groundset of the matroid, `reduced` is a boolean indicating whether `A` is a reduced matrix, and `name` is a custom name.

OUTPUT: `LinearMatroid`

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 0, 1, 1], [0, 1, 0, 1, 2],
....:                            [0, 1, 1, 1, 3]]))
sage: M == loads(dumps(M))  # indirect doctest
True
sage: M.rename('U35')
sage: loads(dumps(M))
U35
```

```
>>> from sage.all import *
>>> M = Matroid(Matrix(GF(Integer(7)), [[Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(1)], [Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(2)],
...                                  [Integer(0), Integer(1),␣
↪Integer(1), Integer(1), Integer(3)]]))
>>> M == loads(dumps(M))  # indirect doctest
True
>>> M.rename('U35')
>>> loads(dumps(M))
U35
```

sage.matroids.unpickling.**unpickle_minor_matroid**(*version*, *data*)

Unpickle a MinorMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer; currently 0
- `data` – tuple `(M, C, D, name)`, where `M` is the original matroid of which the output is a minor, `C` is the set of contractions, `D` is the set of deletions, and `name` is a custom name.

OUTPUT: *MinorMatroid*

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: M = matroids.catalog.Vamos().minor('abc', 'g')
sage: M == loads(dumps(M))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Vamos().minor('abc', 'g')
>>> M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_plus_minus_one_matrix**(*version*, *data*)

Reconstruct an `PlusMinusOneMatrix` object (internal Sage data structure).

Warning

Users should not call this method directly.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = PlusMinusOneMatrix(2, 5)
sage: A == loads(dumps(A))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = PlusMinusOneMatrix(Integer(2), Integer(5))
>>> A == loads(dumps(A))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_quaternary_matrix**(*version*, *data*)

Reconstruct a `QuaternaryMatrix` object (internal Sage data structure).

Warning

Users should not call this method directly.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.matroids.lean_matrix import *
sage: A = QuaternaryMatrix(2, 5, ring=GF(4, 'x'))
sage: A == loads(dumps(A))  # indirect doctest
True
sage: C = QuaternaryMatrix(2, 2, Matrix(GF(4, 'x'), [[1, 1], [0, 1]]))
sage: C == loads(dumps(C))
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> from sage.matroids.lean_matrix import *
>>> A = QuaternaryMatrix(Integer(2), Integer(5), ring=GF(Integer(4), 'x'))
>>> A == loads(dumps(A))  # indirect doctest
True
>>> C = QuaternaryMatrix(Integer(2), Integer(2), Matrix(GF(Integer(4), 'x'),␣
```

(continues on next page)

(continued from previous page)

```
↪[[Integer(1), Integer(1)], [Integer(0), Integer(1)]]))
>>> C == loads(dumps(C))
True
```

sage.matroids.unpickling.**unpickle_quaternary_matroid**(*version*, *data*)

Unpickle a QuaternaryMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer (currently 0)
- `data` – tuple `(A, E, B, name)` where `A` is the representation matrix, `E` is the groundset of the matroid, `B` is the currently displayed basis, and `name` is a custom name.

OUTPUT: `TernaryMatroid`

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = QuaternaryMatroid(Matrix(GF(3), [[1, 0, 0, 1], [0, 1, 0, 1],
....:           [0, 0, 1, 1]]))
sage: M == loads(dumps(M))  # indirect doctest
True
sage: M.rename('U34')
sage: loads(dumps(M))
U34
sage: M = QuaternaryMatroid(Matrix(GF(4, 'x'), [[1, 0, 1],                    #␣
↪needs sage.rings.finite_rings
....:                                           [1, 0, 1]]))
sage: loads(dumps(M)).representation()                                        #␣
↪needs sage.rings.finite_rings
[1 0 1]
[1 0 1]
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = QuaternaryMatroid(Matrix(GF(Integer(3)), [[Integer(1), Integer(0),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(0), Integer(1)],
...           [Integer(0), Integer(0), Integer(1), Integer(1)]]))
>>> M == loads(dumps(M))  # indirect doctest
True
>>> M.rename('U34')
>>> loads(dumps(M))
U34
>>> M = QuaternaryMatroid(Matrix(GF(Integer(4), 'x'), [[Integer(1), Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)],                          # needs sage.rings.finite_rings
...                                                [Integer(1), Integer(0),␣
↪Integer(1)]]))
>>> loads(dumps(M)).representation()                                        #␣
↪needs sage.rings.finite_rings
[1 0 1]
[1 0 1]
```

sage.matroids.unpickling.**unpickle_rational_matrix**(*version*, *data*)

Reconstruct a *sage.matroids.lean_matrix.RationalMatrix* object (internal Sage data structure).

Warning

Users should not call this method directly.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import RationalMatrix
sage: A = RationalMatrix(2, 5)
sage: A == loads(dumps(A))  # indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import RationalMatrix
>>> A = RationalMatrix(Integer(2), Integer(5))
>>> A == loads(dumps(A))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_regular_matroid**(*version*, *data*)

Unpickle a RegularMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer (currently 0)
- `data` – tuple `(A, E, reduced, name)` where `A` is the representation matrix, `E` is the groundset of the matroid, `reduced` is a boolean indicating whether `A` is a reduced matrix, and `name` is a custom name.

OUTPUT: `RegularMatroid`

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: M = matroids.catalog.R10()
sage: M == loads(dumps(M))  # indirect doctest
```

(continues on next page)

(continued from previous page)

```
True
sage: M.rename('R_{10}')
sage: loads(dumps(M))
R_{10}
```

```
>>> from sage.all import *
>>> M = matroids.catalog.R10()
>>> M == loads(dumps(M))  # indirect doctest
True
>>> M.rename('R_{10}')
>>> loads(dumps(M))
R_{10}
```

sage.matroids.unpickling.**unpickle_ternary_matrix**(*version*, *data*)

Reconstruct a TernaryMatrix object (internal Sage data structure).

Warning

Users should not call this method directly.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = TernaryMatrix(2, 5)
sage: A == loads(dumps(A))  # indirect doctest
True
sage: C = TernaryMatrix(2, 2, Matrix(GF(3), [[1, 1], [0, 1]]))
sage: C == loads(dumps(C))
True
```

```
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = TernaryMatrix(Integer(2), Integer(5))
>>> A == loads(dumps(A))  # indirect doctest
True
>>> C = TernaryMatrix(Integer(2), Integer(2), Matrix(GF(Integer(3)), [[Integer(1),
↪ Integer(1)], [Integer(0), Integer(1)]]))
>>> C == loads(dumps(C))
True
```

sage.matroids.unpickling.**unpickle_ternary_matroid**(*version*, *data*)

Unpickle a TernaryMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the load and save commands, and you should never have to call this function directly.

INPUT:

- version – integer (currently 0)
- data – tuple (A, E, B, name) where A is the representation matrix, E is the groundset of the matroid, B is the currently displayed basis, and name is a custom name.

OUTPUT: `TernaryMatroid`

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = TernaryMatroid(Matrix(GF(3), [[1, 0, 0, 1], [0, 1, 0, 1],
....:            [0, 0, 1, 1]]))
sage: M == loads(dumps(M))  # indirect doctest
True
sage: M.rename('U34')
sage: loads(dumps(M))
U34
```

```
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = TernaryMatroid(Matrix(GF(Integer(3)), [[Integer(1), Integer(0),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(1), Integer(0), Integer(1)],
...            [Integer(0), Integer(0), Integer(1), Integer(1)]]))
>>> M == loads(dumps(M))  # indirect doctest
True
>>> M.rename('U34')
>>> loads(dumps(M))
U34
```

`sage.matroids.unpickling.`**`unpickle_transversal_matroid`**(*version*, *data*)

Unpickle a TransversalMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – integer (currently 0)
- `data` – tuple `(sets, groundset, name)`, where `groundset` is a `frozenset` of elements, and `sets` is a `frozenset` of tuples consisting of a name for the set, and a `frozenset` of groundset elements it contains.

OUTPUT: `TransversalMatroid`

Warning

Users should never call this function directly.

EXAMPLES:

```
sage: from sage.matroids.transversal_matroid import *
sage: sets = [range(6)] * 3
sage: M = TransversalMatroid(sets)
```

(continues on next page)

(continued from previous page)

```
sage: M == loads(dumps(M))
True
sage: M.rename('U36')
sage: loads(dumps(M))
U36
```

```
>>> from sage.all import *
>>> from sage.matroids.transversal_matroid import *
>>> sets = [range(Integer(6))] * Integer(3)
>>> M = TransversalMatroid(sets)
>>> M == loads(dumps(M))
True
>>> M.rename('U36')
>>> loads(dumps(M))
U36
```

CHAPTER

EIGHT

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

m

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z