<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Basics</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Database of matroids</td>
<td>125</td>
</tr>
<tr>
<td>3</td>
<td>Concrete implementations</td>
<td>211</td>
</tr>
<tr>
<td>4</td>
<td>Abstract matroid classes</td>
<td>305</td>
</tr>
<tr>
<td>5</td>
<td>Advanced functionality</td>
<td>321</td>
</tr>
<tr>
<td>6</td>
<td>Internals</td>
<td>337</td>
</tr>
<tr>
<td>7</td>
<td>Indices and Tables</td>
<td>373</td>
</tr>
<tr>
<td></td>
<td>Python Module Index</td>
<td>375</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>377</td>
</tr>
</tbody>
</table>
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
```

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

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

```python
sage: F7 = matroids.catalog.Fano()
sage: 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:**

```python
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
```

```python
>>> 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(0), Integer(1)])

>>> M = Matroid(A)

>>> M.is_isomorphic(matroids.catalog.Fano())
True

>>> M = Matroid(graphs.PetersenGraph())
# needs sage.graphs

>>> M.rank()
# needs sage.graphs
```

(continues on next page)
AUTHORS:

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

### 1.1.4 Functions

`sage.matroids.constructor.Matrix(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:

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

or:

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

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) If provided, the groundset of the matroid. Otherwise, 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 – (default: False) boolean. 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 – (default: True) boolean. 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:
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:

   ```python
   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', bases=[['a', 'b'], ['a', 'c'], ...
                   ['a', 'd'], ['b', 'c'], ...
                   ['b', 'd'], ['c', 'd']])
   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
   ```

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

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

(continues on next page)
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(independent_sets=[a, ab, b, ab, a, b, ac])
>>> 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'])
A matroid specified by a list of circuits gets converted to a `CircuitsMatroid` internally:

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

```python
M1 == M2
```

```python
True
```

```python
M = Matroid(groundset='abcd', circuits=['abc', 'abd', 'acd', 'bcd'])
```

```python
type(M)
```

```python
<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!

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

```python
M.is_valid()
```

```python
False
```

4. Flats:

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

```python
sage: M = Matroid(flats={0: ['], 1: ['a', 'b'], 2: ['ab']})
```

```python
M.is_isomorphic(matroids.Uniform(2, 2)) and M.is_valid()
```

```python
True
```

```python
type(M)
```

```python
<class 'sage.matroids.flats_matroid.FlatsMatroid'>
```

```python
from sage.all import *
>>> M = Matroid(flats=(Integer(0): [''], Integer(1): ['a', 'b'], Integer(2): ['ab']))
```

```python
M.is_isomorphic(matroids.Uniform(Integer(2), Integer(2))) and M.is_valid()
```

```python
True
```

```python
type(M)
```

```python
<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., \texttt{is\_valid()}):

```python
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'>
```

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

```python
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)
>>> M.is_isomorphic(matroids.Uniform(2, 2)) and M.is_valid()
True
>>> type(M)
<class 'sage.matroids.flats_matroid.FlatsMatroid'>
```

5. Graph:

Sage has great support for graphs, see \texttt{sage.graphs.graph}.

```python
sage: G = graphs.PetersenGraph()  # needs sage.graphs
sage: 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:

```python
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())
['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:

```python
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)]
```

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

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

1.1. Matroid construction
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 *

sage: G1 = graphs.CycleGraph(Integer(3)); G2 = graphs.DiamondGraph()
>>> G = G1.disjoint_union(G2)
>>> M = Matroid(G, regular=True); M
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 \texttt{G.edge_iterator()} provides:

sage: G = Graph([(0, 1), (0, 2), (0, 2), (1, 2)], multiedges=True)

sage: M = Matroid('abcd', G)
sage: M.rank(['b', 'c'])
1

(continues on next page)
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 \([0..m-1]\):

```python
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)]
```

```python
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]
```

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):

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

```python
>>> from sage.all import *
>>> G = Graph(['(Integer(0), Integer(1))', '(Integer(0), Integer(2))',  # needs sage.graphs
             '(Integer(1), Integer(2))'])
>>> M = Matroid(G, regular=True)  # needs sage.graphs
>>> sorted(M.groundset())  # needs sage.graphs
[0, 1, 2, 3]
```

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

```python
sage: 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()`:
Matroid Theory, Release 10.4

6. Matrix:

The basic input is a Sage matrix:

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

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

Various shortcuts are possible:

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

>>> M1 = Matroid(matrix=[[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(1), Integer(0),
                        Integer(1), Integer(1)],
                        [Integer(0), Integer(0), Integer(1), Integer(0),
                        Integer(1), Integer(1), Integer(1), Integer(1),
                        Integer(0), Integer(0)],
                        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)],
                        [Integer(0), Integer(1), Integer(1), Integer(1)],
                        ring=GF(Integer(2)))

>>> M4 = Matroid([Integer(0), Integer(1), Integer(2), Integer(3), Integer(4),
                Integer(5)], matrix)  

>>> 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:

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

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

>>> Matroid([Integer(0), Integer(1), Integer(2)],
           [[Integer(1), Integer(0), Integer(0),
             Integer(1), Integer(1)],
             [Integer(0), Integer(1), 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:

```python
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
```

```python
>>> 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)],
                                ...]
```

(continues on next page)
We automatically create an optimized subclass, if available:

```python
sage: M = 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: N = 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: P = Matroid([0, 1, 2, 3, 4, 5],
                matrix=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
                field=GF(4, 'x'))
Quaternary matroid of rank 3 on 6 elements
```

Otherwise the generic LinearMatroid class is used:
An integer matrix is automatically converted to a matrix over \( \mathbb{Q} \). If you really want integers, you can specify the ring explicitly:

```python
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=IntegerRing())
sage: M.base_ring()
Integer Ring
```

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

```python
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()
```

(continues on next page)
The keywords `morphism` and `reduced_morphism` are also available:

```python
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
```
### 7. Rank function:

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

```python
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
```

### 8. Circuit closures:

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

```python
sage: M = Matroid(circuit_closures={3: ["edfg", "acdg", "bcfg",
.....:     "cefh", "afgh", "abce", "abdf", "bgh", "bcdf", "adeh"],
.....:     4: ["abcdefg"]})
sage: 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 \(X\):

```python
sage: M = Matroid(circuit_closures=[(2, 'abd'), (3, 'abcde'),
.....:     2, 'bce')])
sage: M.equals(matroids.catalog.Q6())
True
```

---

**1.1. Matroid construction**
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:

```python
sage: M = Matroid("abcdef", "00000000000000", 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"})]
```

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

```python
sage: Matroid("abcdefg", revlex="0++++++++0++++0++++0+++aaa++", 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):

```python
sage: Matroid("abcdef", "00000000000000*")
Traceback (most recent call last):
...  
TypeError: for RevLex-Index, the rank needs to be specified
```

```python
sage: Matroid("abcdef", "00000000000000", rank=3)
Traceback (most recent call last):
...  
ValueError: expected string of length 20 (6 choose 3), got 15
```

```python
sage: M = Matroid("abcdef", "*000000000000000000", rank=4); M
Matroid of rank 4 on 6 elements with 2 bases
sage: M.is_valid()
False
```
>>> from sage.all import *
>>> Matroid("abcdef", "0000000******0**")
Traceback (most recent call last):
...  
TypeError: for RevLex-Index, the rank needs to be specified
>>> Matroid("abcdef", "0000000******0**", rank=Integer(3))
Traceback (most recent call last):
...  
ValueError: expected string of length 20 (6 choose 3), got 15

>>> M = Matroid("abcdef", "*00000000000000*", 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
```
```
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
```
```
Regular matroid of rank 3 on 6 elements with 16 bases
```
```
M == N
```
```
False
```
```
M.is_isomorphic(N)
```
```
True
```
```
Matrix(N)  # random
```
```
[1 0 0 1 1 0]
[0 1 0 1 1 1]
[0 0 1 0 1 1]
```

(continues on next page)
The regular option:

```python
sage: M = Matroid(reduced_matrix=[[1, 1, 0],
                                 [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))
True
```

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

```python
sage: M = matroids.catalog.Fano()
sage: N = Matroid(M, regular=True)
Traceback (most recent call last):
  ... ValueError: input is not a valid regular matroid
sage: N = Matroid(M, regular=True, check=False); N
Regular matroid of rank 3 on 7 elements with 32 bases
sage: N.is_valid()
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]]),
    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(1), Integer(2)]]))
```

```
>>> N = Matroid(Matrix(GF(Integer(3)), [[Integer(1), Integer(0), Integer(1),
    Integer(1), Integer(0), Integer(1), Integer(1), Integer(1), Integer(2)]]),
    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()`
- \texttt{basis()}
- \texttt{max-independent()}
- \texttt{circuit()}
- \texttt{fundamental_circuit()}
- \texttt{closure()}
- \texttt{augment()}
- \texttt{corank()}
- \texttt{full_corank()}
- \texttt{cobasis()}
- \texttt{max_coiindependent()}
- \texttt{cocircuit()}
- \texttt{fundamental_cocircuit()}
- \texttt{coclosure()}
- \texttt{is-independent()}
- \texttt{is-dependent()}
- \texttt{is_basis()}
- \texttt{is_circuit()}
- \texttt{is_closed()}
- \texttt{is_coindependent()}
- \texttt{is_codependent()}
- \texttt{is_cobasis()}
- \texttt{is_cocircuit()}
- \texttt{is_coclosed()}

\item \textbf{Verification}
- \texttt{is_valid()}

\item \textbf{Enumeration}
- \texttt{circuits()}
- \texttt{nonspanning_circuits()}
- \texttt{cocircuits()}
- \texttt{noncospanning_cocircuits()}
- \texttt{circuit_closures()}
- \texttt{nonspanning_circuit_closures()}
- \texttt{bases()}
- \texttt{independent_sets()}
- \texttt{nonbases()}
- \texttt{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()
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._? 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:

```python
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()
```

(continues on next page)
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(0), Integer(1), 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:
Testing. Note that the abstract base class does not support pickling:

```python
>>> 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(1), Integer(1), Integer(1), Integer(1)],
  ...
  [Integer(0), Integer(0), Integer(1), Integer(1),
   Integer(0), Integer(1), Integer(1)])))
>>> N = (M / [Integer(2)]).delete([Integer(3), Integer(4)])
>>> sorted(N.groundset())
[0, 1, 5, 6]
>>> N.full_rank()
2
```

1.2.4 REFERENCES

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

```python
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
```

```python
>>> 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  
...   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:**

```python
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
```

```python
Traceback (most recent call last):
  ... ValueError: ['x'] is not a subset of the groundset
```

```python
sage: sorted(M.augment(['x']))
Traceback (most recent call last):
  ... ValueError: ['x'] is not a subset of the groundset
```

```python
>>> 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
```

(continues on next page)
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

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

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 $I \subseteq F_1 \subseteq F_2 \subseteq \cdots \subseteq F_\ell$.

OUTPUT: a simplicial complex

EXAMPLES:

```sage
sage: M = matroids.catalog.Fano()
```

```sage
sage: A = M.augmented_bergman_complex(); A
```

```
Simplicial complex with 22 vertices and 91 facets
```

```sage
sage: M = matroids.Uniform(2,3)
```

```sage
sage: A = M.augmented_bergman_complex(); A
```

```
Simplicial complex with 7 vertices and 9 facets
```

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

>>> M = matroids.catalog.Fano()

>>> A = M.augmented_bergman_complex(); A

```
Simplicial complex with 22 vertices and 91 facets
```

```sage
>>> M = matroids.Uniform(Integer(2),Integer(3))
```

```sage
>>> A = M.augmented_bergman_complex(); A
```

```
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
sage: sorted(A.faces()[0])
```

```
needs sage.graphs
```

(continues on next page)
[(L0,), (L1,), (L2,), (R[0],), (R[1],), (R[2],), (R[3],)]
sage: sorted(map(sorted, A.faces()[1]))

# needs sage.graphs
[['L0', 'L1'], ['L0', 'L2'], ['L0', 'R[0]'], ['L1', 'L2'], ['L1', 'R[1]'], ['L2', 'R[2]'], ['R[0]', 'R[3]'], ['R[1]', 'R[3]'], ['R[2]', 'R[3]']]

>>> from sage.all import *

>>> sorted(A.faces()[Integer(0)])

# needs sage.graphs
[['L0', 'L1'], ['L0', 'L2'], ['L0', 'R[0]'], ['L1', 'L2'], ['L1', 'R[1]'], ['L2', 'R[2]'], ['R[0]', 'R[3]'], ['R[1]', 'R[3]'], ['R[2]', 'R[3]']]

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 $\sigma$ 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()

(continues on next page)
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]]]
```

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]]
```

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 *

```python
>> 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:**

```python
sage: M = matroids.catalog.Fano()
sage: 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:**

```python
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
```

```python
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.binary_matroid()
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
```

(continues on next page)
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:

```python
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)}
```

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

```python
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
```

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 \textit{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:**
- \textit{ordering} – list (optional); a total ordering of the groundset

**EXAMPLES:**

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

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

**characteristic_polynomial** ($la=None$)

Return the characteristic polynomial of the matroid.

The \textit{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

$$
\sum_{i=0}^n w_i \lambda^{r-i},
$$

where $\{w_i\}_{i=0}^n$ are the Whitney numbers of the first kind.

**INPUT:**
- \textit{la} – a variable or numerical argument (optional)

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

**EXAMPLES:**

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

```python
sage: M.characteristic_polynomial().factor()
(l - 4) * (l - 3) * (l - 2) * (l - 1)
```

(continues on next page)
sage: M.characteristic_polynomial(5)
24

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

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

chow_ring(R=None)

Return the Chow ring of self over $R$.

Let $M$ be a matroid and $R$ be a commutative ring. The **Chow ring of $M$** is the quotient ring

$$A^*(M)_R := R[x_{F_1}, \ldots, x_{F_k}] / (Q_M + L_M),$$

where

- $F_1, \ldots, F_k$ are the non-empty proper flats of $M$,
- $Q_M$ is the ideal generated by all $x_{F_i}x_{F_j}$ where $F_i$ and $F_j$ are incomparable elements in the lattice of flats, and
• $L_M$ is the ideal generated by all linear forms

$$\sum_{i_1 \in F} x_{i_1} - \sum_{i_2 \in F} x_{i_2}$$

for all $i_1 \neq i_2 \in E$.

INPUT:

• $R$ — (default: $\mathbb{Z}$) the base ring

EXAMPLES:

```python
sage: M = matroids.Wheel(2)
sage: A = M.chow_ring(); A
Chow ring of Wheel(2): Regular matroid of rank 2 on 4 elements
with 5 bases over Integer Ring
sage: A.gens()
(A23, A23, A23)
sage: A23 = A.gen(0)
# needs sage.libs.singular
sage: A23*A23
0
```

We construct a more interesting example using the Fano matroid:

```python
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
```

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

```python
sage: G = A.gens()[6:]
sage: Ag, Aabf, Aace, Aadg, Abcd, Abeg, Acfg, Adef = G
(continues on next page)```
REFERENCES:

• [FY2004]

• [AHK2015]

circuit \(X = \text{None}\)

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:

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

```python  
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']]]```
```python
>>> 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']]
```

\textbf{circuits} \((k=None)\)

Return the circuits of the matroid.

**INPUT:**

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

**OUTPUT:** \texttt{SetSystem}

See also:

\texttt{M.circuit()}

**EXAMPLES:**

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

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

\textbf{circuits_iterator} \((k=None)\)

Return an iterator over the circuits of the matroid.

See also:

\texttt{circuit()}

**EXAMPLES:**

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

(continues on next page)
closures (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:

```python
sage: M = matroids.catalog.Vamos()
```

```python
sage: sorted(M.closure(set(['a', 'b', 'c'])))
['a', 'b', 'c', 'd']
```

```python
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:
**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
```
c cocircuits()

Return the cocircuits of the matroid.

OUTPUT: SetSystem

See also:
M.cocircuit()

EXAMPLES:

```python
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 *
```  

```python
>>> 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:

```python
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 *
```  

```python
>>> 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]]
```
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:**

```python
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
```

The 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
```

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]]
```

Put an element in series with another:

```
sage: M = matroids.catalog.Fano()
sage: N = M.coextension(z, [c])
sage: 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 \( M.\text{extensions()} \).

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:

\( M.\text{coextension()} \), \( M.\text{modular_cut()} \), \( M.\text{linear_subclasses()} \), \( sage.\text{matroids.} extension, M.\text{extensions()} \), \( M.\text{dual()} \)

EXAMPLES:

```python
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
```

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

>>> M = matroids.catalog.P8()

>>> len(list(M.coextensions()))
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:

```python
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']]
```

\( \text{coloops}() \)

Return the set of coloops of the matroid.

A \emph{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:

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

\( \text{components}() \)

Return a list of the components of the matroid.

A \emph{component} is an inclusionwise maximal connected subset of the matroid. A subset is \emph{connected} if the matroid resulting from deleting the complement of that subset is \emph{connected}.

OUTPUT: a list of subsets
See also:

\texttt{M.is_connected()}, \texttt{M.delete()}

**EXAMPLES:**

```python
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]])
```

```python
c connectivity (S, T=None)
Evaluatethe connectivity function of the matroid.
If the input is a single subset \( S \) of the groundset \( E \), then the output is \( r(S) + r(E \setminus S) - r(E) \).
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:**

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

```python
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 \( M \cap X \) 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:**

```python
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
```

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

```python
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
```

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

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

```python
sage: 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:**

```python
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
```

```python
>>> 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):
...
(continues on next page)```
... 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')
M.cosimplify().size()
```

```
3
```

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

```sage
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 \( M \setminus X \) 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: sorted(M.delete(['a', 'c']))
['a', 'b', 'd', 'e', 'f', 'g']
sage: M.delete(['a']) == M.delete(['a'])
True
```
One can use a single element, rather than a set:

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

Note that one can iterate over strings:

```python
sage: M = matroids.catalog.Fano()
sage: 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:

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

```python
>>> 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:

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

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

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

Return the dependent sets of fixed size.

**INPUT:**

- \( k \) – integer

**EXAMPLES:**

```python
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']]
```

**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:**

```python
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']]
```

```python
>> 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 \(E = \bigcup_{i=1}^{n} E_i\) and \(I = \bigcup_{i=1}^{n} I_i\).

**INPUT:**

- *matroids* — a matroid or list of matroids

**OUTPUT:** an instance of `MatroidSum`

**EXAMPLES:**

```python
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
```

**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:**

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

(continues on next page)
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)
False
sage: M == M1
False
sage: M == M2
True
```

```bash
>>> 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)],
    ...: [Integer(1), Integer(1), Integer(1), Integer(1)],
    ...: [Integer(1), Integer(1), Integer(1), Integer(1)],
], field=GF(Integer(2)))
>>> M.equals(M1)
True
>>> M.equals(M2)
True
```
\[
\text{LinearMatroid} \text{ instances } M \text{ and } N \text{ satisfy } M = N \text{ if the representations are equivalent up to row operations and column scaling:}
\]

```
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
```

\section*{extension (element=None, subsets=None)}

Return an extension of the matroid.

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

\textbf{INPUT:}

- \texttt{element} – (default: None) the label of the new element. If not specified, a new label will be generated automatically.
- \texttt{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)
N = M.extension(6)
N.is_isomorphic(matroids.Uniform(3, 7))
True
```

```python
>>> 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)
H = [frozenset([0, 1])]
N = M.extension(6, H)
N
Matroid of rank 3 on 7 elements with 34 bases
```

```python
>>> 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
```

Putting an element in parallel with another:

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

```python
>>> 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]

```python
>>> 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:**

```python
sage: from sage.matroids.advanced import setprint
sage: M = matroids.catalog.Fano()
sage: setprint(M.flat_cover())
# needs sage.rings.finite_rings

\{\{a\}, \{b\}, \{c\}, \{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:**

```python
>>> 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\}, \{c\}, \{d\}, \{e\}, \{f\}\}
```
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 \( M.\text{size()} - M.\text{full_rank()} \).

OUTPUT: integer

See also:

\( M.\text{dual()}\), \( M.\text{full_rank()} \)

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:**

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

```python
>>> 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:**

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

```python
>>> 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 $\infty$.

EXAMPLES:

\begin{verbatim}
sage: matroids.Uniform(5, 5).girth()
+Infinity
sage: matroids.catalog.K4().girth()
3
sage: matroids.catalog.Vamos().girth()
4
\end{verbatim}

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:

\begin{verbatim}
sage: M = sage.matroids.matroid.Matroid()
sage: M.groundset()
Traceback (most recent call last):
  ... NotImplementedError: subclasses need to implement this
\end{verbatim}

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 \( s_i(M/F) \) 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:**

```python
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)
```

```python
>>> 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)
```

**has_minor** \((N, \text{certificate}=False)\)

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)`. 

66 Chapter 1. Basics
INPUT:
- \( N \) – an instance of a \texttt{Matroid} object
- \texttt{certificate} – boolean (default: \texttt{False}); if True, returns \texttt{True, (X, Y, dic)} where \( N \) is isomorphic to \texttt{self.minor(X, Y)}, and \texttt{dic} is an isomorphism between \( N \) and \texttt{self.minor(X, Y)}

OUTPUT: boolean or tuple

See also:
- \texttt{M.minor()}, \texttt{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'}))
```

\texttt{hyperplanes()}

Return the hyperplanes of the matroid.

A \textit{hyperplane} is a flat of rank \texttt{self.full_rank()} – 1. A \textit{flat} is a closed set.

OUTPUT: SetSystem

See also:
- \texttt{M.flats()}

EXAMPLES:
Matroid Theory, Release 10.4

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

```python
>>> 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 $\mathbb{R}^n$. 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
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
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:**

```python
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
```

```bash
>>> 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:**

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

1.2. The abstract Matroid class
```python
>>> 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 \((other, weights=\text{None})\)

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 \(\sum \text{weights}(e) \text{ for } e \text{ in } S \).

INPUT:

- \(other \) – a second matroid with the same groundset as this matroid
- \(weights\) – (default: \text{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:

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

```python
>>> 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)
```

(continues on next page)
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.

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 – (default: False) boolean; 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:

```python
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
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()
```

```python
>>> 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(0), Integer(0)],
...: [Integer(0), Integer(1), Integer(0),
...: Integer(1), Integer(2), Integer(0)]]),
```

(continues on next page)
is_3connected

Return True if the matroid is 3-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 3-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
sage: matroids.Uniform(2, 6).is_connected()
sage: matroids.Uniform(2, 6).is_connected()
sage: matroids.Uniform(4, 8).is_connected()
```
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
Matroid Theory, Release 10.4

>>> 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 $\ell$ 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 – (default: False) boolean; if True return True, C, where C is a non $k1 k2$ circuit
OUTPUT: boolean or tuple

See also:

M.chordality()

EXAMPLES:

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

```python
>>> 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(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:

```python
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
```
```python
>>> 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 \( C = A \cup B \) 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:**

```python
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
```

```python
>>> 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)])
(continues on next page)
```
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:

```python
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()
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:**

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

(continues on next page)
is_coindependent \((X)\)

Check if a subset is co-independent in the matroid.

A set is co-independent if it is independent in the dual matroid.

INPUT:

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

OUTPUT: boolean

See also:

\(M\text{.dual()}\), \(M\text{.is_independent()}\)

EXAMPLES:

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

is_connected \((\text{certificate}=False)\)

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:

```python
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
```

```python
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
```

```python
sage: matroids.catalog.Fano().dual()  # Test if the matroid is cosimple.
sage: M.is_cosimple()
True
sage: N = M.delete('a')
sage: N.is_cosimple()
False
```

```python
from sage.all import *

M = matroids.catalog.Fano().dual()
M.is_cosimple()
True
```

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

>>> M = matroids.catalog.Fano().dual()

>>> M.is_cosimple()
True

>>> N = M.delete('a')

>>> N.is_cosimple()
False
```

**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:

```python
sage: M = matroids.catalog.Fano().dual()
sage: M.is_cosimple()
True
```

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

>>> M = matroids.catalog.Fano().dual()

>>> M.is_cosimple()
True

>>> N = M.delete('a')

>>> N.is_cosimple()
False
```

**is_dependent()**

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

INPUT:
• X – a subset (or any iterable) of the groundset

OUTPUT: boolean

EXAMPLES:

```python
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
```  

```python
>>> 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:

```python
sage: M = matroids.catalog.Wheel4()
sage: M.is_graphic()
True
sage: M = matroids.catalog.U24()
sage: M.is_graphic()
False
```  

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

REFERENCES:


**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:

```python
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
```

```python
>>> 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
```

```python
def 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.
```

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

```python
sage: M = matroids.catalog.Fano()

sage: P = PermutationGroup([[('a', 'b', 'c'), 'd', 'e', 'f', 'g')])

sage: M.is_isomorphism(M, P)
True

sage: N = matroids.catalog.Pappus()

sage: def f(x):
... return x

sage: N.is_isomorphism(M, f)
False
```
There is extensive checking for inappropriate input:

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

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

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

```python
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
```

```python
sage: M.is_isomorphism(M, {0: 1, 1: 2, 2: 3})
Traceback (most recent call last):
...  
ValueError: domain of morphism does not contain groundset of this matroid.
```

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

```python
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
```

```python
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.
```

```python
sage: # needs sage.graphs
sage: M = matroids.CompleteGraphic(3)
```

```python
sage: N = Matroid(bases=['ab', 'ac', 'bc'])
```

```python
sage: f = [0, 1, 2]
```

```python
sage: g = {'a': 0, 'b': 1, 'c': 2}
```

```python
sage: N.is_isomorphism(M, f)
Traceback (most recent call last):
...  
ValueError: the morphism argument does not seem to be an isomorphism.
```

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

```python
>>> from sage.all import *
```
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: M = Matroid(m)
sage: M.is_k_closed(3)
False
sage: M.is_k_closed(4)
True

```python
>>> 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))
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 \((k, \text{certificate}=False)\)

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
- \(\text{certificate}\) – (default: False) boolean; 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 a tuple (boolean, frozenset)

See also:

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

ALGORITHM:

Apply linking algorithm to find small separator.

EXAMPLES:

```python
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)
```

(continues on next page)
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)  # 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 $\sum (weights(e) \text{ for } e \text{ in } S)$.

**INPUT:**
- $X$ – (default: the groundset) a subset (or any iterable) of the groundset
- $weights$ – a 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 oth-
erwise 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:**

```python
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
```

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

1.2. The abstract Matroid class
```python
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: 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.
```

```python
>>> 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), ...
True

>>> M.is_max_weight_coindependent_generic(weights={Integer(0): 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 \(\sum\text{weights}(e)\) for \(e\ in S\).

**INPUT:**

- \(X\) – (default: the groundset) a subset (or any iterable) of the groundset
- \(\text{weights}\) – a 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

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})
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 *

(continues on next page)
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:

```python
>>> 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),
...
True
>>> M.is_max_weight_independent_generic(weights={Integer(0): Integer(1),
...
False
>>> M.is_max_weight_independent_generic(weights={Integer(0): Integer(1),
...
True
>>> M.is_max_weight_independent_generic(weights={Integer(0): Integer(2),
...
Traceback (most recent call last):
...
ValueError: nonnegative weights were expected.
```

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:
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:

```python
sage: M = matroids.catalog.Fano()
sage: M.is_simple()
True

sage: N = M / 'a'
sage: N.is_simple()
False
```

is_sparse_paving()

Return if self is sparse-paving.

A matroid is sparse-paving if the symmetric difference of every pair of circuits is greater than 2.

OUTPUT: boolean

EXAMPLES:

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

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

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
- \(k\) – a 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)
is_ternary \(\text{(randomized_tests}=1)\)

Decide if self is a ternary matroid.

INPUT:

- \text{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 \text{randomized_tests} times. Next, test if a ternary matroid local to some basis is isomorphic to \text{self}.

See also:

\text{M.ternary_matroid()}

EXAMPLES:

```python
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
False
```
is_valid()

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.

OUTPUT: boolean

EXAMPLES:

sage: M = matroids.catalog.Vamos()
sage: 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

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:

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

```python
>>> from sage.all import *
>>> M1 = matroids.Wheel(Integer(3))  # needs sage.graphs
>>> 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.
```

`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
- $k$ – a positive integer

EXAMPLES:

```python
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]
```

```python
sage: m = matrix([[0, 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, 1, 1, 1, 0, 1, 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]
```

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

>>> m = matrix([[Integer(1),Integer(2),Integer(5),Integer(2)],
               ....: [Integer(0),
               ....: Integer(2),Integer(1),Integer(0)],
               ....: [Integer(0), Integer(1), 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(0), Integer(0), Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),
               ....: Integer(1),
               ....: [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0), Integer(1), Integer(1), Integer(1),
               ....: Integer(1)]])

>>> M = Matroid(m)

>>> sorted(M.k_closure({Integer(1),Integer(3)}, Integer(2)))
[0, 1, 2, 3]
```

```python
>>> sorted(M.k_closure({Integer(0),Integer(1)}, Integer(1)))
[0, 1, 3]
```

```python
>>> sorted(M.k_closure({Integer(1),Integer(2)}, Integer(1)))
[1, 2]
```

```python
>>> 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(0), Integer(0), Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),
               ....: Integer(1),
               ....: [Integer(0), Integer(0), Integer(0), Integer(1), Integer(0), Integer(1), 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]
```

```python
>>> 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]
```
lattice_of_flats()
Return the lattice of flats of the matroid.

EXAMPLES:

```python
sage: M = matroids.catalog.Fano()
sage: 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:**

```python
sage: M = matroids.catalog.Fano()
sage: len(list(M.linear_subclasses()))
16
sage: len(list(M.linear_subclasses(line_length=3)))
8
sage: 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:

```python
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=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:

$$\max \{ r_N(S) + r_N(T) - r(N) \mid N = M/I \setminus J, E(N) = S \cup T \} =$$

$$\min \{ r_M(X) + r_M(Y) - r(M) \mid X \subseteq S, Y \subseteq T, E = X \cup Y, X \cap Y = \emptyset \}.$$  

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:**

```python
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)
```
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:

```python
sage: M = matroids.catalog.Fano()
sage: M.loops()
frozenset()  
```

matroid_polytope()

Return the matroid polytope of \( \text{self} \).

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 \( \mathbb{R}^n \). An arbitrary labelling of the groundset by \( \{0, \ldots, n-1\} \) is chosen.

See also:

\( \text{independence_matroid_polytope()} \)

EXAMPLES:

```python
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
```
REFERENCES:

[DLHK2007]

\texttt{max\_coindependent}(X)

Compute a maximal coindependent subset of \(X\).

A set is \textit{coindependent} if it is independent in the dual matroid. A set is coindependent if and only if the complement is \textit{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:

\texttt{M.dual()}, \texttt{M.max\_independent()}

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}
```python
>>> sorted(X)  # random
['a', 'c', 'd', 'f']
>>> M.is_coindendent(X)
True
>>> all(M.is_codependent(X.union([y])) for y in M.groundset() if y not in X)
True
>>> M.max_coindendent(['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:**
```python
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
```

```python
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> X = M.max_independent(['a', 'c', 'd', 'e', 'f'])
>>> M.is_independent(X)
True
>>> 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_coindendent(X=None, weights=None)`

Return a maximum-weight coindendent set contained in X.

The weight of a subset S is \(\sum \text{weights}(e)\) for e in S.

**INPUT:**
- X – (default: the groundset) a subset (or any iterable) of the groundset
- weights – a 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 co-indepedent of all that was selected before.

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{from sage.matroids.advanced import setprint} \\
\text{sage: } & M = \text{matroids.catalog.Fano()} \\
\text{sage: } & X = M.\text{max_weightcoindependent}() \\
\text{sage: } & M.\text{is_cobasis}(X) \\
\text{True} \\
\text{sage: } & wt = \{'a': 1, 'b': 2, 'c': 2, 'd': 1/2, 'e': 1, 'f': 2, 'g': 2\} \\
\text{sage: } & \text{setprint}(M.\text{max_weightcoindependent}(\text{weights}=\text{wt})) \\
\{\text{'b'}, \text{'c'}, \text{'f'}, \text{'g'}\} \\
\text{sage: } & wt = \{'a': 1, 'b': -10, 'c': 2, 'd': 1/2, 'e': 1, 'f': 2, 'g': 2\} \\
\text{sage: } & \text{setprint}(M.\text{max_weightcoindependent}(\text{weights}=\text{wt})) \\
\text{Traceback (most recent call last):} \\
\text{...} \\
\text{ValueError: nonnegative weights were expected.} \\
\text{sage: } & \text{def } \text{wt}(x): \\
\text{... } & \text{return } x \\
\text{...:} \\
\text{sage: } & M = \text{matroids.Uniform}(2, 8) \\
\text{sage: } & \text{setprint}(M.\text{max_weightcoindependent}(\text{weights}=\text{wt})) \\
\{2, 3, 4, 5, 6, 7\} \\
\text{sage: } & \text{setprint}(M.\text{max_weightcoindependent}()) \\
\{0, 1, 2, 3, 4, 5\} \\
\text{sage: } & M.\text{max_weightcoindependent}(\text{X=}[], \text{weights=}\{\}) \\
\text{frozenset()} \\
\end{align*}
\]
max_weight_independent \( (X=None, \text{weights}=None) \)

Return a maximum-weight independent set contained in a subset.

The weight of a subset \( S \) is \( \sum(\text{weights}(e) \text{ for } e \text{ in } S) \).

**INPUT:**

- \( X \) – (default: the groundset) a subset (or any iterable) of the groundset
- \( \text{weights} \) – a 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:**

```python
>>> M = matroids.catalog.Fano()
>>> X = M.max_weight_independent()
>>> M.is_basis(X)
True
```

```python
>>> wt = {'a': Integer(1), 'b': Integer(2), 'c': Integer(2), 'd': Integer(1)/2, 'e': Integer(1), ...
        'f': Integer(2), 'g': Integer(2)}
>>> setprint(M.max_weight_independent(weights=wt))
{'b', 'f', 'g'}
```

```python
>>> def wt(x):
    ... return x
...:
>>> M = matroids.Uniform(2, 8)
>>> setprint(M.max_weight_independent(weights=wt))
{6, 7}
>>> setprint(M.max_weight_independent())
{0, 1}
```

```python
>>> M.max_weight_coindependent(X=[], weights={})
frozenset()
```

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

M = matroids.catalog.Fano()
X = M.max_weight_independent()
M.is_basis(X)
```

```python
>>> wt = {'a': 1, 'b': 2, 'c': 2, 'd': 1/2, 'e': 1,
        ...: 'f': 2, 'g': 2}
>>> setprint(M.max_weight_independent(weights=wt))
{'b', 'f', 'g'}
```

```python
>>> def wt(x):
    ... return x
...:
>>> M = matroids.Uniform(2, 8)
>>> setprint(M.max_weight_independent(weights=wt))
{6, 7}
>>> setprint(M.max_weight_independent())
{0, 1}
```

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

M = matroids.catalog.Fano()
X = M.max_weight_independent()
M.is_basis(X)
```
minorminor (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
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'},
    {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.
```
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:
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 $\{b, f\}$:

```python
sage: M = matroids.catalog.S8()
sage: N = M.delete('h')
sage: frozenset('bf') in N.modular_cut(['cg', 'ae'])
True
```

The modular cut of the full groundset is equal to just the groundset:

```python
sage: M = matroids.catalog.Fano()
sage: M.modular_cut([M.groundset()]).difference([frozenset(M.groundset())])
set()
```

\section*{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:**

```python
sage: from sage.matroids.basis_matroid import BasisMatroid
sage: M = BasisMatroid(Matroid(circuits=[[1, 2, 3], [3, 4, 5], [1, 2, 4, 5]]))
```

(continues on next 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)\}

```python
>>> from sage.all import *
>>> from sage.matroids.basis_matroid import BasisMatroid

>>> M = BasisMatroid(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())  # 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(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), (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), (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())
[]

sage: 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())
[]

(continues on next page)
sage: [sorted(X) for X in matroids.catalog.P6().nonbases()]
[['a', 'b', 'c']]
```python
def main():
    from sage.all import *
    M = matroids.catalog.Fano().dual()
    for C in M.noncospanning_cocircuits():
        sorted(C)
    sorted(
        [sorted(C) for C in M.noncospanning_cocircuits()])
    nonspanning_circuit_closures()  # returns dictionary of nonspanning circuit closures
    nonspanning_circuit_closures()[2]  # returns 7 closures
    nonspanning_circuit_closures()[3]  # Key error

if __name__ == '__main__':
    main()
```

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:

```python
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
```

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:

```python
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']]
```
```python
>>> 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:**

```python
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']]
```

```python
>>> 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:**

```python
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}}}
```

```python
sage: G = SymmetricGroup(3);
#... needs sage.groups
```

(continues on next page)
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

sage: # needs sage.groups
sage: G = SymmetricGroup(Integer(3));  # needs sage.groups
sage: OSG = M.orlik_solomon_algebra(QQ, invariant=G)  # needs sage.groups

partition()

Return a minimum number of disjoint independent sets that covers the groundset.

OUTPUT: a 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()
[]
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) a list containing a basis. 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 – An integer specifying positioning method.
  - 0: default positioning
  - 1: use pos_dict if it is not None
  - 2: Force directed (Not yet implemented).

• pos_dict: A 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 repre-
sentation of the matroid.

EXAMPLES:

```
sage: M = matroids.catalog.Fano()
sage: G = M.plot()  # needs sage.plot sage.rings.finite_rings
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
```

**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
```
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 – a 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 – a list of 4 elements [xmin, xmax, ymin, ymax]

EXAMPLES:

```python
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), '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
```

(continues on next page)
\textbf{\texttt{simplify()}}

Return the simplification of the matroid.

A matroid is \textit{simple} if it contains no circuits of length 1 or 2. The \textit{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).

OUT\textsc{put}: matroid

See also:
\texttt{M.is_simple()}, \texttt{M.loops()}, \texttt{M.circuit()}, \texttt{M.cosimplify()}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: M = matroids.catalog.Fano().contract('a')
sage: M.size() - M.simplify().size()
3
\end{verbatim}

\textbf{\texttt{size()}}

Return the size of the groundset.

OUT\textsc{put}: integer

\textbf{EXAMPLES:}

\begin{verbatim}
sage: M = matroids.catalog.Vamos()
sage: M.size()
8
\end{verbatim}

\begin{verbatim}
>>> from sage.all import *
>>> M = matroids.catalog.Vamos()
>>> M.size()
8
\end{verbatim}

\textbf{\texttt{ternary_matroid\;(randomized\_tests=1, verify=True)}}

Return a ternary matroid representing \texttt{self}, if such a representation exists.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{randomized\_tests} – (default: 1) an integer; the number of times a certain necessary condition for being ternary is tested, using randomization
  \item \texttt{verify} – boolean (default: True); if True, any output will be a ternary matroid representing \texttt{self}; if \texttt{False}, any output will represent \texttt{self} if and only if the matroid is ternary
\end{itemize}
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:

```python
sage: M = matroids.catalog.Fano()
sage: M.ternary_matroid() is None
needs sage.graphs
True
```

```python
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:

```python
sage: M = matroids.catalog.Fano()
sage: N = M.truncation()
sage: N.is_isomorphic(matroids.Uniform(2, 7))
```

```
true
```

```python
from sage.all import *
```
**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(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 \(E = \bigcup E_i\). 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 – a matroid or a list of matroids
OUTPUT: an instance of `MatroidUnion`

EXAMPLES:

```python
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-
```

```python
>>> 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:

```python
sage: M = matroids.catalog.BetsyRoss()
sage: M.whitney_numbers()
[1, -11, 35, -25]
```

```python
>>> 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:

```python
sage: M = matroids.catalog.BetsyRoss()
sage: M.whitney_numbers2()
[1, 11, 20, 1]
```

```python
>>> from sage.all import *
>>> M = matroids.catalog.BetsyRoss()
>>> M.whitney_numbers2()
[1, 11, 20, 1]
```
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`
Chapter 2. Database of matroids

- 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`
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.+

AUTHORS:
- Giorgos Mousa (2023-12-08): initial version

sage.matroids.database_collections.AllMatroids(n, r=None, type='all')

Return an iterator 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 ≤ r ≤ 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'

OUTPUT: an iterator over matroids

EXAMPLES:

```
sage: for M in matroids.AllMatroids(2):
    #...
    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
```

(continues on next page)
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"): ˓→
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"): ˓→
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

(continues on next 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"

>>> 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"

REFERENCES:
The underlying database was retrieved from Yoshitake Matsumoto's Database of Matroids; see [Mat2012].
sage.matroids.database_collections.BrettellMatroids()

Return an iterator 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
```

See also:

Matroid catalog, under Brettell's matroid collection.

sage.matroids.database_collections.OxleyMatroids()

Return an iterator 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
```

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()

Return an iterator 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

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

```python
>>> from sage.all import *
>>> M = matroids.catalog.A9(); M
```

(continues on next page)
A9: Quaternary matroid of rank 3 on 9 elements
>>> M.is_valid()
True

sage.matroids.database_matroids.AF12(groundset=None)

Return the matroid $AF_12$.

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

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:

- $n$ – a positive integer; the dimension of the projective space. This is one less than the rank of the resulting matroid.
- $q$ – a positive integer that is a prime power; the order of the finite field
- $x$ – a 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 – a 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: M = 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)

(continues on next page)
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:

```python
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
```

 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:

```python
>>> 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
>>> 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.
\[
\begin{align*}
\{c', e', h'\}, \{f', g', h'\},
3: \{\{a', b', c', d', e', f', g', h'\}\}
\end{align*}
\]

sage: M.is_valid()
True

```python
>>> 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 \(AG\)\(_{23}\)\text{minusDY}.

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

```python
>>> 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
```

\(AG(3,2)\) 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:
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', d', f', h'\},
{b', e', g', h'\}, \{c', d', e', h'\}, \{c', f', g', h'\},
{d', e', f', g'\}\}

>>> setprint(M.noncospanning_cocircuits())
\{|\{a', b', c', d', e', f', g', h'\}\}

>>> 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

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
```python
>>> 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:

```python
sage: M = matroids.catalog.AK12(); M
AK12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
False
```
Matroid Theory, Release 10.4

```python
>>> 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:

```python
sage: M = matroids.catalog.BB9gDY(); M
Segment cosegment exchange on BB9: Quaternary matroid of rank 5 on 9 elements
sage: 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:

```python
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
```

(continues on next page)
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:

```python
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()))
```

```python
sage: 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:

```python
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()))
```

```python
sage: 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:

- $n$ – an 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:

```python
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]))
{0, 1, 2, 4, 5, 7}
sage: M.is_valid()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = matroids.catalog.D10(); M
D10: Quaternary matroid of rank 4 on 10 elements
>>> M.has_minor(matroids.catalog.TQ8())
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = matroids.catalog.D16(); M
D16: Quaternary matroid of rank 4 on 10 elements
>>> M.has_minor(matroids.catalog.TQ8())
True
```

sage.matroids.database_matroids.D10(groundset=None)

Return the matroid \( D_{10} \). 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
```

sage.matroids.database_matroids.D16(groundset=['abcdefgijklmnop'])

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:

```python
sage: M = matroids.catalog.D16(); M
D16: Binary matroid of rank 8 on 16 elements, type (0, 0)
sage: M.is_valid()
True
```

```python
>>> 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='abcdefghijkmnopqrstuvwxyz')

Return the matroid of the extended binary Golay code.

EXAMPLES:

```python
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
```

```python
>>> 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='abcdefghijl')

Return the matroid of the extended ternary Golay code.

This is the unique Steiner system $S(5, 6, 12)$.

EXAMPLES:

```python
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
```

2.3. Database of matroids
>>> 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 $M_{12}$:

```
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
```

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)
```

```
sage: N.is_isomorphic(matroids.catalog.AG23())
True
```

```
sage: 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)])
```

```
sage: N.is_isomorphic(matroids.catalog.P8())
True
```

See also:

GolayCode

REFERENCES:
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:

```python
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'}}
```

REFERENCES:


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 $FF12$.
```

146 Chapter 2. Database of matroids
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
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
FK12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

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
sage: M.is_isomorphic(M.dual())
True
```

```
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
sage: M.is_isomorphic(M.dual())
True
```

2.3. Database of matroids
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:

```python
sage: M = matroids.catalog.FM14(); M
FM14: Quaternary matroid of rank 7 on 14 elements
sage: 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:

```python
sage: M = matroids.catalog.FN9(); M
FN9: Quaternary matroid of rank 3 on 9 elements
sage: 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:

```python
sage: M = matroids.catalog.FP10(); M
FP10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```
sage.matroids.database_matroids.FP12(groundset=None)

Return the matroid $FP_{12}$.

An excluded minor for $K_2$- and $G$-representable matroids (and $H_2$- and $GF(5)$-representable matroids). UPF is $W$. Self-dual.

EXAMPLES:

```python
sage: M = matroids.catalog.FP12(); M
FP12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.FQ12(groundset=None)

Return the matroid $FQ_{12}$.

An excluded minor for $P_4$-representable matroids. UPF is $PT$. Has a $PP_9$-minor (contract 4 and 7, delete 6) and $FF_{10}$-minor (contract ‘c’ and delete ‘d’).

EXAMPLES:

```python
sage: FQ12 = matroids.catalog.FQ12(); FQ12
FQ12: Quaternary matroid of rank 6 on 12 elements
sage: P9 = matroids.catalog.PP9()
sage: P9.is_isomorphic(FQ12.contract([4,7]).delete(6))
True
sage: FF10 = matroids.catalog.FF10()
sage: FF10.is_isomorphic(FQ12.contract('c').delete('d'))
True
```

sage.matroids.database_matroids.FR12(groundset=None)

Return the matroid $FR_{12}$.

An excluded minor for $K_2$-representable matroids. UPF is $P_4$. Self-dual.

EXAMPLES:

```python
sage: M = matroids.catalog.FR12(); M
FR12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```
Matroid Theory, Release 10.4

```python
>>> 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 $FS_{12}$.

An excluded minor for $G$-representable matroids (and $GF(5)$-representable matroids). Rank 5. UPF is $GF(4)$.

EXAMPLES:

```python
sage: M = matroids.catalog.FS12(); M
FS12: Quaternary matroid of rank 5 on 12 elements
sage: M.rank()
5
```

```python
>>> 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 $FT_{10}$.

An excluded minor for $G$-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $GF(4)$.

EXAMPLES:

```python
sage: M = matroids.catalog.FT10(); M
FT10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```python
>>> 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 $FU_{10}$.

An excluded minor for $P_4$-representable matroids. UPF is $G$. Self-dual.

EXAMPLES:

```python
sage: M = matroids.catalog.FU10(); M
FU10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

```python
>>> from sage.all import *
>>> M = matroids.catalog.FU10(); M
FU10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
True
```

(continues on next page)
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:

```python
sage: M = matroids.catalog.FV14(); M
FV14: Quaternary matroid of rank 7 on 14 elements
sage: 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:

```python
sage: M = matroids.catalog.FX9(); M
FX9: Quaternary matroid of rank 4 on 9 elements
sage: 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:

```python
sage: M = matroids.catalog.FY10(); M
FY10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
False
```
sage.matroids.database_matroids.FZ10(groundset=None)

Return the matroid $FZ_{10}$.


EXAMPLES:

```python
sage: M = matroids.catalog.FZ10(); M
FZ10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.FZ12(groundset=None)

Return the matroid $FZ_{12}$.


EXAMPLES:

```python
sage: M = matroids.catalog.FZ12(); M
FZ12: Quaternary matroid of rank 6 on 12 elements
sage: 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:

```python
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)
```

(continues on next page)
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
```

It is also the projective plane of order two, i.e. $PG(2, 2)$:

```
sage: M.is_isomorphic(matroids.PG(2, 2))
True
```

$F_7$ is isomorphic to the unique binary 3-spike:

```
sage: M.is_isomorphic(matroids.Z(3))
True
```

REFERENCES:

[Oxl2011], p. 643.
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()))
sage: F7D.delete(e).is_isomorphic(K2_3)
True
```

REFERENCES:

[Oxl2011], p. 643.
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)

**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
```

(continues on next page)
J: Ternary matroid of rank 4 on 8 elements, type 0-

```python
>>> setprint(M.truncation().nonbases())
[['a', 'b', 'f'], ['a', 'c', 'g'], ['a', 'd', 'h']]
```

```python
>>> M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True
```

```python
>>> M.has_minor(matroids.CompleteGraphic(Integer(4)))
False
```

```python
>>> M.is_valid()
True
```

```python
>>> M.automorphism_group().is_transitive()
False
```

REFERENCES:

[Oxl2011], p. 650.

sage.matroids.database_matroids.K33(groundset='abcdeghi')

Return the graphic matroid $M(K_{3,3})$.

$M(K_{3,3})$ is an excluded minor for the class of cographic matroids.

EXAMPLES:

```python
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()
```

```python
sage: G2 = matroids.catalog.K33dual().automorphism_group()
```

```python
sage: G1.is_isomorphic(G2)
True
```

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

```python
>>> # needs sage.graphs
```

```python
>>> M = matroids.catalog.K33(); M
M(K3, 3): Regular matroid of rank 5 on 9 elements with 81 bases
```

```python
>>> M.is_valid()
True
```

```python
>>> G1 = M.automorphism_group()
```

```python
>>> G2 = matroids.catalog.K33dual().automorphism_group()
```

```python
>>> G1.is_isomorphic(G2)
True
```

REFERENCES:


sage.matroids.database_matroids.K33dual(groundset='abcdeghi')

Return the matroid $M \ast (K_{3,3})$, represented over the regular partial field.

The matroid $M \ast (K_{3,3})$ 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:

```python
sage: # needs sage.graphs
sage: M = matroids.catalog.K33dual(); M
```

(continues on next page)
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
>>> M.automorphism_group().is_transitive()
True

REFERENCES:
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 $M(W_3)$, 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

2.3. Database of matroids
It has a transitive automorphism group:

```
sage: 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
```

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'
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

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

sage.matroids.database_matroids.KP8(groundset=None)
Return the matroid KP8.
An excluded minor for K2-representable matroids. UPF is G. Self-dual. Uniquely GF(5)-representable. (An excluded minor for H2-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.\texttt{KQ9} (\texttt{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
>>> 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.\texttt{KR9} (\texttt{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:

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 *
>>> KP8 = matroids.catalog.KP8()
Matroid Theory, Release 10.4

sage.matroids.database_matroids.KT10(groundset=None)

Return the matroid $KT_{10}$.

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
```

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
```

2.3. Database of matroids 161
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)]
sage: K4ext = Matroid(bases=Bext)
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: M.contract(e).is_isomorphic(K4ext)
True
```
sage.matroids.database_matroids.N1(groundset='abcdefghij')
Return the matroid \(N_1\), represented over \(F_3\).
\(N_1\) is an excluded minor for the dyadic matroids.

EXAMPLES:

```python
>>> 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='abcdefgijkl')
Return the matroid \(N_2\), represented over \(F_3\).
\(N_2\) is an excluded minor for the dyadic matroids.

EXAMPLES:

```python
>>> 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:

```python
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
```

```python
>>> from sage.all import *
>>> N3 = matroids.catalog.N3(); N3
N3: Ternary matroid of rank 7 on 14 elements
>>> 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:

```python
sage: M = matroids.catalog.N3pp(); M
N3pp: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
True
```

```python
>>> from sage.all import *
>>> M = matroids.catalog.N3pp(); M
N3pp: 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:

```python
sage: N4 = matroids.catalog.N4(); N4
N4: Ternary matroid of rank 8 on 16 elements, type 0+
sage: N4.is_isomorphic(N4.dual())
```

(continues on next page)
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

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
NonDesargues: Matroid of rank 3 on 10 elements with 9 nonspanning circuits
>>> M.is_valid()
True
>>> M.automorphism_group().is_transitive()
False

2.3. Database of matroids
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:

```python
sage: from sage.matroids.advanced import setprint
groundset='abcdefg')

non-Fano: 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(4))
True
M.delete('g').is_isomorphic(matroids.CompleteGraphic(4))
False
```

REFERENCES:


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:

```python
sage: 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 $\mathcal{W}_3$:
Matroid Theory, Release 10.4

```python
sage: import random
sage: e = random.choice(list(M.groundset()))
sage: N = M.contract(e)
>>> from sage.all import *
>>> import random
>>> e = random.choice(list(M.groundset()))
>>> N = M.contract(e)
```

**REFERENCES:**


```python
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:**

```python
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
```

```python
>>> from sage.all import *
>>> M = matroids.catalog.NonPappus(); M
>>> 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
```

2.3. Database of matroids 167
Matroid Theory, Release 10.4

REFERENCES:

[Oxl2011], p. 655.

sage.matroids.database_matroids.NonVamos(groundset=None)

Return the non-Vámós matroid.

The non-Vámós matroid, or $V_8^+$ is an 8-element matroid of rank 4. It is a tightening of the Vámós 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
4: \{\{a', b', c', d', e', f', g', h'\}\}
3: \{\{a', b', c', d', e', f', g', h'\}, \{c', d', e', f', \{c', d', g', h'\}, \{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 *
>>> 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:

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:

```python
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
```

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:

```python
sage: M = matroids.catalog.OW14(); M
OW14: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
True
```

2.3. Database of matroids

```python
>>> from sage.all import *
>>> M = matroids.catalog.OW14(); M
>>> 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: M.is_valid()
True
sage: M.automorphism_group().is_transitive()
False
```

References:


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
```

```
REFERENCES:


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:

```python
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
```

REFERENCES:

[Oxl2011], p. 650-1.

sage.matroids.database_matroids.P8p(groundset=None)

Return the matroid $P_8^-$.

$P_8^-$ is obtained by relaxing one of the disjoint circuit-hyperplanes of $P_8$. An excluded minor for 2-regular matroids. UPF is $K_2$. Self-dual.

EXAMPLES:

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

>>> M = matroids.catalog.P8p(); M
P8-: Quaternary matroid of rank 4 on 8 elements

>>> M.is_isomorphic(M.dual())
True

>>> Matroid(matrix=random_matrix(GF(Integer(4), 'a'), ncols=Integer(5), nrows=Integer(5))).has_minor(M)
False

>>> M.bicycle_dimension()
2

```

REFERENCES:

[Oxl2011], p. 650-1.
P8-: Quaternary matroid of rank 4 on 8 elements

```
>>> M.is_isomorphic(M.dual())
True
```

```
>>> from sage.matroids.advanced import *
>>> M = matroids.catalog.P8pp(); M
P8pp: Matroid of rank 4 on 8 elements with 8 nonspanning circuits
```

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: M = matroids.catalog.P8pp(); M
P8pp: Matroid of rank 4 on 8 elements with 8 nonspanning circuits
```

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

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)
```

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:**

- \(n\) – a positive integer; the dimension of the projective space. This is one less than the rank of the resulting matroid.
- \(q\) – a positive integer that is a prime power; the order of the finite field
- \(x\) – a string (default: None); the name of the generator of a non-prime field, used for non-prime fields. If not supplied, 'x' is used.
- \(groundset\) – a string (optional); the groundset of the matroid

**OUTPUT:** a linear matroid whose elements are the points of \(PG(n, q)\)

**EXAMPLES:**

```python
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
```

**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:**

```python
>>> 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) - 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
```

2.3. Database of matroids
REFERENCES:

[Oxl2011], p. 659.

sage.matroids.database_matroids.PK10(groundset=None)

Return the matroid $PK_{10}$.

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
```

sage.matroids.database_matroids.PP10(groundset=None)

Return the matroid $PP_{10}$.

An excluded minor for $P_4$-representable matroids. UPF is $U_{1}^{(3)}$. 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
```
sage.matroids.database_matroids.PP9(groundset=None)

Return the matroid PP9.


**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
```

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
```

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

>>> M = matroids.catalog.Pappus(); M
```

(continues on next page)
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 $\Psi_r$.

The rank-$r$ free swirl; defined for all $r \geq 3$.

INPUT:

- $r$ – an integer ($r \geq 3$); the rank of the matroid
- $groundset$ – a string (optional); the groundset of the matroid

OUTPUT: matroid ($\Psi_r$)

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 $\Psi_r$ 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
sage: r = random.choice(range(Integer(4), Integer(8)))
sage: M = matroids.Psi(r)
sage: M.is_4connected()
False
\[ \Psi_3 \cong U_{3,6} \]

```python
sage: M = matroids.Psi(3)
sage: M.is_isomorphic(matroids.catalog.U36())
True
```

It is identically self-dual with a transitive automorphism group:

```python
>>> from sage.all import *
>>> M = matroids.Psi(Integer(3))
>>> M.is_isomorphic(matroids.catalog.U36())
True
```

REFERENCES:

[Oxl2011], p. 664.

sage.matroids.database_matroids.Q10(groundset='abcdefghij')

Return the matroid \( Q_{10} \), represented over \( F_4 \).

\( Q_{10} \) is a 10-element, rank-5, self-dual matroid. It is representable over \( F_3 \) and \( F_4 \), and hence is a sixth-roots-of-unity matroid. \( Q_{10} \) is a splitter for the class of sixth-root-of-unity matroids.

EXAMPLES:

```python
sage: M = matroids.catalog.Q10(); M
Q10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
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 \( U_{2,5} \):

```python
>>> 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
```
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:**

```python
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}]
```

```python
sage: M.nonspanning_circuits() == M.noncospanning_cocircuits()
False
```

```python
sage: M.automorphism_group().is_transitive()
False
```

**REFERENCES:**

[Oxl2011], p. 641.

The matroid $Q_8$ is an 8-element matroid of rank-4. It is a smallest non-representable matroid.

**EXAMPLES:**

```python
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'}],
{a', c', d', f'}, {a', d', g', h'}, {a', e', f', h'}],
```
{b', c', d', g'}. 

sage: M.is_valid() # long time
True

sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
True

sage: 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:
Every single-element deletion is isomorphic to $M(K_{3,3})$, and every single-element contraction is isomorphic to $M^*(K_{3,3})$:  

```python
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
```

Check the splitter property:

```python
sage: matroids.catalog.R10().linear_extensions(simple=True)
[]
```
>>> from sage.all import *
>>> matroids.catalog.R10().linear_extensions(simple=True)
[]

REFERENCES:

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

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_valid()
True
sage: M.automorphism_group().is_transitive()
False

2.3. Database of matroids

181
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

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

(continues on next page)
**sage**: M.contract(e).is_isomorphic(F7m)
```
True
```

```python
>>> 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.

```python
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 $R_9$, $Q_3(GF(3)\times mes)$, and $AG(2,3)$. It is not graphic, not cographic, and not regular.

EXAMPLES:

```python
sage: M = matroids.catalog.R9(); M
```
```
R9: Matroid of rank 3 on 9 elements with 15 nonspanning circuits
```
```python
sage: M.is_valid()
```
```
True
```
```python
sage: len(M.nonspanning_circuits())
```
```
15
```
```python
sage: M.is_simple() and M.is_ternary()
```
```
True
```
```python
sage: M.automorphism_group().is_transitive()
```
```
False
```

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

>>> M = matroids.catalog.R9(); M
```
```
R9: Matroid of rank 3 on 9 elements with 15 nonspanning circuits
```
```python
>>> M.is_valid()
```
```
True
```
```python
>>> len(M.nonspanning_circuits())
```
```
15
```
```python
>>> M.is_simple() and M.is_ternary()
```
```
True
```
```python
>>> M.automorphism_group().is_transitive()
```
```
False
```

REFERENCES:


```python
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 Tutte group 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:

2.3. Database of matroids
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 $R^B_9$.

The matroid $R^B_9$ is not representable over any field, yet none of the cross-ratios in its Tutte group equal 1. It is one of the 4 matroids on at most 9 elements with this property, the others being $R^*_9$, $R^A_9$ and $R^A_9^*$.

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
>>> 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
>>> 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:
\begin{verbatim}

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, 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, True]

REFERENCES:

sage.matroids.database_matroids.Sp8(groundset=None)

Return the matroid \(Sp8\).

An excluded minor for \(G\)- and \(K_3\)-representable matroids. UPF is \(U_1^{(2)}\). Self-dual. (An excluded minor for \(H_2\)- and \(GF(5)\)-representable matroids.)

EXAMPLES:
\end{verbatim}
sage: M = matroids.catalog.Sp8(); M
Sp8: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True

sage.matroids.database_matroids.Sp8pp(groundset=None)

Return the matroid $Sp_8 = $. 
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

sage.matroids.database_matroids.Spike(r=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:
- $r$ – an integer ($r \geq 3$); the rank of the spike
- $t$ – boolean (default: True); whether the spike is tipped
- $C_3$ – a list (default: []); a list of extra nonspanning circuits. The default (i.e. the empty list) results in a free $r$-spike
- $groundset$ – a 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

(continues on next page)
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],
                           [y1, 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
```
Deleting any element gives a self-dual matroid. The tipless free spike (i.e., when $C_3$ is empty) is identically self-dual:

```python
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
```

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:

```python
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())
True
sage: M.automorphism_group().is_transitive()
True
```

```python
>>> 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
```
REFERENCES:


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:

```python
 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 *

```python
>>> 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:


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:

```python
 sage: M = matroids.catalog.TK10(); M
 TK10: Quaternary matroid of rank 5 on 10 elements
 sage: M.is_isomorphic(M.dual())
 True
```

2.3. Database of matroids
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:

```python
>>> from sage.all import *
>>> M = matroids.catalog.TQ10(); M
TQ10: Quaternary matroid of rank 5 on 10 elements
>>> M.is_isomorphic(M.dual())
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:

```python
>>> 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

sage.matroids.database_matroids.TQ9p(groundset=None)
Return the matroid $TQ9^p$.
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
TQ9p: Quaternary matroid of rank 4 on 9 elements
sage: 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

2.3. Database of matroids
sage.matroids.database_matroids.TernaryDowling3 (groundset='abcdefghi')
Return the matroid $Q_3(\mathbb{GF}(3)^\times)$, represented over $\mathbb{GF}(3)$.

The matroid $Q_3(\mathbb{GF}(3)^\times)$ 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
```

REFERENCES:


sage.matroids.database_matroids.Terrahawk (groundset='abcdefghiklmnop')
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
```

REFERENCES:

[CMO2011]
Return the matroid $\Theta_n$.

Defined for all $n \geq 2$, $\Theta_2 \cong U_{1,2} \oplus U_{1,2}$ and $\Theta_3 \cong M(K_4)$.

**INPUT:**

- $n$ – an integer ($n \geq 2$); the rank of the matroid
- groundset – a string (optional); the groundset of the matroid

**OUTPUT:** matroid ($\Theta_n$)

**EXAMPLES:**

```python
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
```

$\Theta_n$ is self-dual; identically self-dual if and only if $n = 2$:

```python
>>> 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
>>> M = matroids.Theta(Integer(3))
>>> M.is_isomorphic(matroids.catalog.K4())
True
```

For $n \leq 3$, its automorphism group is transitive, while for $n \geq 4$ it is not:

```python
>>> 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
```
```
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
```

REFERENCES:


```
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
```

REFERENCES:

[Hoc]

```
sage.matroids.database_matroids.TippedFree3spike(groundset=None)

Return the tipped free 3-spike.

Unique 3-connected extension of \( U_{36} \). 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
```
```
```
sage.matroids.database_matroids.U24 (groundset='abcd')

The uniform matroid of rank 2 on 4 elements.
The 4-point line; isomorphic to $W^2$, the rank-2 whirl. The unique excluded minor for the class of binary matroids.

EXAMPLES:

```python
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'
```

$U_{2,4}$ is isomorphic to $W^2$:

```python
sage: W2 = matroids.Whirl(2)
sage: W2.is_isomorphic(M)
True
```

identically self-dual:

```python
sage: M.equals(M.dual())
True
```

and 3-connected:

```python
sage: M.is_3connected()
True
```
>>> from sage.all import *
>>> M.is_3connected()
True

REFERENCES:

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:

```python
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:

```python
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()
(continues on next page)
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:

```python
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)  # True
sage: U36.equals(U36.dual())  # True
sage: U36.automorphism_group().structure_description()
'S6'
```

REFERENCES:

[Oxl2011], p. 642.

sage.matroids.database_matroids.UA12(groundset=None)

Return the matroid $UA_{12}$.

An excluded minor for $G$-representable matroids (and $GF(5)$-representable matroids). Not self-dual. UPF is $GF(4)$.

EXAMPLES:

```python
sage: M = matroids.catalog.UA12(); M
UA12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
False
```

REFERENCE:

[Oxl2011], p. 640.
UA12: Quaternary matroid of rank 6 on 12 elements

```python
>>> M.is_isomorphic(M.dual())
False
```

```
sage.matroids.database_matroids.UG10(groundset=None)

Return the matroid $U_9$.  
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:

```python
sage: M = matroids.catalog.UG10(); M
UG10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
True
```

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

```python
>>> M = matroids.catalog.UG10(); M
UG10: Quaternary matroid of rank 5 on 10 elements
```

```python
>>> M.is_isomorphic(M.dual())
True
```

```
sage.matroids.database_matroids.UK10(groundset=None)

Return the matroid $U_{10}$.  
An excluded minor for $G$-representable matroids (and $GF(5)$-representable matroids).  Not self-dual.  UPF is $GF(4)$.

EXAMPLES:

```python
sage: M = matroids.catalog.UK10(); M
UK10: Quaternary matroid of rank 5 on 10 elements
sage: M.is_isomorphic(M.dual())
False
```

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

```python
>>> M = matroids.catalog.UK10(); M
UK10: Quaternary matroid of rank 5 on 10 elements
```

```python
>>> M.is_isomorphic(M.dual())
False
```

```
sage.matroids.database_matroids.UK12(groundset=None)

Return the matroid $U_{12}$.  
An excluded minor for $G$-representable matroids (and $GF(5)$-representable matroids).  Self-dual.  UPF is $I$.

EXAMPLES:

```python
sage: M = matroids.catalog.UK12(); M
UK12: Quaternary matroid of rank 6 on 12 elements
sage: M.is_isomorphic(M.dual())
True
```

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

```python
>>> M = matroids.catalog.UK12(); M
UK12: Quaternary matroid of rank 6 on 12 elements
```

```python
>>> 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:

```python
>>> 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 \((Q(a,b), < -1, a, b, a - 1, b - 1, a - b, a + 1, ab + b - 1, ab - b + 1>)\).

EXAMPLES:

```python
>>> 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:

```python
>>> 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 $UT_{10}$.

An excluded minor for $G$-representable matroids (and $GF(5)$-representable matroids). Self-dual. UPF is $I$.

EXAMPLES:

```python
>>> from sage.all import *
>>> M = matroids.catalog.UT10(); M
UT10: Quaternary matroid of rank 5 on 10 elements
```

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:

- $r$ – a nonnegative integer; the rank of the uniform matroid
- $n$ – a nonnegative integer; the number of elements of the uniform matroid
- groundset – a string (optional); the groundset of the matroid

OUTPUT: the uniform matroid $U_{r,n}$

EXAMPLES:

```python
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}}}
```

(continues on next page)
Check that bug Issue #15292 was fixed:

```python
sage: M = matroids.Uniform(4,4)
sage: len(M.circuit_closures())
0
```

REFERENCES:

[Oxl2011], p. 660.

sage.matroids.database_matroids.VP14(groundset=None)

Return the matroid $VP_{14}$.


EXAMPLES:

```python
sage: M = matroids.catalog.VP14(); M
VP14: Quaternary matroid of rank 7 on 14 elements
sage: M.is_isomorphic(M.dual())
False
```

sage.matroids.database_matroids.Vamos(groundset=None)

Return the Vamos matroid, represented as circuit closures.

The Vamos matroid, or Vamos 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:

```python
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'}}}
```

(continues on next page)
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

REFERENCES:


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 $\zeta$ is solution to $x^2 - x + 1 = 0$ and $a$ is an indeterminate.

EXAMPLES:

```python
sage: M = matroids.catalog.WQ8(); M
WQ8: Quaternary matroid of rank 4 on 8 elements
sage: M.is_isomorphic(M.dual())
True
```

sage.matroids.database_matroids.Wheel(r, field=None, ring=None, groundset=None)

Return the rank-$r$ wheel.

INPUT:
• \( r \) – a positive integer; the rank of the desired matroid
• ring – any ring; if provided, output will be a linear matroid over the ring or field ring. If the ring is \( \mathbb{Z} \), then output will be a regular matroid.
• field – any field; same as ring, but only fields are allowed
• groundset – a string (optional); the groundset of the matroid

OUTPUT: the rank-\( r \) wheel matroid, represented as a regular matroid

EXAMPLES:

```python
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+
```

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:

```python
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 *
>>> 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+
```
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:
sage.matroids.database_matroids.Whirl(r, groundset=None)
Return the rank-r whirl.
The whirl is the unique relaxation of the wheel.
INPUT:
• r – a positive integer; the rank of the desired matroid.
• groundset – a string (optional); the groundset of the matroid
OUTPUT: the rank-r whirl matroid, represented as a ternary matroid
EXAMPLES:
M = matroids.Whirl(5); M
Whirl(5): Ternary matroid of rank 5 on 10 elements, type 0-
M.is_valid()
True
M.tutte_polynomial()
For $r \geq 3$, the whirl is self-dual but not identically self-dual:

```python
sage: import random
sage: r = random.choice(range(3, 10))
```

```python
sage: M = matroids.Whirl(r)
```

```python
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
```

```
True
```

Except for $W^2$, which is isomorphic to $U_{2,4}$, these matroids have non-transitive automorphism groups:

```python
sage: r = random.choice(range(3, 8))
```

```python
sage: M = matroids.Whirl(r)
```

```python
sage: 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 $\begin{bmatrix} -1 & 0 & x \\ 1 & -1 & 0 \\ 0 & 1 & -1 \end{bmatrix}$

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:

```python
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
```

For all elements $e$, neither $\mathcal{W}_3 \setminus \{e\}$ nor $\mathcal{W}_3 / \{e\}$ is 3-connected:

```python
>>> 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:

```python
sage: M = matroids.catalog.Whirl4(); M
Whirl(4): Ternary matroid of rank 4 on 8 elements, type 0+
sage: M.is_valid()
True
sage: M.is_isomorphic(M.dual()) and not M.equals(M.dual())
```

(continues on next page)
True
sage: M.automorphism_group().is_transitive()
False

```python
>>> 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.

\[\text{sage}: \text{matroids.database_matroids.XY13(groundset=None)}\]

Return the matroid \(XY1\)\(_{13}\).

An excluded minor for \(G\)-representable matroids (and \(GF(5)\)-representable matroids). UPF is \(GF(4)\).

EXAMPLES:

```python
sage: M = matroids.catalog.XY13(); M
XY13: Quaternary matroid of rank 6 on 13 elements

sage: M.is_3connected()
True
```

\[\text{sage}: \text{matroids.database_matroids.Z(r, t=True, groundset=None)}\]

Return the unique rank-\(r\) binary spike.

Defined for all \(r \geq 3\).

INPUT:

- \(r\) – an integer \((r \geq 3)\); the rank of the spike
- \(t\) – boolean (default: True); whether the spike is tipped
- \(\text{groundset}\) – a string (optional); the groundset of the matroid

OUTPUT: matroid; the unique rank-\(r\) binary spike (tipped or tipless)

EXAMPLES:

```python
sage: matroids.Z(8)
Z_8: Binary matroid of rank 8 on 17 elements, type (7, 1)

sage: matroids.Z(9)
Z_9: Binary matroid of rank 9 on 19 elements, type (9, None)

sage: matroids.Z(20, False)
Z_20\_\text{t}: Binary matroid of rank 20 on 40 elements, type (20, 0)
```

2.3. Database of matroids
It holds that $Z_3 \setminus e \cong M(K_4)$, for all $e$:

```python
sage: import random
sage: Z3 = matroids.Z(3)
```

```python
sage: E = sorted(Z3.groundset()); E
['t', 'x1', 'x2', 'x3', 'y1', 'y2', 'y3']
```

```python
sage: e = random.choice(E)
```

```python
sage: Z3.delete(e).is_isomorphic(matroids.catalog.K4())
```

```
True
```

$Z_3 \cong F_7$:

```python
sage: F7 = matroids.catalog.Fano()
sage: Z3.is_isomorphic(F7)
```

```
True
```

$Z_4 \setminus t \cong AG(3, 2)$:

```python
sage: Z4 = matroids.Z(4, False)
```

```python
sage: Z4.is_isomorphic(matroids.catalog.AG32())
```

```
True
```

and $Z_4 \setminus e \cong S_8$, for all $e \neq t$:

```python
sage: Z4 = matroids.Z(4)
```

```python
sage: E = sorted(Z4.groundset())
```

```python
sage: e = random.choice(E)
```

```python
sage: S8 = matroids.catalog.S8()
```

(continues on next page)
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 % Integer(2) == Integer(1))  # XOR
True
sage: Z.automorphism_group().is_transitive()  # long time
True
```

REFERENCES:

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:

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

See also `sage.matroids.advanced`.

EXAMPLES:

```python
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
```

```python
>>> 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 $O(\binom{N}{R})$ 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

```python
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:
```
It is possible to provide either bases or nonbases:

```python
>>> 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:

```python
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M1 = BasisMatroid(matroids.Uniform(2, 5))
>>> M2 = BasisMatroid(groundset=range(5), rank=2)
>>> M1 == M2
True
```

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

```python
>>> from sage.matroids.advanced import *
>>> M1 = BasisMatroid(groundset='abcd', bases=['ab', 'cd'])
>>> M1.full_rank()
2
>>> 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:

```python
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
```

```python
>>> 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:

```python
sage: M = Matroid(bases=matroids.catalog.Fano().bases())
sage: M.bases_count()
28
```

```python
>>> from sage.all import *
>>> M = Matroid(bases=matroids.catalog.Fano().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:

```python
dsage: M = Matroid(bases=matroids.catalog.Pappus().bases())
sage: M.dual()
Matroid of rank 6 on 9 elements with 75 bases
```

```python
>>> 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:**

\( M\text{.extensions()}, M\text{.linear_subclasses()}, \text{sage.matroids.extension} \)

**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 \( \text{self.full\_rank()} \) that is not a basis.

**OUTPUT:** iterable containing the nonbases of the matroid

**See also:**

\( \text{Matroid.basis()} \)

**EXAMPLES:**

```
sage: M = Matroid(bases=matroids.catalog.Fano().bases())
sage: len(M.nonbases())
7
```

```
>>> from sage.all import *
>>> M = Matroid(bases=matroids.catalog.Fano().bases())
>>> len(M.nonbases())
7
```
**relabel**(mapping)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element \( e \) by \( \text{mapping}[e] \), where \( \text{mapping} \) is a given injective map. If \( \text{mapping}[e] \) is not defined, then the identity map is assumed.

**INPUT:**

- \( \text{mapping} \) – a Python object such that \( \text{mapping}[e] \) is the new label of \( e \)

**OUTPUT:** matroid

**EXAMPLES:**

```python
sage: from sage.matroids.advanced import BasisMatroid
sage: M = BasisMatroid(matroids.catalog.Fano())
```

```python
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```python
sage: N = M.relabel({'a': 0, 'g': 'x'})
```

```python
sage: from sage.matroids.utilities import cmp_elements_key
sage: sorted(N.groundset(), key=cmp_elements_key)
[0, 'b', 'c', 'd', 'e', 'f', 'x']
```

```python
sage: N.is_isomorphic(M)
True
```

```python
>>> from sage.all import *
>>> from sage.matroids.advanced import BasisMatroid
>>> M = BasisMatroid(matroids.catalog.Fano())
```

```python
>>> sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

```python
>>> N = M.relabel({'a': Integer(0), 'g': 'x'})
```

```python
>>> from sage.matroids.utilities import cmp_elements_key
>>> sorted(N.groundset(), key=cmp_elements_key)
[0, 'b', 'c', 'd', 'e', 'f', 'x']
```

```python
>>> 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:**

\( M.\text{extension()} \), \( M.\text{contract()} \)

**EXAMPLES:**

```python
sage: M = Matroid(bases=matroids.catalog.N2().bases())
```

```python
sage: M.truncation()
```

```
Matroid of rank 5 on 12 elements with 702 bases

sage: M.whitney_numbers2()
[1, 12, 66, 190, 258, 99, 1]
```

```python
sage: M.truncation().whitney_numbers2()
[1, 12, 66, 190, 258, 1]
```
3.2 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:

```python
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
```

AUTHORS:
- Giorgos Mousa (2023-12-23): initial version

```python
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:

```python
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)
(continues on next page)```
broken_circuit_complex \( \text{(ordering=None, reduced=False)} \)

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:**

```python
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], reduced=True)
Simplicial complex with vertex set (1, 2, 3) and facets {(1, 3), (1, 4), (2, 3), (2, 4)}
```
Simplicial complex with vertex set \((1, 2, 3, 4, 5)\) and facets \(\{(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)\}\)

```python
>>> 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:

```python
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
```

circuits \((k=None)\)

Return the circuits of the matroid.

**INPUT:**

- \(k\) – integer (optional); the length of the circuits

**OUTPUT:** SetSystem

**EXAMPLES:**

```python
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})]
```

```python
>>> 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({Integer(0), Integer(1), Integer(2)}),
 frozenset({Integer(0), Integer(1), Integer(3)}),
 frozenset({Integer(0), Integer(2), Integer(3)}),
 frozenset({Integer(1), Integer(2), Integer(3)})]
```

(continues on next page)
circuits_iterator (k=None)

Return an iterator over the circuits of the matroid.

INPUT:

- k – integer (optional); the length of the circuits

EXAMPLES:

```sage
circuits_iterator

circuits_iterator
```

dependent_sets (k)

Return the dependent sets of fixed size.

INPUT:

- k – integer

OUTPUT: SetSystem

EXAMPLES:

```sage
dependent_sets
```
```python
>>> 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:**

```python
sage: M = matroids.Theta(20)
sage: M.full_rank()
20
```

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

```python
>>> 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 $\infty$.

**EXAMPLES:**

```python
sage: matroids.Theta(10).girth()
3
```

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

```python
>>> 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:**

```python
sage: M = matroids.Theta(2)
sage: 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:**

```python
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
```

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:**

```python
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.catalog.Vamos())

sage: M.is_paving()
True
```

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

Test if self obeys the matroid axioms.

For a matroid defined by its circuits, we check the circuit axioms.

OUTPUT: boolean

EXAMPLES:

```python
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()
True
sage: C = [[1, 2, 3], [3, 4, 5]]
sage: M = Matroid(circuits=C)
sage: M.is_valid()
False
```

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

```
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])
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]]
```

```python
>>> 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:

```python
sage: M = Matroid(circuits=[[1, 2, 3], [3, 4, 5], [1, 2, 4, 5]])
```
```
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())
```
```
Simplicial complex with vertex set (1, 2, 3, 4, 5)
and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

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

```python
sage: M = Matroid(circuits=[[1, 2, 3], [1, 4, 5], [2, 3, 4, 5]])
```
```
Simplicial complex with vertex set (1, 2, 3, 4, 5)
and facets {(1, 3, 5), (2, 3, 5), (2, 4, 5), (3, 4, 5)}
```

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

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

```python
sage: M = Matroid(circuits=[[1, 2, 3], [1, 4, 5], [2, 3, 4, 5]])
```
```
Simplicial complex with vertex set (1, 2, 3, 4, 5)
and facets {(1, 3, 5), (2, 3, 5), (2, 4, 5), (3, 4, 5)}
```
```
SimplicialComplex(M.no_broken_circuits_sets([Integer(5), Integer(4)])
```
```
Simplicial complex with vertex set (1, 2, 3, 4, 5)
and facets {(1, 3, 5), (1, 4, 5), (2, 3, 5), (2, 4, 5)}
```

```python
>>> 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)]])
```
```
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 = matroids.Theta(5)
```
```
SetSystem of 15 sets over 10 elements
```
```
from sage.matroids.circuits_matroid import CircuitsMatroid
```
```
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.matroids.circuits_matroid import CircuitsMatroid
```
```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
```
```
sage: M = CircuitsMatroid(matroids.Uniform(Integer(2), Integer(4)))
```
```
sage: M.nonspanning_circuits()
```
```
SetSystem of 0 sets over 4 elements
```
```
sage: M = matroids.Theta(Integer(5))
```
```
sage: M.nonspanning_circuits()
```
```
SetSystem of 15 sets over 10 elements
```
```
3.2. Circuits matroids

EXAMPLES:

```python
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.matroids.circuits_matroid import CircuitsMatroid
```
```
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
```
```
sage: M = CircuitsMatroid(matroids.Uniform(Integer(2), Integer(4)))
```
```
sage: M.nonspanning_circuits()
```
```
SetSystem of 0 sets over 4 elements
```
```
sage: M = matroids.Theta(Integer(5))
```
```
sage: M.nonspanning_circuits()
```
```
SetSystem of 15 sets over 10 elements
```
```
3.2. Circuits matroids
nonspanning_circuits_iterator()

Return an iterator over the nonspanning circuits of the matroid.

EXAMPLES:

```python
sage: from sage.matroids.circuits_matroid import CircuitsMatroid
sage: M = CircuitsMatroid(matroids.Uniform(2, 4))
sage: list(M.nonspanning_circuits_iterator())
[]
```

```python
>>> 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:

```python
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
```

```python
>>> 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
```
3.3 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.

3.3.1 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:

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

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

See also sage.matroids.advanced.

EXAMPLES:

```python
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
```

Note that the class does not implement custom minor and dual operations:

```python
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
```
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
\[
(k, \{ \text{closure}(C) : C \text{ circuit of } M, \ r(C) = k \}) \text{ for } k = 0, \ldots, r(M) - 1
\]

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
\[
|X \cap \text{closure}(C)| \leq k \text{ for all circuits } C \text{ of } M \text{ with } r(C) = k.
\]

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 – dict (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:

```python
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'\}\}\}
```

```python
sage: M = CircuitClosuresMatroid(groundset='abcdefgh',
..:      circuit_closures={3: ['edfg', 'acdg', 'bcfg', 'cefh',
..:                          'afgh', 'abce', 'abdf', 'begh', 'bcdh', 'adeh']})
```
(continues on next page)
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', 'bgh', 'bcdh', 'adeh'],
...       Integer(4): ['abcdefgh']})
>>> 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):
...

3.3. Circuit closures matroids

229
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
```

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']
```

relabel(mapping)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element $e$ by $\text{mapping}[e]$, where $\text{mapping}$ is a given injective map. If $\text{mapping}[e]$ is not defined, then the identity map is assumed.

INPUT:

- mapping – a Python object such that $\text{mapping}[e]$ is the new label of $e$

OUTPUT: matroid

EXAMPLES:
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
```

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:
```python
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.Uniform(3, 4))
sage: sorted(M.flats(2), key=str)
[\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}]
```

flats_iterator \( (k) \)
Return an iterator over the flats of the matroid of specified rank.

INPUT:
• \( k \) – integer

EXAMPLES:
```python
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)], key=str)
[[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:
Matroid Theory, Release 10.4

```python
 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:**

```python
 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()**

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.

**OUTPUT:** boolean

**EXAMPLES:**

```python
 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'],
```

(continues on next page)
If we input a list or a lattice of flats, the method checks whether the lattice of flats is geometric:

```python
sage: M = Matroid(flats=[[0], [1], [0, 1]]).is_valid()
False
sage: M = Matroid(matroids.catalog.Fano().lattice_of_flats()).is_valid()
True
```

If we input a list or a lattice of flats, the method checks whether the lattice of flats is geometric:

```python
sage: M = Matroid(flats={{0}, [1], [0, 1]}).is_valid()
True
sage: M = Matroid(matroids.catalog.Fano().lattice_of_flats()).is_valid()
True
```
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',
... '012c', '89abc',
... '0123456789abc'])

Some invalid lists of flats are recognized before calling is_valid, upon the attempted matroid definition:

```python
sage: M = Matroid(flats=[[], [0], [1]]) # missing groundset
Traceback (most recent call last):
... ValueError: not a join-semilattice: no top element
```

```python
sage: Matroid(flats=[[0], [1], [0, 1]]) # missing an intersection
Traceback (most recent call last):
... ValueError: not a meet-semilattice: no bottom element
```

```python
sage: Matroid(flats=[[0], [0, 1], [2], [0], [1], [0, 1, 2]])
Traceback (most recent call last):
... ValueError: the poset is not ranked
```

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

```python
>>> from sage.matroids.flats_matroid import FlatsMatroid
```

```
lattice_of_flats()

Return the lattice of flats of the matroid.

EXAMPLES:
```
Matroid Theory, Release 10.4

(continued from previous page)

```python
>>> 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 \( \text{mapping}[e] \), where \( \text{mapping} \) is a given injective map. If \( \text{mapping}[e] \) is not defined, then the identity map is assumed.

**INPUT:**

- \( \text{mapping} \) – a Python object such that \( \text{mapping}[e] \) is the new label of \( e \)

**OUTPUT:** matroid

**EXAMPLES:**

```python
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
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
```

**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:**

```python
sage: from sage.matroids.flats_matroid import FlatsMatroid
sage: M = FlatsMatroid(matroids.catalog.BetsyRoss())
sage: 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:

```python
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]
```

3.5 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.

3.5.1 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:

```python
sage: 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 \( A = [IB] \):

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

(continues on next page)
3.5.2 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

### 3.5.3 Methods

```python
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.
```

3.5. Linear matroids
The simplest way to create a `BinaryMatroid` is by giving only a matrix $A$. Then, the groundset defaults to \texttt{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 $[I \ B]$ 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` – (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 sub-matrix 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
```
```python
>>> 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)

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 $\mathbb{F}_2$.

**EXAMPLES:**

```python
sage: M = matroids.catalog.Fano()
sage: 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:**

```python
sage: M = matroids.catalog.Fano()
sage: 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, Brown's invariant is not defined. Otherwise, \( B(V) = \sqrt{2}^k \exp(\sigma \pi i / 4) \) for some integers \( k, \sigma \). In that case, \( k \) equals the bycyle dimension of \( V \), and Brown's invariant for \( V \) is defined as \( \sigma \) 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
```

```
>>> from sage.all import *
>>> M = matroids.catalog.Fano()
>>> M.brown_invariant()
0
```

```
>>> 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]
])
>>> M.brown_invariant()
None
```

(continues on next page)
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
```
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.

\texttt{is\_valid()}

Test if the data obey the matroid axioms.

Since this is a linear matroid over the field $\mathbb{F}_2$, this is always the case.

OUTPUT:

True.

EXAMPLES:

\begin{verbatim}
sage: M = Matroid(Matrix(GF(2), [[1, 1]]))
sage: M.is_valid()
True
\end{verbatim}
relabel (mapping)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element $e$ by $\text{mapping}[e]$, where $\text{mapping}$ is a given injective map. If $\text{mapping}[e]$ is not defined, then the identity map is assumed.

INPUT:

- $\text{mapping}$ – a Python object such that $\text{mapping}[e]$ is the new label of $e$

OUTPUT: matroid

EXAMPLES:

```python
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']
```

```python
>>> 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 $\text{range}(A.\text{ncols})$. 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:

- $\text{matrix}$ – (default: None) a matrix whose column vectors represent the matroid.
- $\text{reduced_matrix}$ – (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 $\text{matrix}$ and $\text{reduced_matrix}$ should be provided.
- $\text{groundset}$ – (default: None) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of $\text{matrix}$ or the number of rows plus the number of columns of $\text{reduced_matrix}$.
- $\text{ring}$ – (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.
- $\text{keep_initial_representation}$ – (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.

3.5. Linear matroids 245
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 *
```

```python
>>> 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
```

```python
>>> 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:**

```python
sage: M = Matroid(matrix=Matrix(GF(5), [[1, 0, 1, 1], ....: [0, 1, 1, 2, 3]]))
sage: M.base_ring()
Finite Field of size 5
```

**characteristic()**

Return the characteristic of the base ring of the matrix representing the matroid.

**EXAMPLES:**

```python
sage: 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:**
cross_rations (hyperlines=None)

Return the set of cross ratios that occur in this linear matroid.

Consider the following matrix with columns labeled by \( \{a, b, c, d\} \).

\[
\begin{array}{cccc}
1 & 0 & 1 & 1 \\
0 & 1 & x & 1
\end{array}
\]

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:

```python
sage: M = Matroid(Matrix(GF(7), [[1, 0, 0, 1, 1],
....: [0, 1, 0, 1, 2],
....: [0, 0, 1, 3, 2]]))
sage: M.cross_ratio([0], 1, 2, 3)
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
```
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 $[I_1 \ A]$, then the dual is represented by $[-A^T \ I_2]$ for appropriately sized identity matrices $I_1, I_2$.

**OUTPUT:**

The dual matroid.

**EXAMPLES:**

```python
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
```

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

>>> A = Matrix(GF(Integer(7)), [[Integer(1), Integer(1), Integer(0),
Integer(1)],
[Integer(0), Integer(1), Integer(0), Integer(1)],
[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)

>>> M.cross_ratios() # needs sage.graphs
set()
```
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), \[
\begin{bmatrix}
1, & 0, & 1, & 1, \\
0, & 1, & 1, & 4,
\end{bmatrix}
\]))
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)), \[
\begin{bmatrix}
Integer(1), & Integer(0), & Integer(1), & Integer(1), \\
Integer(0), & Integer(1), & Integer(1), & Integer(4),
\end{bmatrix}
\]))
>>> 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 \(M.fundamental_cycle(B, e)\) to elements of the ring.

**See also:**

\(M.fundamental_cycle()\)

**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.

\( \text{minor}(\text{contractions}=F) \) has a \( U_{2,k} \) restriction or False, None.

**OUTPUT:**

Boolean or tuple.

**EXAMPLES:**

```python
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’}))
```

```python
>>> 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’}))
```

**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:

```python
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
```

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

```python
>>> 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(1), Integer(0), Integer(1),
                                    Integer(1),
                                    Integer(1)],
                                    [Integer(1), Integer(1), Integer(0),
                                    Integer(1),
                                    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
```
**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',
    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:

```python
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'
```

```python
def from sage.matroids.advanced import *
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
```

```python
def from sage.all import *
from sage.matroids.advanced 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'
```

```python
def from sage.matroids.advanced import *
M4 = BinaryMatroid(Matrix(M1))
M5 = LinearMatroid(reduced_matrix=Matrix(GF(Integer(2)), [[-Integer(1), -Integer(0), Integer(1)], [Integer(0), Integer(1), -Integer(1)], [Integer(1), -Integer(1), Integer(0)]]))
M4.is_field_isomorphic(M5)
True
```

```python
>>> M1 = Matroid(groundset=[0, 1, 2, 3], matrix=Matrix(GF(7), [[1, 0, 1, 1], [0, 1, 1, 2]]))
>>> M2 = Matroid(groundset=[0, 1, 2, 3], matrix=Matrix(GF(7), [[1, 0, 1, 1], [0, 1, 2, 1]]))
>>> M1.is_field_isomorphic(M2)
True
>>> M1.is_field_equivalent(M2)
False
```

(continues on next page)
is_field_isomorphic (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
```python
>>> 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), Integer(0), Integer(1)],
  [Integer(1), Integer(0), Integer(1)]])
>>> morphism = {'a':Integer(0), 'b':Integer(1), 'c': Integer(2), 'd':
... 'e':Integer(4), 'f':Integer(5)}
>>> 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(1), Integer(3):Integer(3)}
>>> mf2 = {Integer(0):Integer(0), Integer(1):Integer(1),
... Integer(2):Integer(2), Integer(3):Integer(3)}
>>> M1.is_field_isomorphism(M2, mf1)
False
>>> M1.is_field_isomorphism(M2, mf2)
True
```

**is_valid()**

Test if the data represent an actual matroid.

Since this matroid is linear, we test the representation matrix.

**OUTPUT:**

- **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:**
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 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 0]
[0 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)
sage: M1 == M2.dual()
True

3.5. Linear matroids
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 \(N = M([A_0; -c1]),\) 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:**

- \(\text{F} = \) (default: self.groundset()) a subset of the groundset.
- \(\text{cosimple} = \) (default: False) boolean
- \(\text{fundamentals} = \) (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 \(\text{F}\), if given
- so that the cochain does not generate a series extension or coloop, if \(\text{cosimple} = \text{True}\)
- so that in the coextension generated by this cochain, the cross ratios are restricted to \(\text{fundamentals}\), if given.

**See also:**

\(M\.\text{linear_coextension()}\), \(M\.\text{linear_coextensions()}\), \(M\.\text{cross_ratios()}\)

**EXAMPLES:**

```python
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}]
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}}]
```

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

>>> M = Matroid(reduced_matrix=Matrix(GF(Integer(2)),
[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
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))
```
linear_coextensions \( (element=\text{None}, F=\text{None}, \text{cosimple}=\text{False}, \text{fundamentals}=\text{None}) \)

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 ground set.
- **F** – (default: None) a subset of the ground set.
- **cosimple** – (default: False) boolean
- **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 \( \text{cosimple} = \text{True} \).
- so that in the representation of the coextension, the cross ratios are restricted to \( \text{fundamentals} \), if given. Note that it is assumed that the cross ratios of the input matroid already satisfy this condition.

**See also:**

\( \text{M.linear_coextension()}, \text{M.linear_coextension_cochains()}, \text{M.cross_ratios()} \)

**EXAMPLES:**

```sage
sage: M = Matroid(ring=GF(2),
             reduced_matrix=[[1, 0, 1], [1, 1, 0], [0, 1, 1]])
```

```sage
sage: S = M.linear_coextensions(cosimple=True)
```

```sage
sage: S[0].is_field_isomorphic(F7.dual())
True
```

```sage
sage: M = Matroid(ring=QQ,
             reduced_matrix=[[1, 0, 1], [1, 1, 0], [0, 1, 1]])
```

```sage
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(cosimpie=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()

>>> 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]]
```

```
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(1), Integer(0)],
...
Integer(1), Integer(0), Integer(1), Integer(0),
...
Integer(0), Integer(1), Integer(0), Integer(1)],
...
Integer(1), 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]]
```

`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` – (default: `False`) boolean
- `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 \( \text{simple} = \text{True} \)
- so that in the extension generated by this chain, the cross ratios are restricted to \( \text{fundamentals} \), if given.

See also:

\[ M.\text{linear_extension}(), M.\text{linear_extensions}(), M.\text{cross_ratios}() \]

**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
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, 0, -1], [1, -1, 0], [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
```

(continues on next page)
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 – (default: False) boolean
- 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:

```python
sage: M = Matroid(ring=GF(2),
....:     reduced_matrix=[[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: len(M.linear_extensions())
8
sage: M = M.linear_extensions(simple=True)
[Binary matroid of rank 3 on 7 elements, type (3, 0)]
```

```python
sage: S = M.linear_extension_chains(); S
...:     [0].is_field_isomorphic(matroids.catalog.Fano())
...:     True
```

```python
sage: M = Matroid(ring=GF(2),
....:     reduced_matrix=[[1, 0, 0], [0, 1, 0], [0, 0, 1]])
```

```python
>>> all(D in L for D in result)
True
```
orlik_terao_algebra \((R=None, \text{ordering}=None, **\text{kwargs})\)

Return the Orlik-Terao algebra of \(\text{self}\).

**INPUT:**

- \(R\) – (default: the base ring of \(\text{self}\)) the base ring
- \(\text{ordering}\) – (optional) an ordering of the groundset

**See also:**

OrlikTeraoAlgebra

**EXAMPLES:**

```sage
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: OTG1 = M.orlik_terao_algebra(QQ, invariant=(G, action))
```

(continues on next page)
sage: OTG2 = M.orlik_terao_algebra(QQ, invariant=(action, G))
sage: OTG1 is OTG2
True

relabel(mapping)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element $e$ by \text{mapping}[e], where \text{mapping} is a given injective
map. If \text{mapping}[e] is not defined, then the identity map is assumed.

INPUT:

- \text{mapping} – a Python object such that \text{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 Wheel(Integer(3))
% 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
% 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
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 – (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 \( \text{groundset_list()} \) 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 \( \text{reduced} == False \) and \( \text{order} == None \) then this method will always output the same matrix (except when \( M._\text{forget()} \) 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 \( \text{order} \) 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
sage: M = matroids.catalog.Fano()
sage: M.representation()
[1 0 0 0 1 1 1]
```

(continues on next page)
sage: Matrix(M) == M.representation()
True
sage: M.representation(labels=True)
([1 0 0 1 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 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 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: 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))

from sage.matroids.advanced import lift_cross_ratios, lift_map, ...

3.5. Linear matroids 269
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 1 0 1 1 |
1 | 0 1 0 1 0 1 1 |
2 | 0 0 1 1 0 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}
   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 1 1 0 0 |
1 | 0 1 1 0 1 0 |
2 | 1 1 0 0 0 1 |
```

(continues on next page)
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\(\begin{matrix}0 & 1 & 1 & 1\end{matrix}\)
b\(\begin{matrix}1 & 0 & 1 & 1\end{matrix}\)
c\(\begin{matrix}1 & 1 & 0 & 1\end{matrix}\)

\[
\begin{align*}
\ggg & \text{from sage.all import } * \\
\ggg & M = matroids.catalog.Fano() \\
\ggg & A = M.representation(order=True); A \\
& \text{Generic morphism:} \\
& \text{From: Free module generated by } \{'a', 'b', 'c', 'd', 'e', 'f', 'g'\} \\
& \text{over Finite Field of size 2} \\
& \text{To: Free module generated by } \{0, 1, 2\} \text{ over Finite Field of size 2} \\
\ggg & \text{print}(A._unicode_art_matrix()) \\
\ggg & a\ b\ c\ d\ e\ f\ g \\
\ggg & 0\{1\ 0\ 0\ 1\ 1\ 1\} \\
\ggg & 1\{0\ 1\ 0\ 1\ 1\ 0\} \\
\ggg & 2\{0\ 0\ 1\ 1\ 0\ 1\} \\
\ggg & A = M.representation(B='efg', order=True); A \\
& \text{Generic morphism:} \\
& \text{From: Free module generated by } \{'a', 'b', 'c', 'd', 'e', 'f', 'g'\} \\
& \text{over Finite Field of size 2} \\
& \text{To: Free module generated by } \{0, 1, 2\} \text{ over Finite Field of size 2} \\
\ggg & \text{print}(A._unicode_art_matrix()) \\
\ggg & a\ b\ c\ d\ e\ f\ g \\
\ggg & 0\{1\ 1\ 0\ 1\ 0\ 0\} \\
\ggg & 1\{1\ 0\ 1\ 0\ 1\ 0\} \\
\ggg & 2\{1\ 1\ 0\ 0\ 0\ 1\} \\
\ggg & A = M.representation(B='abc', order=True, reduced=True); A \\
& \text{Generic morphism:} \\
& \text{From: Free module generated by } \{'d', 'e', 'f', 'g'\} \\
& \text{over Finite Field of size 2} \\
& \text{To: Free module generated by } \{'a', 'b', 'c'\} \text{ over Finite Field of size 2} \\
\ggg & \text{print}(A._unicode_art_matrix()) \\
\ggg & d\ e\ f\ g \\
\ggg & a\{0\ 1\ 1\ 1\} \\
\ggg & b\{1\ 0\ 1\ 1\} \\
\ggg & c\{1\ 1\ 0\ 1\}
\]

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)]
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 description.

The simplest way to create a QuaternaryMatroid is by giving only a matrix $A$. Then, the groundset defaults to $\text{range}(A.\text{ncols})$. 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 $\mathbb{F}_4$.
- **keep_initial_representation** – (default: True) boolean. 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 *
```
Matroid Theory, Release 10.4

```
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), 'y')
>>> 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), 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']
>>> 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 \( \mathbb{F}_4 \).

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
```

(continues on next page)
#### bicycle_dimension()

Return the bicycle dimension of the quaternary matroid.

The bicycle dimension of a linear subspace \( V \) is \( \dim(\mathcal{V} \cap \mathcal{V}^\perp) \). We use the inner product \( \langle v, w \rangle = v_1w_1^* + \ldots + v_nw_n^* \), 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:**

```python
sage: M = matroids.catalyst.Q10()
    # needs sage.rings.finite_rings
sage: M.bicycle_dimension()
    # 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:**

```python
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
```

(continues on next page)
needs sage.rings.finite_rings
2

is_valid()
Test if the data obey the matroid axioms.
Since this is a linear matroid over the field $\mathbb{F}_4$, this is always the case.

OUTPUT:
True.

EXAMPLES:
sage: M = Matroid(Matrix(GF(4, 'x'), [[]))
# needs sage.rings.finite_rings
sage: M.is_valid()  # needs sage.rings.finite_rings
True

sage: M = Matroid(Matrix(GF(Integer(4), 'x'), [[]))
# needs sage.rings.finite_rings
sage: 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 $\text{mapping}[e]$, where $\text{mapping}$ is a given injective map. If $\text{mapping}[e]$ is not defined, then the identity map is assumed.

INPUT:
• mapping – a Python object such that $\text{mapping}[e]$ 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

3.5. Linear matroids
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 $\text{range}(A.n_cols())$. 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` -- (default: True) boolean. 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
def from sage.matroids.advanced import *
```  
```sage>>> from sage.all import *
```
```sage>>> 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 #...
```

Regular matroid of rank 2 on 4 elements with 5 bases

(continues on next page)
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']

base_ring()

Return the base ring of the matrix representing the matroid, in this case \( \mathbb{Z} \).

EXAMPLES:

sage: M = matroids.catalog.R10()
sage: M.base_ring()
Integer Ring

>> 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]
>>> M = Matroid(matrix=A, groundset='abcd', regular=True)  # needs sage.graphs
>>> sorted(M.groundset())  # needs sage.graphs
['a', 'b', 'c', 'd']

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

3.5. Linear matroids
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)
```

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
```

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:**

```python
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(3, hyperlines=[[a', 'b', 'c'], 
                                   ['a', 'b', 'd']])
(True, frozenset({'a', 'b', 'c'}))
```

```python
>>> from sage.all import *
>>> M = matroids.catalog.R10()
>>> 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, 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()
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:**

```python
sage: M = matroids.catalog.R10()
sage: M.is_regular()
True
```

```python
>>> 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:**

```python
sage: N = matroids.catalog.R10()
sage: N.is_ternary()
True
```

```python
>>> from sage.all import *
>>> N = matroids.catalog.R10()
>>> N.is_ternary()
True
```

**is_valid()**

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\}.

**OUTPUT:** boolean

**EXAMPLES:**

```python
sage: # needs sage.graphs
sage: M = Matroid(Matrix(ZZ, [[1, 0, 0, 1, 1, 0, 1],
....:                         [0, 1, 0, 1, 0, 1, 1]],
```

(continues on next page)
relabel (mapping)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element $e$ by $\text{mapping}[e]$, where $\text{mapping}$ is a given injective map. If $\text{mapping}[e]$ is not defined, then the identity map is assumed.

**INPUT:**

- $\text{mapping}$ – a Python object such that $\text{mapping}[e]$ is the new label of $e$

**OUTPUT:** matroid

**EXAMPLES:**

```python
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']
```

ternary_matroid (randomized_tests=1, verify=True)

Return a ternary matroid representing $\text{self}$.

**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+
```

```python
>>> 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 \( \text{range}(A.ncols()) \). 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 – (default: True) boolean. 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),...
...
>>> 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),...
...
>>> 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 $\mathbb{F}_3$.

**EXAMPLES:**
```python
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:**

```python
sage: M = matroids.catalog.NonFano()
sage: M.bicycle_dimension()
0
```

```python
>>> 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:**

```python
sage: M = matroids.catalog.NonFano()
sage: M.character()
2
```

```python
>>> 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:**

```python
sage: M = matroids.catalog.NonFano()
sage: M.characteristic()
3
```
```python
>>> 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:**

```python
sage: N = matroids.catalog.NonFano()
sage: N.is_ternary()
True
```

```python
>>> from sage.all import *
>>> N = matroids.catalog.NonFano()
>>> N.is_ternary()
True
```

**is_valid()**

Test if the data obey the matroid axioms.

Since this is a linear matroid over the field \(\mathbb{F}_3\), this is always the case.

**OUTPUT:**

True.

**EXAMPLES:**

```python
sage: M = Matroid(Matrix(GF(3), [[]]))
sage: M.is_valid()
True
```

```python
>>> 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:

```python
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']
```

```python
>>> 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=I, 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:**

```python
sage: N = matroids.catalog.NonFano()
sage: N.ternary_matroid() is N
True
```

```python
>>> from sage.all import *
>>> N = matroids.catalog.NonFano()
>>> N.ternary_matroid() is N
True
```
3.6 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.

3.6.1 Constructions

Any function can be used, but no checks are performed, so be careful.

EXAMPLES:

```python
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
```

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:

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

>>> from sage.all import *

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

288 Chapter 3. Concrete implementations
AUTHORS:

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

### 3.6.2 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:**

```python
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
```

```python
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:**

```python
from sage.matroids.advanced import *
M = RankMatroid(range(6),
    ....:     rank_function=matroids.Uniform(3, 6).rank)
sage: sorted(M.groundset())
[0, 1, 2, 3, 4, 5]
```

```python
from sage.all import *
from sage.matroids.advanced import *
M = RankMatroid(range(Integer(6)),
    ...     rank_function=matroids.Uniform(Integer(3), Integer(6)).rank)
(continues on next page)```
3.7 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 $C$.

3.7.1 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()
```

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)
```
```
True
```
True

```python
sage: isinstance(M1, RegularMatroid)
False
```

```python
>>> 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

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:

```python
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
```

AUTHORS:

- Zachary Gershkoff (2017-07-07): initial version

```python
class sage.matroids.graphic_matroid.GraphicMatroid
    Bases: Matroid
```

3.7. Graphic Matroids
The graphic matroid class.

**INPUT:**

- $G$ - Graph
- $\text{groundset} -$ list (optional); in 1-1 correspondence with $G.$edge\_iterator() 

**OUTPUT:** $\text{GraphicMatroid}$ where the groundset elements are the edges of $G$

**Note:** If a disconnected graph is given as input, the instance of $\text{GraphicMatroid}$ will connect the graph components and store this as its graph.

**EXAMPLES:**

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

A disconnected input will get converted to a connected graph internally:

```python
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
```

```python
>>> from sage.all import *
```
You can still locate an edge using the vertices of the input graph:

```python
sage: G1 = graphs.CycleGraph(3); G2 = graphs.DiamondGraph()
>>> G1.disjoint_union(G2)
sage: M = Matroid(G)
sage: H = M.graph()
sage: 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:**

```python
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')]
```

```python
sage: M = Matroid(graphs.CompleteGraph(5))
sage: 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
- \( \text{element} \) – (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 \( X \).
**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 (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')],
 [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)
[[[Integer(0), Integer(1), (Integer(0), Integer(2), (Integer(0), Integer(3),
  (Integer(0), Integer(4), (Integer(0), Integer(5), 'a')],
 [Integer(0), Integer(1), (Integer(0), Integer(2), (Integer(0), Integer(3),
  (Integer(0), Integer(4), (Integer(0), Integer(5), 'a')],
 [Integer(0), Integer(1), (Integer(0), Integer(2), (Integer(0), Integer(3),
  (Integer(0), Integer(4), (Integer(0), Integer(5), 'a')],
 [Integer(0), Integer(1), (Integer(0), Integer(2), (Integer(0), Integer(3),
  (Integer(0), Integer(5), 'a')],
 [Integer(0), Integer(1), (Integer(0), Integer(3), (Integer(0), Integer(4),
  (Integer(0), Integer(5), 'a')],
 [Integer(0), Integer(1), (Integer(0), Integer(3), (Integer(0), Integer(5),
  'a')],
 [Integer(0), Integer(2), (Integer(0), Integer(3), (Integer(0), Integer(4),
  (Integer(0), Integer(5), 'a')],
 [Integer(0), Integer(2), (Integer(0), Integer(3), (Integer(0), Integer(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')

...  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']

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 – (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.
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, 3), (2, 3, 'a'), (2, 3, 4)]

M = Matroid(graphs.CompleteBipartiteGraph(3, 3))
I = M.graphic_extensions(simple=True)
sum (1 for i in I)
6

M = Matroid(graphs.CompleteBipartiteGraph(3, 3))
I = M.graphic_extensions(vertices=[0, 1, 2])
sum (1 for i in I)
4

groundset()

Return the groundset of the matroid as a frozenset.

EXAMPLES:

M = Matroid(graphs.DiamondGraph())
sorted(M.groundset())
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]

G = graphs.CompleteGraph(3).disjoint_union(graphs.CompleteGraph(4))
M = Matroid(range(G.num_edges()), G); sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6, 7, 8]

M = Matroid(Graph([(0, 1, 'a'), (0, 2, 'b'), (0, 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:
```python
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
```
is_valid()
Test if the data obey the matroid axioms.
Since a graph is used for the data, this is always the case.
OUTPUT: True
EXAMPLES:

```python
sage: M = matroids.CompleteGraphic(4); M
M(K4): Graphic matroid of rank 3 on 6 elements
sage: M.is_valid()
True
```

```python
>>> 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:

```python
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),
(continues on next page)```
(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),
(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:

```python
sage: M = matroids.CompleteGraphic(5); M
M(K5): Graphic matroid of rank 4 on 10 elements
sage: N = M.regular_matroid(); N
Regular matroid of rank 4 on 10 elements with 125 bases
sage: M.equals(N)
True
sage: M == N
False
```

```python
>>> 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:**

```python
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
```

```python
>>> 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
Matroid Theory, Release 10.4

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
```

```
>>> 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)])
Traceback (most recent call last):
```

(continues on next page)
... 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}
```
CHAPTER
FOUR

ABSTRACT MATROID CLASSES

4.1 Dual matroids

4.1.1 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
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
```

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

4.1.2 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
sage: from sage.matroids.advanced import *
```

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

See also `sage.matroids.advanced`. 
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
from sage.all import *
M = matroids.catalog.Pappus().dual()
N = M.dual()
N.rank()
3
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']

is_valid()

Test if self obeys the matroid axioms.

For a DualMatroid, we check its dual.

OUTPUT: boolean

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

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.

4.1. Dual matroids
INPUT:

- mapping – a Python object such that mapping[e] is the new label of e

OUTPUT: matroid

EXAMPLES:

```python
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
```

```python
>>> 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
```

4.2 Minors of matroids

4.2.1 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 $M \setminus D$ (this also works in Sage). Contraction is the dual operation: $M / C == (M.dual() \setminus C).dual()$. 

EXAMPLES:

```python
sage: M = matroids.catalog.Fano()
sage: M.delete(['a', 'c']) == M.delete(['a', 'c'])
True
sage: M / 'a' == M.contract('a')
True
sage: (M / 'c').delete('ab') == M.minor(contractions='c', deletions='ab')
True
```

```python
>>> from sage.all import *
>>>
>>> M = matroids.catalog.Fano()
>>> M.delete(['a', 'c']) == M.delete(['a', 'c'])
True
>>> M / 'a' == M.contract('a')
```

(continues on next page)
True
>>> (M / 'c').delete('ab') == M.minor(contractions='c', deletions='ab')
True

If a contraction set is not independent (or a deletion set not co-indepent), this is taken care of:

```python
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
```

See also:

\[ M.delete(), M.contract(), M.minor, \]

### 4.2.2 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:

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

See also `sage.matroids.advanced`.

AUTHORS:

- Rudi Pendavingh, Michael Welsh, Stefan van Zwam (2013-04-01): initial version
4.2.3 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
{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:

```python
sage: M = matroids.catalog.Pappus().contract(['c'])
sage: sorted(M.groundset())
['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i']
```

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

```python
>>> M = matroids.catalog.Pappus().contract(['c'])
```

```python
>>> sorted(M.groundset())
['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i']
```

4.3 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

```python
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. 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
sage: M = matroids.catalog.N1()
sage: M.bases_count()
184
```

```python
>>> 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']

`circuits` *(k=None)*

Return the circuits of the matroid.

OUTPUT: iterable containing all circuits

See also:

*M.circuit()*

EXAMPLES:

```python
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]]
```

```python
>>> 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:

```python
>>> 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]]
```
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
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
```

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:

\texttt{M.is\_connected()}, \texttt{M.delete()}

EXAMPLES:

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

\texttt{dependent\_sets}(k)

Return the dependent sets of fixed size.

INPUT:

\begin{itemize}
  \item k – integer
\end{itemize}

OUTPUT: iterable containing all dependent sets of size \( k \)

EXAMPLES:

```python
>>> M = matroids.catalog.N1()
>>> len(M.nonbases())
68
>>> [len(M.dependent_sets(k)) for k in range(M.full_rank() + 1)]
[0, 0, 0, 0, 9, 68]
```

\texttt{flats}(k)

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

A flat is a closed set.

INPUT:

\begin{itemize}
  \item k – integer
\end{itemize}

OUTPUT: \texttt{SetSystem}

See also:

\texttt{Matroid.closure()}

4.3. Basis exchange matroids
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
```

```plaintext
>>> 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
```

```plaintext
>>> 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

**EXAMPLES:**

```python
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]
```

**is_valid**

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 \(B\).

**OUTPUT:** boolean

**EXAMPLES:**

```python
>>> 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]
```
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()
False

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

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
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]
```
5.1 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

### 5.2 Some useful functions for the matroid class.

For direct access to the methods `newlabel()`, `setprint()` and `get_nonisomorphic_matroids()`, type:

```python
sage: 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:**

```python
sage: from sage.matroids.utilities import cmp_elements_key
sage: 1 = ['a', 'b', 1, 3, 2, 10, 111, 100, 'c', 'aa']

sage: sorted(1, key=cmp_elements_key)
[1, 2, 3, 10, 100, 111, 'a', 'aa', 'b', 'c']
```

```python
>>> from sage.all import *
>>> from sage.matroids.utilities import cmp_elements_key

>>> from sage.matroids.utilities import cmp_elements_key
>>> 1 = ['a', 'b', Integer(1), Integer(3), Integer(2), Integer(10), Integer(111),
       Integer(100), 'c', 'aa']

>>> sorted(1, 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:

```python
sage: 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 → 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 ∈ S, 1 ∈ T;
  • for all x ∈ S: f(x) = 1 if and only if x = 1;
  • for all x, y ∈ S: if x + y = 0 then f(x) + f(y) = 0;
  • for all x, y ∈ S: if x + y = 1 then f(x) + f(y) = 1;
  • for all x, y, z ∈ S: if xy = z then f(x)f(y) = f(z).

5.2. Some useful functions for the matroid class.
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 $\text{lift_map()}$.

See also:

$\text{lift_map()}$

EXAMPLES:

```python
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: Z = lift_cross_ratios(A, to_sixth_root_of_unity)  # needs sage.rings.finite_rings sage.rings.number_field
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())
[z, -z + 1]
sage: M.is_isomorphism(N, {e:e for e in M.groundset()})  # needs sage.rings.finite_rings sage.rings.number_field
True
```

(continues on next page)
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 – a string describing the target (partial) field.

OUTPUT:

- a dictionary

Depending on the value of target, the following lift maps will be created:

- “reg”: a lift map from $\mathbb{F}_3$ to the regular partial field $(\mathbb{Z}, < -1 >)$.
- “sru”: a lift map from $\mathbb{F}_7$ to the sixth-root-of-unity partial field $(\mathbb{Q}(z), < z >)$, where $z$ is a sixth root of unity. The map sends 3 to $z$.
- “dyadic”: a lift map from $\mathbb{F}_{11}$ to the dyadic partial field $(\mathbb{Q}, < -1, 2 >)$.
- “gm”: a lift map from $\mathbb{F}_{19}$ to the golden mean partial field $(\mathbb{Q}(t), < -1, t >)$, 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:

```python
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:
    - 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')
```
sage.matroids.utilities.make_regular_matroid_from_matroid(matroid)

Attempt to construct a regular representation of a matroid.

INPUT:

• matroid – a 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(matroids.CompleteGraphic(6)).is_isomorphic(matroids.CompleteGraphic(6))
True

sage.matroids.utilities.newlabel(groundset)

Create a new element label different from the labels in groundset.

INPUT:

• groundset – A set of objects.

OUTPUT:

A string not in the set groundset.

For direct access to newlabel, run:

```python
>>> from sage.all import *
>>> from sage.matroids.utilities import make_regular_matroid
>>> make_regular_matroid_from_matroid(matroids.CompleteGraphic(Integer(6))).is_isomorphic(matroids.CompleteGraphic(Integer(6)))
True
```
sage: from sage.matroids.advanced import *

>>> from sage.all 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:

```python
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'
```

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

>>> from sage.matroids.advanced import newlabel

>>> S = set(['a', Integer(42), 'b'])

>>> newlabel(S)
in S
False
```
EXAMPLES:

```python
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.
```

```python
>>> 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, [1, 2, 3], 'efg'))
Traceback (most recent call last):
  ... ValueError: [1, 2, 3] is not a subset of the groundset

>>> 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

>>> 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:

```python
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:

```python
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}]
```

Note that for iterables, the effect can be undesirable:

```python
>>> 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}]
```

An exception was made for subclasses of `SageObject`:

```python
sage: G = graphs.PetersenGraph()  # needs sage.graphs
sage: list(G)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: setprint(G)  # needs sage.graphs
(continues on next page)
```

5.2. Some useful functions for the matroid class. 329
needs sage.graphs
Petersen graph: Graph on 10 vertices

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

G = graphs.PetersenGraph()
list(G)
setprint(G)
```

sage.matroids.utilities.setprint_s(X, toplevel=False)

Create the string for use by setprint().

INPUT:

- X – any Python object
- toplevel – (default: False) indicates whether this is a recursion or not.

OUTPUT:

A string representation of the object, with nice notation for sets and frozen sets.

EXAMPLES:

```python
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}]'
```

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

>>> 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:

```python
sage: X = 'abcd'
sage: setprint_s(X)
'"abcd"'
sage: setprint_s(X, toplevel=True)
'abcd'
```

```python
>>> 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 non-zero, 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:

```python
sage: from sage.matroids.utilities import spanning_forest
sage: len(spanning_forest(matrix([[1,1,1],[1,1,1],[1,1,1]])))
5

sage: len(spanning_forest(matrix([[0,0,1],[0,1,0],[0,1,0]])))
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 non-zero, 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:

```python
sage: from sage.matroids.utilities import spanning_stars
sage: edges = spanning_stars(matrix([[1,1,1],[1,1,1],[1,1,1]]))
sage: Graph([(x+3, y) for x,y in edges]).is_connected()
True
```

(continues on next page)
sage.matroids.utilities.split_vertex(G, u=None, 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) An 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:

```python
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)])
```

5.3 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

5.3.1 Methods

```python
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.
```

```python
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:
    - \( M \) – a matroid.
    - `line_length` – (default: None) an integer.
    - `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:
Note that this class is intended for runtime, internal use, so no loads/dumps mechanism was implemented.

```python
>>> 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.
6.1 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

```python
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:
```
```
C = BinaryMatrix(Integer(2), Integer(2))

A == B and A == C

True

base_ring()

Return $GF(2)$.

EXAMPLES:

```python
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(4, 4)
sage: A.base_ring()
Finite Field of size 2
```

characteristic()

Return the characteristic of self.base_ring().

EXAMPLES:

```python
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(3, 4)
sage: 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.
```python
sage: M = Matroid(ring=GF(5), matrix=[[1, 0, 1, 1], [0, 1, 1, 2, 3]])
# indirect doctest
sage: M.is_isomorphic(matroids.Uniform(2, 5))
True
```

### base_ring()

Return the base ring of `self`.

**EXAMPLES:**

```python
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
```

```python
>>> 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:**

```python
sage: from sage.matroids.lean_matrix import GenericMatrix
sage: A = GenericMatrix(3, 4, ring=GF(5))
sage: A.characteristic()
5
```

```python
>>> 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:

```python
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
```

```python
>>> 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:

```python
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.
```

```python
>>> 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:

```python
sage: from sage.matroids.lean_matrix import GenericMatrix
sage: A = GenericMatrix(3, 4, ring=GF(5))
sage: A.characteristic()
5
```

```python
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import GenericMatrix
>>> A = GenericMatrix(Integer(3), Integer(4), ring=GF(Integer(5)))
>>> A.characteristic()
5
```
Maths Theory, Release 10.4

**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)
```

(continues on next page)
sage: M.is_isomorphic(matroids.Wheel(3))
True

from sage.all import *
M = Matroid(graphs.CompleteGraph(Integer(4)).incidence_matrix(oriented=True),
regular=True)
M.is_isomorphic(matroids.Wheel(Integer(3)))
True

**base_ring()**

Return the base ring of self.

**EXAMPLES:**

```python
sage: from sage.matroids.lean_matrix import PlusMinusOneMatrix
sage: A = PlusMinusOneMatrix(3, 4)
sage: A.base_ring()
Integer Ring
```

```python
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:**

```python
sage: from sage.matroids.lean_matrix import PlusMinusOneMatrix
sage: A = PlusMinusOneMatrix(3, 4)
sage: A.characteristic()
0
```

```python
from sage.all import *
from sage.matroids.lean_matrix import PlusMinusOneMatrix
A = PlusMinusOneMatrix(Integer(3), Integer(4))
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) A 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:

```python
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

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

>>> from sage.matroids.lean_matrix import *

>>> A = QuaternaryMatrix(Integer(2), Integer(2), Matrix(GF(Integer(4), x),

˓→[[0, 0], [0, 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:

```python
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"""

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

>>> from sage.matroids.lean_matrix import *

>>> A = QuaternaryMatrix(Integer(2), Integer(2), ring=GF(Integer(4), 'f'))

```

"""Finite Field in f of size 2^2"""

**characteristic()**

Return the characteristic of self.base_ring().

EXAMPLES:

```python
sage: from sage.matroids.lean_matrix import *
sage: A = QuaternaryMatrix(200, 5000, ring=GF(4, 'x'))

sage: A.characteristic()
```

2

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

>>> from sage.matroids.lean_matrix import *

>>> A = QuaternaryMatrix(Integer(200), Integer(5000), ring=GF(Integer(4), 'x'))

```

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 *
>>> from sage.matroids.lean_matrix import RationalMatrix
>>> A = RationalMatrix(Integer(3), Integer(4))
>>> A.base_ring()
Rational Field
```

```
>>> 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 \( n \times n \) 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]
```

6.2 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 `createlinet` 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 line orders
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>}
```
6.2.1 REFERENCES


EXAMPLES:

```python
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, 1, 0, 0, 1, 0],
                     ....:  [0, 0, 1, 0, 1, 1, 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)}
```

```python
sage: M1._cached_info = {plot_positions: pos_dict, plot_lineorders: None}
```

```python
sage: matroids_plot_helpers.geomrep(M1, sp=True)
```

```
needs sage.plot sage.rings.finite_rings
```

```
Graphics object consisting of 22 graphics primitives
```

```python
>>> from sage.all import *
>>> from sage.matroids import matroids_plot_helpers
>>> M1 = Matroid(ring=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(0),Integer(1),Integer(1),Integer(0),Integer(0),Integer(1)],
                     ....:  [Integer(1), Integer(0), Integer(0),Integer(1),
                     ....:  Integer(1),Integer(1),Integer(0),Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)])
```

```python
>>> pos_dict = {Integer(0): (Integer(0), Integer(0)), Integer(1): (Integer(2),
                     ....:  Integer(1)), Integer(2): (Integer(1), Integer(2)), Integer(3): (RealNumber('1.5'),
                     ....:  RealNumber('1.0'))},
```

```python
>>> M1._cached_info = {plot_positions: pos_dict, plot_lineorders: None}
```

```python
>>> 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** – A 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** – A 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, 1, 1],
   ....: [0, 1, 0, 1, 0, 1, 1, 1, 0],
   ....: [0, 0, 1, 1, 1, 0, 1, 1, 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
```

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 – A list of 3 ground set elements that are to be placed on vertices of the triangle.
- ptsdict – A dictionary (at least) containing ground set elements in tripts as keys and their (x,y) position as values.
- nontripts – A 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
```
\begin{verbatim}
sage: ptsdict={\texttt{a}: (0,0), \texttt{b}: (1,2), \texttt{c}: (2,0)}
sage: ptsdict_1=matroids_plot_helpers.addnontripts(\{'a','b','c'\},
\quad ....:
\quad [\texttt{d}', \texttt{e}', \texttt{f}'], ptsdict)
sage: setprint(ptsdict_1)
\{\texttt{a}: \{0, 0\}, \texttt{b}: \{1, 2\}, \texttt{c}: \{0, 2\}, \texttt{d}: \{0.6666666666666666, 1.0\},
\texttt{e}: \{0.6666666666666666, 0.8888888888888888\},
\texttt{f}: \{0.8888888888888888, 1.3333333333333333\}\}
sage: ptsdict_2=matroids_plot_helpers.addnontripts(\{'a','b','c'\},
\quad ....:
\quad [\texttt{d}', \texttt{e}', \texttt{f}', \texttt{g}', \texttt{h}'], ptsdict)
sage: setprint(ptsdict_2)
\{\texttt{a}: \{0, 0\}, \texttt{b}: \{1, 2\}, \texttt{c}: \{0, 2\}, \texttt{d}: \{0.6666666666666666, 1.0\},
\texttt{e}: \{0.6666666666666666, 0.8888888888888888\},
\texttt{f}: \{0.8888888888888888, 1.3333333333333333\},
\texttt{g}: \{0.2222222222222222, 1.0\},
\texttt{h}: \{0.5185185185185185, 0.5555555555555555\}\}
\end{verbatim}

Note: This method does NOT do any checks.

\texttt{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.

\textbf{INPUT:}

- \texttt{ptsdict} – A dictionary containing keys and their (x,y) position as values.
- \texttt{ll} – A list of keys in \texttt{ptsdict} through which a line is to be drawn.
- \texttt{lineorders2} – (optional) A list of ordered lists of keys in \texttt{ptsdict} such that if \texttt{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.

\textbf{OUTPUT:}

A tuple containing 4 elements in this order:

1. Ordered list of x-coordinates of values of keys in \texttt{ll} specified in \texttt{ptsdict}.
2. Ordered list of y-coordinates of values of keys in \texttt{ll} specified in \texttt{ptsdict}.

6.2. Helper functions for plotting the geometric representation of matroids 349
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:

```python
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)]
[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
```

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:**

- \(M1\) – A matroid.
- \(B1\) – (optional) A 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) A 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) A 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:**

```python
from sage.matroids import matroids_plot_helpers

M = matroids.catalog.P7()
G = matroids_plot_helpers.geomrep(M) # needs sage.plot
G.show(xmin=-2, xmax=3, ymin=-2, ymax=3) # needs sage.plot
```

```python
M = matroids.catalog.P7()
G = matroids_plot_helpers.geomrep(M, lineorders1=[['f','e','d']]) # needs sage.plot
G.show(xmin=-2, xmax=3, ymin=-2, ymax=3) # needs sage.plot
```

```python
G = matroids_plot_helpers.geomrep(M)
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:**

- \(M\) – A matroid.
• B1 – A list of groundset elements of M that corresponds to a basis of matroid M.
• nB1 – A list of elements in the ground set of M that corresponds to M.simplify.groundset() \ B1.
• lps – A 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 can 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
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]])
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, lorders=None)
Determine if an order is specified for a line

INPUT:

- l – A line specified as a list of ground set elements.
- lorders – (optional) A 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 lorders that is setwise equal to l.
2. A list specifying an order on set(l) if 1. is True, otherwise an empty list.

EXAMPLES:

```python
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, [])
```

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 – A list of ordered lists specifying orders on subsets of ground set.
• lineorders2 – A 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:

```python
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 – A matroid.
• posdict – A 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:

```python
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, 1], ....: [0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1], ....: [0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0]])
```

```python
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, 0.66666666666666666666)}
```

```python
sage: matroids_plot_helpers.posdict_is_sane(M1, pos_dict)
#... needs sage.rings.finite_rings
True
```

```python
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, 0.66666666666666666666666666)}
```

```python
sage: matroids_plot_helpers.posdict_is_sane(M1, pos_dict)
#... needs sage.rings.finite_rings
False
```

```python
>>> 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(0), Integer(1), ....: Integer(0), Integer(1), Integer(0), Integer(1), Integer(1), Integer(1), Integer(0), Integer(0), Integer(1), ....: Integer(0)], ....: [Integer(0), Integer(0), Integer(1), ....: Integer(0)], ....: [Integer(0), Integer(0), Integer(1), ....: Integer(0)]},
```

(continues on next page)
Note: This method does NOT do any checks. \( M_1 \) is assumed to be a matroid and \( \text{posdict} \) is assumed to be a dictionary.

\[ \text{sage.matroids.matroids_plot_helpers.slp}(M1, \text{pos dict}=\text{None}, B=\text{None}) \]

Return simple matroid, loops and parallel elements of given matroid.

**INPUT:**

- \( M_1 \) – A matroid.
- \( \text{pos dict} \) – (optional) A dictionary containing non loopy elements of \( M \) as keys and their (x,y) positions. While simplifying the matroid, all except one element in a parallel class that is also specified in \( \text{pos dict} \) will be retained.
- \( B \) – (optional) A basis of \( M_1 \) that has been chosen for placement on vertices of triangle.

**OUTPUT:**

A tuple containing 3 elements in this order:

1. Simple matroid corresponding to \( M_1 \).
2. Loops of matroid \( M_1 \).
3. Elements that are in \( M_1.\text{groundset}() \) but not in ground set of 1 or in 2

**EXAMPLES:**

```python
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, 1, 0, 1, 0, 1],
                ....: [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0],
                ....: [0, 0, 1, 1, 1, 0, 1, 0, 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
```

(continues on next page)
sage: setprint([L,P])                           # needs sage.rings.finite_rings
[{10, 8, 9}, {7}]

sage: M1 = Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0],
... [0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1],
... [0, 0, 1, 1, 1, 0, 1, 0, 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
[{10, 8, 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(0), Integer(1)],
... [Integer(0), Integer(1), Integer(0), Integer(1), Integer(1),
... Integer(0), Integer(0), Integer(0),
... Integer(0), Integer(1)]])

>>> [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}]

(continues on next page)
>>> 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 – A list with 4 elements [xmin, xmax, ymin, ymax]
• x_i – New x values to track
• y_i – New y values to track
OUTPUT:
A 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 – A list of 3 lists of the form [x,y] where x and y are the Cartesian coordinates of a point.
OUTPUT:
A 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:
### 6.3 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

```python
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
```

To access the sets in this structure, simply iterate over them. The simplest way must be:

```python
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([(1, 2, 3, 4), ([1, 2], [3, 4], [1, 2, 4])])
```

#### Note

This method does NOT do any checks.
Or immediately use it to iterate:

```python
sage: from sage.matroids.set_system import SetSystem
game: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
[sage: [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:**

```python
game: from sage.matroids.set_system import SetSystem
game: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
game: S.is_connected()
True
game: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4]])
game: S.is_connected()
False
game: S = SetSystem([1], [])
game: S.is_connected()
True
```

(continues on next page)
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:

```python
>>> from sage.matroids.set_system import SetSystem
>>> S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
>>> type(S.__iter__())
<... 'sage.matroids.set_system.SetSystemIterator'>
```

6.4 Unpickling methods

Python saves objects by providing a pair \((f, \text{data})\) such that \(f(\text{data})\) reconstructs the object. This module collects the loading (\(_\text{unpickling}_\) 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` – an integer, expected to be 0
- `data` – a tuple \((E, R, \text{name}, \text{BB})\) in which \(E\) is the groundset of the matroid, \(R\) is the rank, \text{name} is a custom name, and \text{BB} is the bitpacked list of bases, as pickled by Sage’s `bitset_pickle`. 
OUTPUT:
A matroid.

**Warning:** Users should never call this function directly.

**EXAMPLES:**
```python
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid(matroids.catalog.Vamos())
sage: M == loads(dumps(M))  # indirect doctest
True
```

```python
>>> from sage.all import *
>>> from sage.matroids.advanced import *
>>> M = BasisMatroid(matroids.catalog.Vamos())
>>> M == loads(dumps(M))  # indirect doctest
True
```

```python
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:**
```python
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(2, 5)
sage: A == loads(dumps(A))  # indirect doctest
True
```

```python
sage: C = BinaryMatrix(2, 2, Matrix(GF(2), [[1, 1], [0, 1]]))
sage: C == loads(dumps(C))
True
```

```python
>>> from sage.all import *
>>> from sage.matroids.lean_matrix import *
>>> A = BinaryMatrix(Integer(2), Integer(5))
>>> A == loads(dumps(A))  # indirect doctest
True
```

```python
>>> A = BinaryMatrix(Integer(2), Integer(5))
>>> A == loads(dumps(A))  # indirect doctest
True
```

```python
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** – an integer (currently 0).  

6.4. Unpickling methods 361
• data—an tuple \((A, E, B, \text{name})\) where \(A\) is the representation matrix, \(E\) is the groundset of the matroid, \(B\) is the currently displayed basis, and \(\text{name}\) is a custom name.

OUTPUT:
A BinaryMatroid instance.

Warning: Users should never call this function directly.

EXAMPLES:

```python
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
```

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—an integer, expected to be 0

• data—an tuple \((E, CC, \text{name})\) in which \(E\) is the groundset of the matroid, \(CC\) is the dictionary of circuit closures, and \(\text{name}\) is a custom name.

OUTPUT:
A matroid.

Warning: Users should never call this function directly.

EXAMPLES:

```python
sage: M = matroids.catalog.Vamos()
sage: M == loads(dumps(M))  # indirect doctest
True
```
Unpickling 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** – an integer, expected to be 0
- **data** – a 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:
A matroid.

**Warning:** Users should never call this function directly.

**EXAMPLES:**

```python
sage: M = matroids.Theta(5)
sage: M == loads(dumps(M))  # indirect doctest
True
```

6.4. Unpickling methods
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 – an integer, expected to be 0
- data – a 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:**

A 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

>>>

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 – an integer (currently 0).
• data – a tuple consisting of a SageMath graph and a name.

OUTPUT:

A GraphicMatroid instance.

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

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 – an integer (currently 0).
• data – a tuple \((A, E, \text{reduced}, \text{name})\) where \(A\) is the representation matrix, \(E\) is the groundset of the matroid, \(\text{reduced}\) is a boolean indicating whether \(A\) is a reduced matrix, and \text{name} is a custom name.

OUTPUT:

A LinearMatroid instance.
**Warning:** Users should never call this function directly.

**EXAMPLES:**

```python
sage: M = Matroid(Matrix(GF(7), [[1, 0, 0, 1, 1], [0, 1, 0, 1, 2], ...
          [0, 1, 1, 1, 3]]))
M == loads(dumps(M))  # indirect doctest
True
sage: M.rename("U35")
U35
```

```python
>>> 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(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` – an integer, currently 0.
- `data` – a 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:**

A `MinorMatroid` instance.

**Warning:** Users should never call this function directly.

**EXAMPLES:**

```python
sage: M = matroids.catalog.Vamos().minor('abc', 'g')
M == loads(dumps(M))  # indirect doctest
True
```

```python
>>> 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 a PlusMinusOneMatrix object (internal Sage data structure).

**Warning:** Users should not call this method directly.

**EXAMPLES:**

```python
sage: from sage.matroids.lean_matrix import *
sage: A = PlusMinusOneMatrix(2, 5)
sage: A == loads(dumps(A))  # indirect doctest
True
```

```python
>>> 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:**

```python
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
```

```python
>>> 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'),
                                                      [[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** – an integer (currently 0).
- **data** – a tuple \((A, E, B, \text{name})\) where \(A\) is the representation matrix, \(E\) is the groundset of the matroid, \(B\) is the currently displayed basis, and \(\text{name}\) is a custom name.

**OUTPUT:**
A TernaryMatroid instance.

**Warning:** Users should never call this function directly.

**EXAMPLES:**

```python
def pickle_rational_matrix(version, data):
    return 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

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 – an integer (currently 0).
• data – a 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:

A RegularMatroid instance.

Warning: Users should never call this function directly.

EXAMPLES:

sage: M = matroids.catalog.R10()
sage: M == loads(dumps(M))  # indirect doctest
True
sage: M.rename("R_{10}")
sage: 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))

>>> C = TernaryMatrix(Integer(2), Integer(2), Matrix(GF(Integer(3)), [[Integer(1), Integer(1)],
... [Integer(0), Integer(1)]]))

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 – an integer (currently 0).
- data – a 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:

A TernaryMatroid instance.

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(1), Integer(0), Integer(1), Integer(1)],
... Integer(0), Integer(1), Integer(0), Integer(1),
... Integer(0), Integer(0), Integer(1), Integer(1)]]))

(continues on next page)
>>> M.rename("U34")
>>> loads(dumps(M))
U34
CHAPTER SEVEN

INDICES AND TABLES

• Index
• Module Index
• Search Page
m

sage.matroids.advanced, 321
sage.matroids.basis_exchange_matroid, 311
sage.matroids.basis_matroid, 211
sage.matroids.circuit_closures_matroid, 227
sage.matroids.circuits_matroid, 217
sage.matroids.constructor, 1
sage.matroids.database_collections, 129
sage.matroids.database_matroids, 133
sage.matroids.dual_matroid, 305
sage.matroids.extension, 332
sage.matroids.flats_matroid, 231
sage.matroids.graphic_matroid, 290
sage.matroids.lean_matrix, 337
sage.matroids.linear_matroid, 237
sage.matroids.matroid, 21
sage.matroids.matroids_catalog, 125
sage.matroids.matroids_plot_helpers, 346
sage.matroids.minor_matroid, 308
sage.matroids.rank_matroid, 288
sage.matroids.set_system, 358
sage.matroids.unpickling, 360
sage.matroids.utilities, 322
INDEX

A
A9 () (in module sage.matroids.database_matroids), 133
addlp() (in module sage.matroids.matroids_plot_helpers), 347
addnontripts() (in module sage.matroids.matroids_plot_helpers), 348
AF12 () (in module sage.matroids.database_matroids), 133
AG() (in module sage.matroids.database_matroids), 134
AG23 () (in module sage.matroids.database_matroids), 134
AG23minus() (in module sage.matroids.database_matroids), 135
AG23minusDY() (in module sage.matroids.database_matroids), 136
AG32() (in module sage.matroids.database_matroids), 136
AG32prime() (in module sage.matroids.database_matroids), 137
AK12 () (in module sage.matroids.database_matroids), 139
AllMatroids() (in module sage.matroids.database_collections), 129
augment() (sage.matroids.matroid.Matrix method), 31
augmented_bergman_complex() (sage.matroids.matroid.Matrix method), 32
automorphism_group() (sage.matroids.Matrix method), 33

B
base_ring() (sage.matroids.lean_matrix.BinaryMatrix method), 338
base_ring() (sage.matroids.lean_matrix.GenericMatrix method), 339
base_ring() (sage.matroids.lean_matrix.LeanMatrix method), 340
base_ring() (sage.matroids.lean_matrix.PlusMinusOneMatrix method), 342
base_ring() (sage.matroids.lean_matrix.QuaternaryMatrix method), 343
base_ring() (sage.matroids.lean_matrix.RationalMatrix method), 344
base_ring() (sage.matroids.linear_matroid.BinaryMatroid method), 241
base_ring() (sage.matroids.linear_matroid.LinearMatroid method), 246
base_ring() (sage.matroids.linear_matroid.QuaternaryMatroid method), 273
base_ring() (sage.matroids.linear_matroid.RegularMatroid method), 277
base_ring() (sage.matroids.linear_matroid.TernaryMatroid method), 284
bases() (sage.matroids.basis_matroid.BasisMatroid method), 213
bases() (sage.matroids.matroid.termination.Basis method), 246
bases_count() (sage.matroids.basis_exchange_matroid.termination.Basis method), 312
bases_count() (sage.matroids.basis_matroid.termination.Basis method), 214
bases_count() (sage.matroids.linear_matroid.termination.Basis method), 277
bases_iterator() (sage.matroids.circuits_matrix.termination.CircuitsMatroid method), 217
bases_iterator() (sage.matroids.matroid.termination.Basis method), 34
basis() (sage.matroids.basis_exchange_matroid.termination.Basis method), 312
basis() (sage.matroids.matroid.termination.Basis method), 35
BasisExchangeMatroid (class in sage.matroids.termination), 377
BasisMatroid (class in sage.matroids.termination), 212
BB9 () (in module sage.matroids.database_matroids), 139
BB9gDY() (in module sage.matroids.database_matroids), 140
bergman_complex() (sage.matroids.matroid.termination method), 35
BetsyRoss() (in module sage.matroids.database_matroids), 140
bicycle_dimension() (sage.matroids.linear_matroid.BinaryMatroid method), 241
bicycle_dimension() (sage.matroids.linear_matroid.LinearMatroid method), 246

377
bicycle_dimension() (sage.matroids.linear_matroid.TernaryMatroid method), 285
binary_matroid() (sage.matroids.linear_matroid.BinaryMatroid method), 214
binary_matroid() (sage.matroids.linear_matroid.RegularMatroid method), 278
binary_matroid() (sage.matroids.matroid.Matroid method), 36
BinaryMatrix (class in sage.matroids.lean_matrix), 337
BinaryMatroid (class in sage.matroids.linear_matroid), 239
Block_9_4() (in module sage.matroids.database_matroids), 141
Block_10_5() (in module sage.matroids.database_matroids), 141
BrettellMatroids() (in module sage.matroids.database_collections), 131
broken_circuit_complex() (sage.matroids.circuit_closures_matroid.CircuitClosuresMatroid method), 218
broken_circuit_complex() (sage.matroids.matroid.Matroid method), 37
broken_circuits() (sage.matroids.matroid.Matroid method), 37
brown_invariant() (sage.matroids.linear_matroid.BinaryMatroid method), 242
characteristic() (sage.matroids.linear_matroid.TernaryMatroid method), 285
characteristic() (sage.matroids.linear_matroid.BinaryMatroid method), 214
characteristic() (sage.matroids.linear_matroid-RegularMatroid method), 278
characteristic() (sage.matroids.matroid.Matroid method), 36
characteristic() (sage.matroids.linear_matroid.RegularMatroid method), 278
characteristic() (sage.matroids.linear_matroid.TernaryMatroid method), 285
characteristic() (sage.matroids.matroid.Matroid method), 36
chordality() (sage.matroids.matroid.Matroid method), 39
chow_ring() (sage.matroids.matroid.Matroid method), 39
circuit() (sage.matroids.matroid.Matroid method), 41
circuit_closures() (sage.matroids.circuit_closures_matroid.CircuitClosuresMatroid method), 229
circuit_closures() (sage.matroids.matroid.Matroid method), 42
CircuitClosuresMatroid (class in sage.matroids.circuit_closures_matroid), 228
circuits() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 313
circuits() (sage.matroids.circuit_matroid.CircuitMatroid method), 219
circuits() (sage.matroids.matroid.Matroid method), 43
circuits_iterator() (sage.matroids.circuit_matroid.CircuitMatroid method), 220
circuits_iterator() (sage.matroids.matroid.Matroid method), 43
CircuitsMatroid (class in sage.matroids.circuit_matroid), 217
closure() (sage.matroids.matroid.Matroid method), 44
cmp_elements_key() (in module sage.matroids.utilities), 322
cobasis() (sage.matroids.matroid.Matroid method), 44
cocircuit() (sage.matroids.matroid.Matroid method), 45
cocircuits() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 313
cocircuits() (sage.matroids.matroid.Matroid method), 46
cocircuits_iterator() (sage.matroids.matroid.Matroid method), 46
coclosure() (sage.matroids.matroid.Matroid method), 47
coextension() (sage.matroids.matroid.Matroid method), 47
coextensions() (sage.matroids.matroid.Matroid method), 48
coflats() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 314
coflats() (sage.matroids.matroid.Matroid method), 49
coloops() (sage.matroids.matroid.Matroid method), 50
CompleteGraphic() (in module sage.matroids.database_matroids), 141
components() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 314
components() (sage.matroids.matroid.Matroid method), 50
connectivity() (sage.matroids.matroid.Matroid method), 51
contract() (sage.matroids.matroid.Matroid method), 51
corank() (sage.matroids.matroid.Matroid method), 51
cosimplify() (sage.matroids.matroid.Matroid method), 54
createline() (in module sage.matroids.matroids_plot_helpers), 349
cross_ratio() (sage.matroids.linear_matroid.LinearMatroid method), 247
cross_ratios() (sage.matroids.linear_matroid.LinearMatroid method), 248
CutNode (class in sage.matroids.extension), 333
D
D10() (in module sage.matroids.database_matroids), 142
D16() (in module sage.matroids.database_matroids), 142
delete() (sage.matroids.matroid.Matroid method), 54
dependent_r_sets() (sage.matroids.matroid.Matroid method), 55
dependent_sets() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 315
dependent_sets() (sage.matroids.circuits_matroid.CircuitsMatroid method), 220
dependent_sets() (sage.matroids.matroid.Matroid method), 55
dependent_sets_iterator() (sage.matroids.matroid.Matroid method), 56
direct_sum() (sage.matroids.matroid.Matroid method), 56
dual() (sage.matroids.basis_matroid.BasisMatroid method), 214
dual() (sage.matroids.dual_matroid.DualMatroid method), 306
dual() (sage.matroids.linear_matroid.LinearMatroid method), 249
dual() (sage.matroids.matroid.Matroid method), 57
DualMatroid (class in sage.matroids.dual_matroid), 306
E
equals() (sage.matroids.matroid.Matroid method), 58
ExtendedBinaryGolayCode() (in module sage.matroids.database_matroids), 143
ExtendedTernaryGolayCode() (in module sage.matroids.database_matroids), 143
extension() (sage.matroids.matroid.Matroid method), 59
extensions() (sage.matroids.matroid.Matroid method), 60
full_rank() (sage.matroids.circuit_closures_matroid.CircuitClosuresMatroid method), 230
full_rank() (sage.matroids.circuits_matroid.CircuitsMatroid method), 221
full_rank() (sage.matroids.flats_matroid.FlatsMatroid method), 232
full_rank() (sage.matroids.matroid.Matroid method), 63
fundamental_circuit() (sage.matroids.matroid.Matroid method), 63
fundamental_cocircuit() (sage.matroids.matroid.Matroid method), 64
fundamental_cocycle() (sage.matroids.linear_matroid.LinearMatroid method), 250
fundamental_cycle() (sage.matroids.linear_matroid.LinearMatroid method), 250
FV14() (in module sage.matroids.database_matroids), 151
FX9() (in module sage.matroids.database_matroids), 151
FY10() (in module sage.matroids.database_matroids), 151
FZ10() (in module sage.matroids.database_matroids), 151
FZ12() (in module sage.matroids.database_matroids), 152
G
generic_identity() (in module sage.matroids.lean_matrix), 345
GenericMatrix (class in sage.matroids.lean_matrix), 338
geomrep() (in module sage.matroids.matroids_plot_helpers), 350
get_nonisomorphic_matroids() (in module sage.matroids.utilities), 322
girth() (sage.matroids.circuits_matroid.CircuitsMatroid method), 221
girth() (sage.matroids.matroid.Matroid method), 64
GK10() (in module sage.matroids.database_matroids), 154
GP10() (in module sage.matroids.database_matroids), 154
GP12() (in module sage.matroids.database_matroids), 155
graph() (sage.matroids.graphic_matroid.GraphicMatroid method), 293
graphic_coextension() (sage.matroids.graphic_matroid.GraphicMatroid method), 293
graphic_coextensions() (sage.matroids.graphic_matroid.GraphicMatroid method), 294
graphic_extension() (sage.matroids.graphic_matroid.GraphicMatroid method), 296
graphic_extensions() (sage.matroids.graphic_matroid.GraphicMatroid method), 297
GraphicMatroid (class in sage.matroids.graphic_matroid), 291
groundset() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 317
groundset() (sage.matroids.circuit_closures_matroid.CircuitClosuresMatroid method), 230
groundset() (sage.matroids.circuits_matroid.CircuitsMatroid method), 221
groundset() (sage.matroids.dual_matroid.DualMatroid method), 307
groundset() (sage.matroids.flats_matroid.FlatsMatroid method), 233
groundset() (sage.matroids.graphic_matroid.GraphicMatroid method), 298
groundset() (sage.matroids.matroid.Matroid method), 65
groundset() (sage.matroids.minor_matroid.MinorMatroid method), 310
groundset() (sage.matroids.rank_matroid.RankMatroid method), 289
groundset_list() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 317
groundset_to_edges() (sage.matroids.matroid.Matroid method), 298
H
has_field_minor() (sage.matroids.linear_matroid.LinearMatroid method), 251
has_line_minor() (sage.matroids.linear_matroid.LinearMatroid method), 251
has_line_minor() (sage.matroids.linear_matroid.RegularMatroid method), 278
has_line_minor() (sage.matroids.matroid.Matroid method), 65
has_minor() (sage.matroids.matroid.Matroid method), 66
hyperplanes() (sage.matroids.matroid.Matroid method), 67
I
independence_matroid_polytope() (sage.matroids.matroid.Matroid method), 68
independent_r_sets() (sage.matroids.matroid.Matroid method), 68
independent_sets() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 318
independent_sets() (sage.matroids.circuits_matroid.CircuitsMatroid method), 222
independent_sets() (sage.matroids.matroid.MatrixMethod), 68
independent_sets_iterator() (sage.matroids.matroid.MatrixMethod), 69
intersection() (sage.matroids.matroid.MatrixMethod), 70
intersection_unweighted() (sage.matroids.matroid.MatrixMethod), 71
is_3connected() (sage.matroids.matroid.MatrixMethod), 71
is_4connected() (sage.matroids.matroid.MatrixMethod), 73
is_basis() (sage.matroids.matroid.MatrixMethod), 74
is_binary() (sage.matroids.linear_matroid.BinaryMatroid), 75
is_chordal() (sage.matroids.matroid.MatrixMethod), 75
is_circuit() (sage.matroids.matroid.MatrixMethod), 76
is_circuit_chordal() (sage.matroids.matroid.MatrixMethod), 77
is_closed() (sage.matroids.matroid.MatrixMethod), 78
is_cobasis() (sage.matroids.matroid.MatrixMethod), 78
is_codirected() (sage.matroids.matroid.MatrixMethod), 79
is_codependent() (sage.matroids.matroid.MatrixMethod), 80
is_coindependent() (sage.matroids.matroid.MatrixMethod), 81
is_connected() (sage.matroids.matroid.MatrixMethod), 81
is_connected() (sage.matroids.set_system.SetSystem), 359
is_cocircuit() (sage.matroids.matroid.MatrixMethod), 78
is_coclosed() (sage.matroids.matroid.MatrixMethod), 79
is_codependent() (sage.matroids.matroid.MatrixMethod), 80
is_coindependent() (sage.matroids.matroid.MatrixMethod), 81
is_connected() (sage.matroids.matroid.MatrixMethod), 81
is_connected() (sage.matroids.set_system.SetSystem), 359
is_cosimple() (sage.matroids.matroid.MatrixMethod), 82
is_dependent() (sage.matroids.matroid.MatrixMethod), 82
is_distinguished() (sage.matroids.basis_matrix.MatrixMethod), 215
is_field_equivalent() (sage.matroids.linear_matroid.LinearMatroid), 252
is_field_isomorphic() (sage.matroids.linear_matroid.LinearMatroid), 254
is_field_isomorphism() (sage.matroids.linear_matroid.LinearMatroid), 256
is_valid() (sage.matroids.graphic_matroid.GraphicMatroid), 299
is_valid() (sage.matroids.linear_matroid.BinaryMatroid), 243
is_valid() (sage.matroids.linear_matroid.RegularMatroid), 280
is_valid() (sage.matroids.matroid.MatrixMethod), 83
is_valid() (sage.matroids.matroid.MatrixMethod), 84
is_valid() (sage.matroids.matroid.MatrixMethod), 85
is_valid() (sage.matroids.matroid.MatrixMethod), 89
is_valid() (sage.matroids.matroid.MatrixMethod), 90
is_valid() (sage.matroids.matroid.MatrixMethod), 92
is_valid() (sage.matroids.linear_matroid.BinaryMatroid), 222
is_valid() (sage.matroids.linear_matroid.BinaryMatroid), 222
is_valid() (sage.matroids.linear_matroid.RegularMatroid), 299
is_valid() (sage.matroids.linear_matroid.RegularMatroid), 281
is_valid() (sage.matroids.matroid.MatrixMethod), 94
is_valid() (sage.matroids.matroid.MatrixMethod), 95
is_valid() (sage.matroids.matroid.MatrixMethod), 96
is_valid() (sage.matroids.linear_matroid.TernaryMatroid), 286
is_valid() (sage.matroids.matroid.MatrixMethod), 97
is_valid() (sage.matroids.basis_exchange_matrix), 318
is_valid() (sage.matroids.circuits_matroid.CircuitsMatroid), 222
is_valid() (sage.matroids.dual_matroid.DualMatroid), 307
is_valid() (sage.matroids.flats_matroid.FlatsMatroid), 233
is_valid() (sage.matroids.graphic_matroid.GraphicMatroid), 299
is_valid() (sage.matroids.linearmatroid.BinaryMatroid method), 244
is_valid() (sage.matroids.linearmatroid.LinearMatroid method), 257
is_valid() (sage.matroids.linearmatroid.QuaternaryMatroid method), 275
is_valid() (sage.matroids.linearmatroid.RegularMatroid method), 281
is_valid() (sage.matroids.linearmatroid.TernaryMatroid method), 286
is_valid() (sage.matroids.matroid.Matroid method), 98
isomorphism() (sage.matroids.matroid.Matroid method), 98
it() (in module sage.matroids.matroids_plot_helpers), 351
J
J() (in module sage.matroids.database_matroids), 155
K
K4() (in module sage.matroids.database_matroids), 157
K5() (in module sage.matroids.database_matroids), 158
K5dual() (in module sage.matroids.database_matroids), 158
K33() (in module sage.matroids.database_matroids), 156
K33dual() (in module sage.matroids.database_matroids), 156
k_closure() (sage.matroids.matroid.Matroid method), 99
KB12() (in module sage.matroids.database_matroids), 159
KF10() (in module sage.matroids.database_matroids), 159
KP8() (in module sage.matroids.database_matroids), 159
KQ9() (in module sage.matroids.database_matroids), 160
KR9() (in module sage.matroids.database_matroids), 160
KT10() (in module sage.matroids.database_matroids), 161
L
L8() (in module sage.matroids.database_matroids), 161
lattice_of_flats() (sage.matroids.flatsmatroid.FlatsMatroid method), 235
lattice_of_flats() (sage.matroids.matroid.Matroid method), 101
LeanMatrix (class in sage.matroids.lean_matrix), 339
lift_cross_ratios() (in module sage.matroids.utilities), 323
lift_map() (in module sage.matroids.utilities), 325
line_hasorder() (in module sage.matroids.matroids_plot_helpers), 353
linear_coextension() (sage.matroids.linearmatroid.LinearMatroid method), 258
linear_coextension_cochains() (sage.matroids.linearmatroid.LinearMatroid method), 259
linear_coextensions() (sage.matroids.linearmatroid.LinearMatroid method), 261
linear_extension() (sage.matroids.linearmatroid.LinearMatroid method), 262
linear_extension_chains() (sage.matroids.linearmatroid.LinearMatroid method), 263
linear_extensions() (sage.matroids.linearmatroid.LinearMatroid method), 265
linear_subclasses() (sage.matroids.matroid.Matroid method), 101
LinearMatroid (class in sage.matroids.linearmatroid), 245
LinearSubclasses (class in sage.matroids.extension), 333
LinearSubclassesIter (class in sage.matroids.extension), 334
lineorders_union() (in module sage.matroids.matroids_plot_helpers), 353
link() (sage.matroids.matroid.Matroid method), 102
loops() (sage.matroids.matroid.Matroid method), 103
LP8() (in module sage.matroids.database_matroids), 162
M
M8591() (in module sage.matroids.database_matroids), 162
make_regular_matroid_from_matroid() (in module sage.matroids.utilities), 326
Matroid (class in sage.matroids.matroid), 29
Matroid() (in module sage.matroids.constructor), 3
matroid_polytope() (sage.matroids.matroid.Matroid method), 103
MatroidExtensions (class in sage.matroids.extension), 334
max_coinddependent() (sage.matroids.matroid.Matroid method), 104
max_independent() (sage.matroids.matroid.Matroid method), 105
max_weight_coinddependent() (sage.matroids.matroid.Matroid method), 105
max_weight_independent() (sage.matroids.matroid.Matroid method), 107
minor() (sage.matroids.matroid.Matroid method), 108
MinorMatroid (class in sage.matroids.minor_matroid), 310
modular_cut() (sage.matroids.matroid.Matroid method), 110
module
sage.matroids.advanced, 321
sage.matroids.basis_exchange_matroid, 311
sage.matroids.basis_matroid, 211
sage.matroids.circuit_closures_matroid, 227
sage.matroids.circuits_matroid, 217
sage.matroids.constructor, 1
sage.matroids.database_collections, 129
sage.matroids.database_matroids, 133
sage.matroids.dual_matroid, 305
sage.matroids.extension, 332
sage.matroids.flats_matroid, 231
sage.matroids.graphic_matroid, 290
sage.matroids.lean_matrix, 337
sage.matroids.linear_matroid, 237
sage.matroids.matroid, 21
sage.matroids.matroids_catalog, 125
sage.matroids.matroids_plot_helpers, 346
sage.matroids.minor_matroid, 308
sage.matroids.rank_matroid, 288
sage.matroids.set_system, 358
sage.matroids.unpickling, 360
sage.matroids.utilities, 322

N

N1() (in module sage.matroids.database_matroids), 163
N2() (in module sage.matroids.database_matroids), 163
N3() (in module sage.matroids.database_matroids), 163
N3pp() (in module sage.matroids.database_matroids), 164
N4() (in module sage.matroids.database_matroids), 164
ncols() (sage.matroids.lean_matrix.LeanMatrix method), 340
NestOfTwistedCubes() (in module sage.matroids.database_matroids), 165
newlabel() (in module sage.matroids.utilities), 326
no_broken_circuits_facets() (sage.matroids.circuits_matroid.CircuitsMatroid method), 224
no_broken_circuits_sets() (sage.matroids.circuits_matroid.CircuitsMatroid method), 224
no_broken_circuits_sets() (sage.matroids.matroid.Matroid method), 111
no_broken_circuits_sets_iterator() (sage.matroids.matroid.Matroid method), 112
nonbases() (sage.matroids.basis_matroid.BasisMatroid method), 215
nonbases() (sage.matroids.matroid.Matroid method), 113
nonbases_iterator() (sage.matroids.matroid.Matroid method), 114
noncospanning_cocircuits() (sage.matroids.matroid.Matroid method), 114
NonDesargues() (in module sage.matroids.database_matroids), 165
NonFano() (in module sage.matroids.database_matroids), 166
NonFanoDual() (in module sage.matroids.database_matroids), 166
NonPappus() (in module sage.matroids.database_matroids), 167
nonspanning_circuit_closures() (sage.matroids.matroid.Matroid method), 115
nonspanning_circuits() (sage.matroids.basis_exchange_matroid.BasisExchangeMatroid method), 319
nonspanning_circuits() (sage.matroids.circuits_matroid.CircuitsMatroid method), 225
nonspanning_circuits() (sage.matroids.matroid.Matroid method), 115
nonspanning_circuits_iterator() (sage.matroids.circuits_matroid.CircuitsMatroid method), 225
nonspanning_circuits_iterator() (sage.matroids.matroid.Matroid method), 116
NonVamos() (in module sage.matroids.database_matroids), 168
NotP8() (in module sage.matroids.database_matroids), 168
nrows() (sage.matroids.lean_matrix.LeanMatrix method), 341

O

O7() (in module sage.matroids.database_matroids), 169
one_sum() (sage.matroids.graphic_matroid.GraphicMatroid method), 300
orlik_solomon_algebra() (sage.matroids.matroid.Matroid method), 116
orlik_terao_algebra() (sage.matroids.linear_matroid.LinearMatroid method), 266
OW14() (in module sage.matroids.database_matroids), 169
OxleyMatroids() (in module sage.matroids.database_collections), 132

P

P6() (in module sage.matroids.database_matroids), 169
P7() (in module sage.matroids.database_matroids), 170
P8() (in module sage.matroids.database_matroids), 171
P8p() (in module sage.matroids.database_matroids), 171
P8pp() (in module sage.matroids.database_matroids), 172
P9() (in module sage.matroids.database_matroids), 172
Pappus() (in module sage.matroids.database_matroids), 175
partition() (sage.matroids.matroid.Matroid method), 117
PG() (in module sage.matroids.database_matroids), 172
PG23() (in module sage.matroids.database_matroids), 173
FK10() (in module sage.matroids.database_matroids), 174
plot() (sage.matroids.matroid.Matroid method), 118

PlusMinusOneMatrix (class in sage.matroids.lean_matrix), 341

posdict_is_sane() (in module sage.matroids.matroids_plot_helpers), 354

PP9() (in module sage.matroids.database_matroids), 174
PP10() (in module sage.matroids.database_matroids), 174

Psi() (in module sage.matroids.database_matroids), 176

Q6() (in module sage.matroids.database_matroids), 178
Q8() (in module sage.matroids.database_matroids), 178
Q10() (in module sage.matroids.database_matroids), 177
QuaternaryMatrix (class in sage.matroids.lean_matrix), 342
QuaternaryMatroid (class in sage.matroids.linear_matroid), 272

R6() (in module sage.matroids.database_matroids), 181
R8() (in module sage.matroids.database_matroids), 182
R9() (in module sage.matroids.database_matroids), 183
R9A() (in module sage.matroids.database_matroids), 183
R9B() (in module sage.matroids.database_matroids), 184
R10() (in module sage.matroids.database_matroids), 179
R12() (in module sage.matroids.database_matroids), 181

rank() (sage.matroids.matroid.Matroid method), 119
RelaxedNonFano() (in module sage.matroids.database_matroids), 184
representation() (sage.matroids.linear_matroid.LinearMatroid method), 267
representation_vectors() (sage.matroids.linear_matroid.LinearMatroid method), 271

S
S8() (in module sage.matroids.database_matroids), 184

sage.matroids.advanced module, 321
sage.matroids.basis_exchange_matroid module, 311
sage.matroids.basis_matroid module, 211
sage.matroids.circuit_closures_matroid module, 227
sage.matroids.circuits_matroid module, 217
sage.matroids.constructor module, 1
sage.matroids.database_collections module, 129
sage.matroids.database_matroids module, 133
sage.matroids.dual_matroid module, 305
sage.matroids.extension module, 332
sage.matroids.flats_matroid module, 231
sage.matroids.graphic_matroid module, 290
sage.matroids.lean_matrix module, 337
sage.matroids.linear_matroid module, 237
sage.matroids.matroid module, 21
sage.matroids.matroids_catalog...