Sage Reference Manual: Matroid Theory

Release 8.2

The Sage Development Team

May 07, 2018

CONTENTS

CHAPTER

ONE

BASICS

1.1 Matroid construction

1.1.1 Theory

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

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

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

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

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

1.1.2 Built-in matroids

For built-in matroids, do the following:

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

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

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

or:

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

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

```
sage: F7 = matroids.named_matroids.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:

```
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.named_matroids.Fano())
True

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

AUTHORS:

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

1.1.4 Functions

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

Construct a matroid.

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

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

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

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

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

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

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

or:

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

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

INPUT:

- `groundset` – (optional) 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.
- `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.
- `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 [MatroidDatabase] 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*, *CircuitClosuresMatroid*, *LinearMatroid*, *BinaryMatroid*, *TernaryMatroid*, *QuaternaryMatroid*, *RegularMatroid*, or *RankMatroid*. To import these classes (and other useful functions) directly into Sage's main namespace, type:

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

See *sage.matroids.advanced*.

EXAMPLES:

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

1. List of bases:

   All of the following inputs are allowed, and equivalent:

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

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

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

   Bases may be repeated:

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

   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
```

3. List of circuits:

```
sage: M1 = Matroid(groundset='abc', circuits=['bc'])
sage: M2 = Matroid(bases=['ab', 'ac'])
sage: M1 == M2
True
```

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

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

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

```
sage: M = Matroid('abcd', circuits=['ab', 'acd'])
sage: M.is_valid()
True
sage: [sorted(C) for C in M.circuits()]
[['a']]
```

4. Graph:

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

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

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

```
sage: G = Graph([(0, 1, 'a'), (0, 2, 'b'), (1, 2, 'c')])
sage: M = Matroid(G)
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:

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

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

sage: G1 = graphs.CycleGraph(3); G2 = graphs.DiamondGraph() sage: G = G1.disjoint_union(G2) sage: M = Matroid(G) sage: 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

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

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

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

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

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

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

```
sage: Matroid(graph=':I`AKGsaOs`cI]Gb~')
Graphic matroid of rank 9 on 17 elements
```

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

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

5. Matrix:

   The basic input is a `Sage matrix`:

```
sage: A = Matrix(GF(2), [[1, 0, 0, 1, 1, 0],
....:                    [0, 1, 0, 1, 0, 1],
....:                    [0, 0, 1, 0, 1, 1]])
sage: M = Matroid(matrix=A)
sage: M.is_isomorphic(matroids.CompleteGraphic(4))
True
```

   Various shortcuts are possible:

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

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

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

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

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

We automatically create an optimized subclass, if available:

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

Otherwise the generic LinearMatroid class is used:

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

An integer matrix is automatically converted to a matrix over $\mathbf{Q}$. If you really want integers, you can specify the ring explicitly:

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

6. Rank function:

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

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

7. Circuit closures:

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

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

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

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

8. RevLex-Index:

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

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

```
 frozenset({'a', 'b', 'e', 'f'}),
 frozenset({'a', 'c', 'e', 'f'}),
 frozenset({'b', 'c', 'e', 'f'}),
 frozenset({'b', 'd', 'e', 'f'}),
 frozenset({'c', 'd', 'e', 'f'})]
```

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

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

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

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

9. Matroid:

   Most of the time, the matroid itself is returned:

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

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

```
sage: M = Matroid(circuit_closures={2:['adb', 'bec', 'cfa',
....:                                  'def'], 3:['abcdef']})
sage: N = Matroid(M, regular=True)
sage: N
Regular matroid of rank 3 on 6 elements with 16 bases
sage: Matrix(N)
[1 0 0 1 1 0]
[0 1 0 1 1 1]
[0 0 1 0 1 1]
```

The `regular` option:

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

```
sage: M = matroids.named_matroids.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)
sage: 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()
True
sage: N.is_isomorphic(M)
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).

- **Ground set:**
  - *groundset()*
  - *size()*
- **Rank, bases, circuits, closure**
  - *rank()*
  - *full_rank()*
  - *basis()*
  - *max_independent()*
  - *circuit()*
  - *fundamental_circuit()*
  - *closure()*
  - *augment()*
  - *corank()*
  - *full_corank()*
  - *cobasis()*

  - *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()*
- **Representation**
  - *binary_matroid()*
  - *is_binary()*
  - *ternary_matroid()*
  - *is_ternary()*
- **Optimization**
  - *max_weight_independent()*

  - *max_weight_coindependent()*
  - *is_max_weight_independent_generic()*
  - *intersection()*
  - *intersection_unweighted()*
- **Invariants**
  - *tutte_polynomial()*
  - *flat_cover()*
- **Visualization**
  - *show()*
  - *plot()*
- **Construction**
  - *union()*
- **Misc**
  - *broken_circuit_complex()*
  - *chow_ring()*
  - *matroid_polytope()*
  - *independence_matroid_polytope()*
  - *orlik_solomon_algebra()*

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._<tab>`. Then `M._rank?` followed by `<tab>` will bring up the documentation string of the `_rank()` method.

1.2.2 Creating new Matroid subclasses

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

EXAMPLES:

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

```
sage: import sage.matroids.matroid
sage: class PartitionMatroid(sage.matroids.matroid.Matroid):
....:     def __init__(self, partition):
....:         self.partition = partition
....:         E = set()
....:         for P in partition:
....:             E.update(P)
....:         self.E = frozenset(E)
....:     def groundset(self):
....:         return self.E
....:     def _rank(self, X):
....:         X2 = set(X)
....:         used_indices = set()
....:         rk = 0
....:         while len(X2) > 0:
....:             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)
....:                         rk = rk + 1
....:                     break
....:         return rk
....:
sage: M = PartitionMatroid([[1, 2], [3, 4, 5], [6, 7]])
sage: M.full_rank()
3
sage: M.tutte_polynomial(var('x'), var('y'))
x^2*y^2 + 2*x*y^3 + y^4 + x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

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

Necessary:

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

Representation:

- `def _repr_(self)`

Comparison:

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

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

Copying, loading, saving:

- `def __copy__(self)`

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

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

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

1.2.3 Some examples

EXAMPLES:

Construction:

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

Minors:

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

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

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

1.2.4 REFERENCES

- [BC1977]
- [Cun1986]
- [CMO2011]
- [CMO2012]
- [GG2012]
- [GR2001]
- [Hli2006]

- [Hoc]
- [Lyo2003]
- [Oxl1992]
- [Oxl2011]
- [Pen2012]
- [PvZ2010]
- [Raj1987]

AUTHORS:

- Michael Welsh (2013-04-03): Changed flats() to use SetSystem
- Michael Welsh (2013-04-01): Added is_3connected(), using naive algorithm
- Rudi Pendavingh, Stefan van Zwam (2013-04-01): initial version

1.2.5 Methods

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

Bases: sage.structure.sage_object.SageObject

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

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

EXAMPLES:

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

```
sage: class PartitionMatroid(sage.matroids.matroid.Matroid):
....:     def __init__(self, partition):
....:         self.partition = partition
....:         E = set()
....:         for P in partition:
....:             E.update(P)
....:         self.E = frozenset(E)
....:     def groundset(self):
....:         return self.E
....:     def _rank(self, X):
....:         X2 = set(X)
....:         used_indices = set()
....:         rk = 0
....:         while len(X2) > 0:
....:             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)
....:                         rk = rk + 1
....:                     break
....:         return rk
....:
sage: M = PartitionMatroid([[1, 2], [3, 4, 5], [6, 7]])
sage: M.full_rank()
```

```
3
sage: M.tutte_polynomial(var('x'), var('y'))
x^2*y^2 + 2*x*y^3 + y^4 + x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

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

Necessary:

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

Representation:

- `def _repr_(self)`

Comparison:

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

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

Copying, loading, saving:

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

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

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

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

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

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

---

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

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

INPUT:

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

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

OUTPUT:

A subset of $Y - X$.

EXAMPLES:

```
sage: M = matroids.named_matroids.Vamos()
sage: sorted(M.augment(['a'], ['e', 'f', 'g', 'h']))
['e', 'g', 'h']
sage: sorted(M.augment(['a']))
['b', 'c', 'e']
sage: sorted(M.augment(['x']))
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
sage: sorted(M.augment(['a'], ['x']))
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

bases()

Return the list of bases of the matroid.

A *basis* is a maximal independent set.

OUTPUT:

An iterable containing all bases of the matroid.

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_r_sets()

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:

Set of elements.

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

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` – (default: `True`), a Boolean; if `True`, any output will be a binary matroid representing `self`; if `False`, any output will represent `self` if and only if the matroid is binary

OUTPUT:

Either a BinaryMatroid, or `None`

ALGORITHM:

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

See also:

`M.local_binary_matroid()`

EXAMPLES:

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

broken_circuit_complex(*ordering=None*)

Return the broken circuit complex of `self`.

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

INPUT:

- `ordering` – a total ordering of the groundset given as a list

OUTPUT:

A simplicial complex of the NBC sets under inclusion.

EXAMPLES:

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

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

broken_circuits(*ordering=None*)

Return the list of broken circuits of `self`.

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

INPUT:

- `ordering` – a total ordering of the groundset given as a list

EXAMPLES:

```
sage: M = Matroid(circuits=[[1,2,3], [3,4,5], [1,2,4,5]])
sage: M.broken_circuits()
frozenset({frozenset({2, 3}), frozenset({4, 5}), frozenset({2, 4, 5})})
sage: M.broken_circuits([5,4,3,2,1])
frozenset({frozenset({1, 2}), frozenset({3, 4}), frozenset({1, 2, 4})})
```

```
sage: M = Matroid(circuits=[[1,2,3], [1,4,5], [2,3,4,5]])
sage: M.broken_circuits([5,4,3,2,1])
frozenset({frozenset({1, 2}), frozenset({1, 4}), frozenset({2, 3, 4})})
```

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.named_matroids.NonFano()
sage: M.chordality()
5
sage: M = matroids.named_matroids.Fano()
sage: 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

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

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

INPUT:

- `R` – (default: $\mathbf{Z}$) the base ring

EXAMPLES:

```
sage: M = matroids.Wheel(2)
sage: A = M.chow_ring()
sage: 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)
sage: A23*A23
0
```

We construct a more interesting example using the Fano matroid:

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

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

```
sage: G = A.gens()[6:]
sage: Ag, Aabf, Aace, Aadg, Abcd, Abeg, Acfg, Adef = G
sage: Ag * Ag
2*Adef^2
sage: Ag * Abeg
-Adef^2
sage: matrix([[x * y for x in G] for y in G])
[2*Adef^2        0        0  -Adef^2        0  -Adef^2  -Adef^2        0]
[       0   Adef^2        0        0        0        0        0        0]
[       0        0   Adef^2        0        0        0        0        0]
[ -Adef^2        0        0   Adef^2        0        0        0        0]
[       0        0        0        0   Adef^2        0        0        0]
[ -Adef^2        0        0        0        0   Adef^2        0        0]
[ -Adef^2        0        0        0        0        0   Adef^2        0]
[       0        0        0        0        0        0        0   Adef^2]
```

REFERENCES:

- [FY2004]
- [AHK2015]

circuit (*X=None*)

Return a circuit.

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

INPUT:

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

OUTPUT:

Set of elements.

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

EXAMPLES:

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

circuit_closures()

Return the list of closures of circuits of the matroid.

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

OUTPUT:

A dictionary containing the circuit closures of the matroid, indexed by their ranks.

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.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']]
```

circuits()

Return the list of circuits of the matroid.

OUTPUT:

An iterable containing all circuits.

See also:

M.circuit()

EXAMPLES:

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

closure(*X*)

Return the closure of a set X.

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

INPUT:

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

OUTPUT:

Set of elements containing X.

EXAMPLES:

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

cobasis()

Return an arbitrary cobasis of the matroid.

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

Note: Output can change between calls.

OUTPUT:

A set of elements.

See also:

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

EXAMPLES:

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

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

cocircuits()

Return the list of cocircuits of the matroid.

OUTPUT:

An iterable containing all cocircuits.

See also:

M.cocircuit()

EXAMPLES:

```
sage: M = matroids.named_matroids.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']]
```

coclosure(*X*)

Return the coclosure of a set `X`.

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

INPUT:

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

OUTPUT:

A set of elements containing `X`.

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

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

Return a coextension of the matroid.

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

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

INPUT:

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

OUTPUT:

A 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.named_matroids.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.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 ground set. If given, restricts the output to extensions where the new element is contained in all cohyperplanes that contain an element of `subsets`.

OUTPUT:

An iterable containing matroids.

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

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

coflats(*r*)

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

A *coflat* is a coclosed set.

INPUT:

- r – a nonnegative integer.

OUTPUT:

An iterable containing all coflats of corank r.

See also:

M.coclosure()

EXAMPLES:

```
sage: M = matroids.named_matroids.Q6()
sage: sorted([sorted(F) for F in M.coflats(2)])
[['a', 'b'], ['a', 'c'], ['a', 'd', 'f'], ['a', 'e'], ['b', 'c'],
['b', 'd'], ['b', 'e'], ['b', 'f'], ['c', 'd'], ['c', 'e', 'f'],
['d', 'e']]
```

coloops()

Return the set of coloops of the matroid.

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

OUTPUT:

A set of elements.

See also:

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

EXAMPLES:

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

components()

Return a list of the components of the matroid.

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

OUTPUT:

A list of subsets.

See also:

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

EXAMPLES:

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

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

Evaluate the connectivity function of the matroid.

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

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

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

INPUT:

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

OUTPUT:

An integer.

EXAMPLES:

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

contract(*X*)

Contract elements.

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

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

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

INPUT:

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

OUTPUT:

The matroid obtained by contracting the element(s) in `X`.

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.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:

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

Note that one can iterate over strings:

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

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

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

corank (*X=None*)

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

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

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

INPUT:

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

OUTPUT:

Integer.

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

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:

A matroid.

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano().dual().delete('a')
sage: 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.

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

INPUT:

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

OUTPUT:

The matroid obtained by deleting the element(s) in `X`.

See also:

M.contract() M.minor()

EXAMPLES:

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

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

```
sage: M = matroids.CompleteGraphic(4)
sage: M.delete(1) == M.delete([1])
True
sage: M \ 1
Graphic matroid of rank 3 on 5 elements
```

Note that one can iterate over strings:

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

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

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

dependent_r_sets(*r*)

Return the list of dependent subsets of fixed size.

INPUT:

- `r` – a nonnegative integer.

OUTPUT:

An iterable containing all dependent subsets of size `r`.

EXAMPLES:

```
sage: M = matroids.named_matroids.Vamos()
sage: M.dependent_r_sets(3)
[]
sage: sorted([sorted(X) for X in
....: matroids.named_matroids.Vamos().dependent_r_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 `r`

dual()

Return the dual of the matroid.

Let M be a matroid with ground set 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.

OUTPUT:

The dual matroid.

EXAMPLES:

```
sage: M = matroids.named_matroids.Pappus()
sage: N = M.dual()
sage: N.rank()
6
sage: N
Dual of 'Pappus: Matroid of rank 3 on 9 elements with
circuit-closures
{2: {{'a', 'b', 'c'}, {'a', 'f', 'h'}, {'c', 'e', 'g'},
{'b', 'f', 'g'}, {'c', 'd', 'h'}, {'d', 'e', 'f'},
{'a', 'e', 'i'}, {'b', 'd', 'i'}, {'g', 'h', 'i'}},
3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'}}}'
```

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` – A 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.named_matroids.Fano()
sage: M
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
sage: M1 = BasisMatroid(M)
sage: M2 = Matroid(groundset='abcdefg', reduced_matrix=[
....:       [0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 0, 1]], field=GF(2))
sage: M.equals(M1)
True
sage: M.equals(M2)
True
sage: M == M1
False
sage: M == M2
True
```

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

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

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

Return an extension of the matroid.

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

INPUT:

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

OUTPUT:

A matroid.

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

See also:

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

EXAMPLES:

First we add an element in general position:

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

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

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

Putting an element in parallel with another:

```
sage: M = matroids.named_matroids.Fano()
sage: N = M.extension('z', ['c'])
sage: 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 ground set. 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.named_matroids.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
```

f_vector()

Return the f-vector of the matroid.

The *f-vector* is a vector $(f_0, ..., f_r)$, where f_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.named_matroids.BetsyRoss()
sage: M.f_vector()
[1, 11, 20, 1]
```

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

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

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

INPUT:

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

See also:

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

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Fano()
sage: setprint(M.flat_cover())
[{'a', 'b', 'f'}, {'a', 'c', 'e'}, {'a', 'd', 'g'},
 {'b', 'c', 'd'}, {'b', 'e', 'g'}, {'c', 'f', 'g'},
 {'d', 'e', 'f'}]
```

flats(*r*)

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

A *flat* is a closed set.

INPUT:

- `r` – A natural number.

OUTPUT:

An iterable containing all flats of rank `r`.

See also:

M.closure()

EXAMPLES:

```
sage: M = matroids.named_matroids.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']]
```

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.named_matroids.Vamos()
sage: 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.named_matroids.Vamos()
sage: M.full_rank()
4
sage: M.dual().full_rank()
4
```

fundamental_circuit(*B*, *e*)

Return the B-fundamental circuit using e.

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

INPUT:

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

OUTPUT:

A set of elements.

See also:

M.circuit(), M.basis()

EXAMPLES:

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

fundamental_cocircuit(*B*, *e*)

Return the B-fundamental cocircuit using e.

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

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

INPUT:

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

OUTPUT:

A set of elements.

See also:

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

EXAMPLES:

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

groundset()

Return the groundset of the matroid.

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

OUTPUT:

A set.

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

EXAMPLES:

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

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

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

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

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

INPUT:

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

OUTPUT:

Boolean or tuple.

See also:

Matroid.has_minor()

EXAMPLES:

```
sage: M = matroids.named_matroids.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'}))
```

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

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

INPUT:

- `N` – An instance of a `Matroid` object,
- `certificate` – Boolean (Defalt: `False`) If `True`, returns `True, (X, Y, dic)` where ``N`` is isomorphic to `self.minor(X, Y)`, and `dic` is an isomorphism between `N` and `self.minor(X, Y)`.

OUTPUT:

boolean or tuple.

See also:

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

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

EXAMPLES:

```
sage: M = matroids.Whirl(3)
sage: matroids.named_matroids.Fano().has_minor(M)
False
sage: matroids.named_matroids.NonFano().has_minor(M)
True
sage: matroids.named_matroids.NonFano().has_minor(M, certificate=True)
(True, (frozenset(), frozenset({'g'}),
    {0: 'b', 1: 'c', 2: 'a', 3: 'd', 4: 'e', 5: 'f'}))
sage: M = matroids.named_matroids.Fano()
sage: M.has_minor(M, True)
(True,
 (frozenset(),
  frozenset(),
  {'a': 'a', 'b': 'b', 'c': 'c', 'd': 'd', 'e': 'e', 'f': 'f', 'g': 'g'}))
```

hyperplanes()

Return the set of hyperplanes of the matroid.

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

OUTPUT:

An iterable containing all hyperplanes of the matroid.

See also:

M.flats()

EXAMPLES:

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

independence_matroid_polytope()

Return the independence matroid polytope of `self`.

This is defined as the convex hull of the vertices

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

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

See also:

matroid_polytope()

EXAMPLES:

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

sage: M = matroids.named_matroids.NonFano()
sage: M.independence_matroid_polytope()
A 7-dimensional polyhedron in ZZ^7 defined as the convex hull
of 58 vertices
```

REFERENCE:

[DLHK2007]

independent_r_sets(*r*)

Return the list of size-`r` independent subsets of the matroid.

INPUT:

- `r` – a nonnegative integer.

OUTPUT:

An iterable containing all independent subsets of the matroid of cardinality `r`.

EXAMPLES:

```
sage: M = matroids.named_matroids.Pappus()
sage: M.independent_r_sets(4)
[]
sage: sorted(sorted(M.independent_r_sets(3))[0])
['a', 'c', 'e']
```

ALGORITHM:

Test all subsets of the groundset of cardinality `r`

See also:

M.independent_sets() M.bases()

independent_sets()

Return the list of independent subsets of the matroid.

OUTPUT:

An iterable containing all independent subsets of the matroid.

EXAMPLES:

```
sage: M = matroids.named_matroids.Pappus()
sage: I = M.independent_sets()
sage: len(I)
121
```

See also:

M.independent_r_sets()

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

Return a maximum-weight common independent set.

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

INPUT:

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

OUTPUT:

A subset of the groundset.

EXAMPLES:

```
sage: M = matroids.named_matroids.T12()
sage: N = matroids.named_matroids.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.named_matroids.Fano()
sage: N = matroids.Uniform(4, 7)
sage: 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:

A subset of the groundset.

EXAMPLES:

```
sage: M = matroids.named_matroids.T12()
sage: N = matroids.named_matroids.ExtendedTernaryGolayCode()
sage: len(M.intersection_unweighted(N))
```

```
6
sage: M = matroids.named_matroids.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*, *separation=False*)

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`) a 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:

```
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.named_matroids.BetsyRoss().is_3connected()
True
sage: M = matroids.named_matroids.R6()
sage: M.is_3connected()
False
sage: B, X = M.is_3connected(True)
sage: M.connectivity(X)
1
```

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

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

INPUT:

- `certificate` – (default: `False`) a boolean; if `True`, then return `True, None` if the matroid is 4-connected, and `False, X` otherwise, where X is a < 4-separation
- `algorithm` – (default: `None`); specify which algorithm to compute 4-connectivity:
  - `None` – The most appropriate algorithm is chosen automatically.
  - `"intersection"` – an algorithm based on matroid intersection, equivalent to calling `is_kconnected(4,certificate)`.
  - `"shifting"` – an algorithm based on the shifting algorithm [Raj1987].

OUTPUT:

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

See also:

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

EXAMPLES:

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

is_basis(*X*)

Check if a subset is a basis of the matroid.

INPUT:

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

OUTPUT:

Boolean.

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

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:

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

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

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

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

INPUT:

- `k1` – (optional) the integer k_1
- `k2` – (optional) the integer k_2; if not specified, then this method returns if `self` is k_1-chordal
- `certificate` – (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:

```
sage: M = matroids.Uniform(2,4)
sage: [M.is_chordal(i) for i in range(4, 8)]
[True, True, True, True]
sage: M = matroids.named_matroids.NonFano()
sage: [M.is_chordal(i) for i in range(4, 8)]
[False, True, True, True]
sage: M = matroids.named_matroids.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({'a', 'b', 'e', 'f', 'g'}))
```

is_circuit(*X*)

Test if a subset is a circuit of the matroid.

A *circuit* is an inclusionwise minimal dependent subset.

INPUT:

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

OUTPUT:

Boolean.

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

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

Check if the circuit `C` has a chord.

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

INPUT:

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

OUTPUT:

- boolean or tuple

EXAMPLES:

```
sage: M = matroids.named_matroids.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: M.is_circuit_chordal(['a','b','d','e'], certificate=True)
(True, ('c', frozenset({'b', 'c', 'd'}), frozenset({'a', 'c', 'e'})))
```

is_closed(*X*)

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

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

INPUT:

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

OUTPUT:

Boolean.

See also:

M.closure()

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

is_cobasis(*X*)

Check if a subset is a cobasis of the matroid.

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

INPUT:

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

OUTPUT:

Boolean.

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.Vamos()
sage: M.is_cobasis('abc')
False
sage: M.is_cobasis('abce')
```

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

is_cocircuit(*X*)

Test if a subset is a cocircuit of the matroid.

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

INPUT:

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

OUTPUT:

Boolean.

See also:

M.dual(), M.is_circuit()

EXAMPLES:

```
sage: M = matroids.named_matroids.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.named_matroids.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
```

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.named_matroids.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
```

is_coindependent(*X*)

Check if a subset is coindependent in the matroid.

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

INPUT:

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

OUTPUT:

Boolean.

See also:

M.dual(), M.is_independent()

EXAMPLES:

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

is_connected(*certificate=False*)

Test if the matroid is connected.

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

OUTPUT:

Boolean.

See also:

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

EXAMPLES:

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

is_cosimple()

Test if the matroid is cosimple.

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

Dual method of *M.is_simple()*.

OUTPUT:

Boolean.

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano().dual()
sage: M.is_cosimple()
True
sage: N = M \ 'a'
sage: N.is_cosimple()
False
```

is_dependent(*X*)

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

INPUT:

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

OUTPUT:

Boolean.

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

is_independent(*X*)
Check if a subset X is independent in the matroid.

INPUT:

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

OUTPUT:

Boolean.

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

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 – A matroid,
- optional parameter certificate – Boolean.

OUTPUT:

Boolean, and, if certificate = True, a dictionary or None

EXAMPLES:

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


sage: M1 = matroids.named_matroids.Fano()
sage: M2 = matroids.named_matroids.NonFano()
sage: M1.is_isomorphic(M2)
False
sage: 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` – A 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.named_matroids.Pappus()
sage: N = matroids.named_matroids.NonPappus()
sage: N.is_isomorphism(M, {e:e for e in M.groundset()})
False

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

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

```
sage: M = matroids.named_matroids.Fano()
sage: P = PermutationGroup([[('a', 'b', 'c'),
....:                        ('d', 'e', 'f'), ('g')]]).gen()
sage: M.is_isomorphism(M, P)
True

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

There is extensive checking for inappropriate input:

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

sage: M = matroids.CompleteGraphic(4)
sage: sorted(M.groundset())
```

```
[0, 1, 2, 3, 4, 5]
sage: M.is_isomorphism(M, {0: 1, 1: 2, 2: 3})
Traceback (most recent call last):
...
ValueError: domain of morphism does not contain groundset of this
matroid.

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

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

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

is_k_closed(*k*)

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

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

EXAMPLES:

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

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

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

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

INPUT:

- `k` – a integer greater or equal to 1.

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

```
sage: matroids.Uniform(2, 3).is_kconnected(3)
True
sage: M = Matroid(ring=QQ, matrix=[[1, 0, 0, 1, 1, 0],
....:                              [0, 1, 0, 1, 2, 0],
....:                              [0, 0, 1, 0, 0, 1]])
sage: M.is_kconnected(3)
False
sage: N = Matroid(circuit_closures={2: ['abc', 'cdef'],
....:                               3: ['abcdef']},
....:             groundset='abcdef')
sage: N.is_kconnected(3)
False
sage: matroids.named_matroids.BetsyRoss().is_kconnected(3)
True
sage: matroids.AG(5,2).is_kconnected(4)
True
sage: M = matroids.named_matroids.R6()
sage: M.is_kconnected(3)
False
sage: B, X = M.is_kconnected(3,True)
sage: M.connectivity(X)<3
True
```

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

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

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

INPUT:

- `X` – (default: the groundset) a subset (or any iterable) of the groundset
- `weights` – 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 otherwise use the ordering in which `X` lists its elements. Then greedily select elements if they are coindependent of all that was selected before. If an element is not coindependent of the previously selected elements, then we check if it is coindependent with the previously selected elements with higher weight.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.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 [GriRei2014] (Example 7.56 in v3):

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

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

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

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

INPUT:

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

Here is an example from [GriRei2014] (Example 7.56 in v3):

```
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
```

is_simple()

Test if the matroid is simple.

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

OUTPUT:

Boolean.

See also:

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

EXAMPLES:

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

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: Q = RootSystem(['D',4]).root_lattice()
sage: m = matrix([x.to_vector() for x in Q.positive_roots()])
sage: m = m.transpose(); m
[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
```

is_ternary(*randomized_tests=1*)

Decide if `self` is a ternary matroid.

INPUT:

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

OUTPUT:

A Boolean.

ALGORITHM:

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

See also:

M.ternary_matroid()

EXAMPLES:

```
sage: N = matroids.named_matroids.Fano()
sage: N.is_ternary()
False
sage: N = matroids.named_matroids.NonFano()
sage: N.is_ternary()
True
```

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.named_matroids.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` – A matroid.

OUTPUT:

A dictionary, or `None`.

EXAMPLES:

```
sage: M1 = matroids.Wheel(3)
sage: M2 = matroids.CompleteGraphic(4)
sage: morphism=M1.isomorphism(M2)
sage: M1.is_isomorphism(M2, morphism)
True
sage: G3 = graphs.CompleteGraph(4)
```

```
sage: M1.isomorphism(G3)
Traceback (most recent call last):
...
TypeError: can only give isomorphism between matroids.

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

k_closure(*X*, *k*)

Return the k-closure of X.

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

INPUT:

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

EXAMPLES:

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

sage: Q = RootSystem(['D',4]).root_lattice()
sage: m = matrix([x.to_vector() for x in Q.positive_roots()])
sage: m = m.transpose(); m
[1 0 0 0 1 0 0 0 1 1 1 1]
[0 1 0 0 1 1 1 1 1 1 1 2]
[0 0 1 0 0 0 1 1 1 0 1 1]
[0 0 0 1 0 1 0 1 0 1 1 1]
sage: M = Matroid(m)
sage: sorted(M.k_closure({0,2,3,11}, 3))
[0, 2, 3, 11]
sage: sorted(M.k_closure({0,2,3,11}, 4))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
sage: sorted(M.k_closure({0,1}, 4))
[0, 1, 4]
```

lattice_of_flats()

Return the lattice of flats of the matroid.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: M.lattice_of_flats()
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 ground set. If given, restricts the output to linear subclasses such that each hyperplane contains an element of `subsets`.

OUTPUT:

An iterable collection of linear subclasses.

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

See also:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.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:

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

link(S, T)

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

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

Here M denotes this matroid.

INPUT:

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

OUTPUT:

A tuple `(I, X)` containing a frozenset `I` and a frozenset `X`.

ALGORITHM:

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

EXAMPLES:

```
sage: M = matroids.named_matroids.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\J
sage: N.connectivity(S)
2
```

loops()

Return the set of loops of the matroid.

A *loop* is an element u of the groundset such that the one-element set $\{u\}$ is dependent.

OUTPUT:

A set of elements.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: M.loops()
frozenset()
sage: (M / ['a', 'b']).loops()
frozenset({'f'})
```

matroid_polytope()

Return the matroid polytope of `self`.

This is defined as the convex hull of the vertices

$$e_B = \sum_{i \in B} e_i$$

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

See also:

independence_matroid_polytope()

EXAMPLES:

```
sage: M = matroids.Whirl(4)
sage: P = M.matroid_polytope(); P
A 7-dimensional polyhedron in ZZ^8 defined as the convex hull
of 46 vertices
```

```
sage: M = matroids.named_matroids.NonFano()
sage: M.matroid_polytope()
A 6-dimensional polyhedron in ZZ^7 defined as the convex hull
of 29 vertices
```

REFERENCE:

- [DLHK2007]

max_coindependent(*X*)

Compute a maximal coindependent subset of X.

A set is *coindependent* if it is independent in the dual matroid. A set is coindependent if and only if the complement is *spanning* (i.e. contains a basis of the matroid).

INPUT:

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

OUTPUT:

A subset of X.

See also:

M.dual(), M.max_independent()

EXAMPLES:

```
sage: M = matroids.named_matroids.Vamos()
sage: sorted(M.max_coindependent(['a', 'c', 'd', 'e', 'f']))
['a', 'c', 'd', 'e']
sage: M.max_coindependent(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

max_independent(*X*)

Compute a maximal independent subset of X.

INPUT:

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

OUTPUT:

Subset of X.

EXAMPLES:

```
sage: M = matroids.named_matroids.Vamos()
sage: sorted(M.max_independent(['a', 'c', 'd', 'e', 'f']))
['a', 'd', 'e', 'f']
sage: M.max_independent(['x'])
Traceback (most recent call last):
...
ValueError: ['x'] is not a subset of the groundset
```

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

Return a maximum-weight coindependent set contained in X.

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

INPUT:

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

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Fano()
sage: X = M.max_weight_coindependent()
sage: M.is_cobasis(X)
True

sage: wt = {'a': 1, 'b': 2, 'c': 2, 'd': 1/2, 'e': 1, 'f': 2,
....:       'g': 2}
sage: setprint(M.max_weight_coindependent(weights=wt))
{'b', 'c', 'f', 'g'}
sage: def wt(x):
....:   return x
....:
sage: M = matroids.Uniform(2, 8)
sage: setprint(M.max_weight_coindependent(weights=wt))
{2, 3, 4, 5, 6, 7}
sage: setprint(M.max_weight_coindependent())
{0, 1, 2, 3, 4, 5}
sage: M.max_weight_coindependent(X=[], weights={})
frozenset()
```

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

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

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

INPUT:

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

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Fano()
sage: X = M.max_weight_independent()
sage: M.is_basis(X)
```

```
True

sage: wt = {'a': 1, 'b': 2, 'c': 2, 'd': 1/2, 'e': 1,
....:       'f': 2, 'g': 2}
sage: setprint(M.max_weight_independent(weights=wt))
{'b', 'f', 'g'}
sage: def wt(x):
....:   return x
....:
sage: M = matroids.Uniform(2, 8)
sage: setprint(M.max_weight_independent(weights=wt))
{6, 7}
sage: setprint(M.max_weight_independent())
{0, 1}
sage: M.max_weight_coindependent(X=[], weights={})
frozenset()
```

minor(*contractions=None*, *deletions=None*)

Return the minor of `self` obtained by contracting, respectively deleting, the element(s) of `contractions` and `deletions`.

A *minor* of a matroid is a matroid obtained by repeatedly removing elements in one of two ways: either *contract* or *delete* them. It can be shown that the final matroid does not depend on the order in which elements are removed.

INPUT:

- `contractions` – (default: `None`) an element or set of elements to be contracted.
- `deletions` – (default: `None`) an element or set of elements to be deleted.

OUTPUT:

A 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
```

The sets of contractions and deletions need not be independent, respectively coindependent:

```
sage: M = matroids.named_matroids.Fano()
sage: M.rank('abf')
2
sage: M.minor(contractions='abf')
Binary matroid of rank 1 on 4 elements, type (1, 0)
```

However, they need to be subsets of the groundset, and disjoint:

```
sage: M = matroids.named_matroids.Vamos()
sage: M.minor('abc', 'defg')
M / {'a', 'b', 'c'} \ {'d', 'e', 'f', 'g'}, where M is Vamos:
Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'},
{'e', 'f', 'g', 'h'}, {'a', 'b', 'g', 'h'}, {'c', 'd', 'e', 'f'}},
4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}

sage: M.minor('defgh', 'abc')
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'},
{'e', 'f', 'g', 'h'}, {'a', 'b', 'g', 'h'}, {'c', 'd', 'e', 'f'}},
4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}

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.
```

Warning: There can be ambiguity if elements of the groundset are themselves iterable, and their elements are in the groundset. The main example of this is when an element is a string. See the documentation of the methods *contract()* and *delete()* for an example of this.

modular_cut (*subsets*)

Compute the modular cut generated by `subsets`.

A *modular cut* is a collection C of flats such that

- If $F \in C$ and F' is a flat containing F, then $F' \in C$
- If $F_1, F_2 \in C$ form a modular pair of flats, then $F_1 \cap F_2 \in C$.

A *flat* is a closed set, a *modular pair* is a pair F_1, F_2 of flats with $r(F_1)+r(F_2) = r(F_1 \cup F_2)+r(F_1 \cap F_2)$, where r is the rank function of the matroid.

The modular cut *generated by* `subsets` is the smallest modular cut C for which closure‘(S) in C‘ for all S in `subsets`.

There is a one-to-one correspondence between the modular cuts of a matroid and the single-element extensions of the matroid. See [Oxl2011] Section 7.2 for more information.

Note: Sage uses linear subclasses, rather than modular cuts, internally for matroid extension. A linear subclass is the set of hyperplanes (flats of rank $r(M) - 1$) of a modular cut. It determines the modular cut uniquely (see [Oxl2011] Section 7.2).

INPUT:

- `subsets` – A collection of subsets of the groundset.

OUTPUT:

A collection of subsets.

See also:

M.flats(),M.linear_subclasses(),M.extension()

EXAMPLES:

Any extension of the Vamos matroid where the new point is placed on the lines through elements $\{a, b\}$ and through $\{c, d\}$ is an extension by a loop:

```
sage: M = matroids.named_matroids.Vamos()
sage: frozenset([]) in M.modular_cut(['ab', 'cd'])
True
```

In any extension of the matroid $S_8 \setminus h$, a point on the lines through $\{c, g\}$ and $\{a, e\}$ also is on the line through $\{b, f\}$:

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

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

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

no_broken_circuits_sets(*ordering=None*)

Return the no broken circuits (NBC) sets of `self`.

An NBC set is a subset A of the ground set under some total ordering $<$ such that A contains no broken circuit.

INPUT:

- `ordering` – a total ordering of the groundset given as a list

EXAMPLES:

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

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

nonbases()

Return the list of nonbases of the matroid.

A *nonbasis* is a set with cardinality `self.full_rank()` that is not a basis.

OUTPUT:

An iterable containing the nonbases of the matroid.

See also:

M.basis()

EXAMPLES:

```
sage: M = matroids.Uniform(2, 4)
sage: list(M.nonbases())
[]
sage: [sorted(X) for X in matroids.named_matroids.P6().nonbases()]
[['a', 'b', 'c']]
```

ALGORITHM:

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

noncospanning_cocircuits()

Return the list of noncospanning cocircuits of the matroid.

A *noncospanning cocircuit* is a cocircuit whose corank is strictly smaller than the corank of the matroid.

OUTPUT:

An iterable containing all nonspanning circuits.

See also:

M.cocircuit(), *M.corank()*

EXAMPLES:

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

nonspanning_circuit_closures()

Return the list of closures of nonspanning circuits of the matroid.

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

OUTPUT:

A dictionary containing the nonspanning circuit closures of the matroid, indexed by their ranks.

See also:

M.nonspanning_circuits(), *M.closure()*

EXAMPLES:

```
sage: M = matroids.named_matroids.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 list of nonspanning circuits of the matroid.

A *nonspanning circuit* is a circuit whose rank is strictly smaller than the rank of the matroid.

OUTPUT:

An iterable containing all nonspanning circuits.

See also:

M.circuit(), *M.rank()*

EXAMPLES:

```
sage: M = matroids.named_matroids.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']]
```

orlik_solomon_algebra(*R*, *ordering=None*)

Return the Orlik-Solomon algebra of `self`.

INPUT:

- `R` – the base ring
- `ordering` – (optional) an ordering of the ground set

See also:

`OrlikSolomonAlgebra`

EXAMPLES:

```
sage: M = matroids.Uniform(3, 4)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS
Orlik-Solomon algebra of U(3, 4): Matroid of rank 3 on 4 elements
 with circuit-closures
 {3: {{0, 1, 2, 3}}}
```

partition()

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

OUTPUT:

A list of disjoint independent sets that covers the goundset.

EXAMPLES:

```
sage: M = matroids.named_matroids.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 ground set 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 ground set elements to their (x,y) positions.
- `save_pos`: A boolean indicating that point placements (either internal or user provided) and line orders (if provided) will be cached in the matroid (`M._cached_info`) and can be used for reproducing the geometric representation during the same session

OUTPUT:

A sage graphics object of type <class 'sage.plot.graphics.Graphics'> that corresponds to the geometric representation of the matroid

EXAMPLES:

```
sage: M=matroids.named_matroids.Fano()
sage: G=M.plot()
sage: type(G)
<class 'sage.plot.graphics.Graphics'>
sage: G.show()
```

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.named_matroids.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
```

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 ground set 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 ground set elements to their (x,y) positions.
- `save_pos` – A 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:

```
sage: M=matroids.named_matroids.TernaryDowling3()
sage: M.show(B=['a','b','c'])
sage: M.show(B=['a','b','c'],lineorders=[['f','e','i']])
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])
```

simplify()
Return the simplification of the matroid.

A matroid is *simple* if it contains no circuits of length 1 or 2. The *simplification* of a matroid is obtained by deleting all loops (circuits of length 1) and deleting all but one element from each parallel class (a closed set of rank 1, that is, each pair in it forms a circuit of length 2).

OUTPUT:

A matroid.

See also:

`M.is_simple()`, *`M.loops()`*, *`M.circuit()`*, *`M.cosimplify()`*

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano().contract('a')
sage: M.size() - M.simplify().size()
3
```

size()

Return the size of the groundset.

OUTPUT:

Integer.

EXAMPLES:

```
sage: M = matroids.named_matroids.Vamos()
sage: M.size()
8
```

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

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

INPUT:

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

OUTPUT:

Either a *TernaryMatroid*, or `None`

ALGORITHM:

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

See also:

`M._local_ternary_matroid()`

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: M.ternary_matroid() is None
True
sage: N = matroids.named_matroids.NonFano()
sage: N.ternary_matroid()
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:

A matroid.

See also:

M.extension(), *M.contract()*

EXAMPLES:

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

tutte_polynomial (*x=None*, *y=None*)

Return the Tutte polynomial of the matroid.

The *Tutte polynomial* of a matroid is the polynomial

$$T(x,y) = \sum_{A \subseteq E} (x-1)^{r(E)-r(A)}(y-1)^{r^*(E)-r^*(E \setminus A)},$$

where E is the groundset of the matroid, r is the rank function, and r^* is the corank function. Tutte defined his polynomial differently:

$$T(x,y) = \sum_{B} x^i(B)y^e(B),$$

where the sum ranges over all bases of the matroid, $i(B)$ is the number of internally active elements of B, and $e(B)$ is the number of externally active elements of B.

INPUT:

- `x` – (optional) a variable or numerical argument.
- `y` – (optional) a variable or numerical argument.

OUTPUT:

The Tutte-polynomial $T(x,y)$, where x and y are substituted with any values provided as input.

Todo: Make implementation more efficient, e.g. generalizing the approach from trac ticket #1314 from graphs to matroids.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: M.tutte_polynomial()
y^4 + x^3 + 3*y^3 + 4*x^2 + 7*x*y + 6*y^2 + 3*x + 3*y
sage: M.tutte_polynomial(1, 1) == M.bases_count()
True
```

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 ground set E_i. The *matroid union* M of $(M_1, M_2, \ldots, M_k)$ has ground set $E = \cup E_i$. Moreover, a set $I \subseteq E$ is independent in M if and only if the restriction of I to E_i is independent in M_i for every i.

INPUT:

- `matroids` - a matroid or a list of matroids

OUTPUT:

An instance of MatroidUnion.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: N = M.union(matroids.named_matroids.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-
```

CHAPTER

TWO

BUILT-IN FAMILIES AND INDIVIDUAL MATROIDS

2.1 Catalog of matroids

A module containing constructors for several common matroids.

A list of all matroids in this module is available via tab completion. Let <tab> indicate pressing the tab key. So begin by typing `matroids.<tab>` to see the various constructions available. Many special matroids can be accessed from the submenu `matroids.named_matroids.<tab>`.

To create a custom matroid using a variety of inputs, see the function *Matroid()*.

- **Parametrized matroid constructors**
  - *matroids.AG*
  - *matroids.CompleteGraphic*
  - *matroids.PG*
  - *matroids.Uniform*
  - *matroids.Wheel*
  - *matroids.Whirl*
- **Named matroids (`matroids.named_matroids.<tab>`)**
  - *matroids.named_matroids.AG23minus*
  - *matroids.named_matroids.AG32prime*
  - *matroids.named_matroids.BetsyRoss*
  - *matroids.named_matroids.Block_9_4*
  - *matroids.named_matroids.Block_10_5*
  - *matroids.named_matroids.D16*
  - *matroids.named_matroids.ExtendedBinaryGolayCode*
  - *matroids.named_matroids.ExtendedTernaryGolayCode*
  - *matroids.named_matroids.F8*
  - *matroids.named_matroids.Fano*
  - *matroids.named_matroids.J*
  - *matroids.named_matroids.K33dual*
  - *matroids.named_matroids.L8*

- *matroids.named_matroids.N1*
- *matroids.named_matroids.N2*
- *matroids.named_matroids.NonFano*
- *matroids.named_matroids.NonPappus*
- *matroids.named_matroids.NonVamos*
- *matroids.named_matroids.NotP8*
- *matroids.named_matroids.O7*
- *matroids.named_matroids.P6*
- *matroids.named_matroids.P7*
- *matroids.named_matroids.P8*
- *matroids.named_matroids.P8pp*
- *matroids.named_matroids.P9*
- *matroids.named_matroids.Pappus*
- *matroids.named_matroids.Q6*
- *matroids.named_matroids.Q8*
- *matroids.named_matroids.Q10*
- *matroids.named_matroids.R6*
- *matroids.named_matroids.R8*
- *matroids.named_matroids.R9A*
- *matroids.named_matroids.R9B*
- *matroids.named_matroids.R10*
- *matroids.named_matroids.R12*
- *matroids.named_matroids.S8*
- *matroids.named_matroids.T8*
- *matroids.named_matroids.T12*
- *matroids.named_matroids.TernaryDowling3*
- *matroids.named_matroids.Terrahawk*
- *matroids.named_matroids.TicTacToe*
- *matroids.named_matroids.Vamos*

2.2 Documentation for the matroids in the catalog

This module contains implementations for many of the functions accessible through *matroids.* and *matroids.named_matroids.* (type those lines in Sage and hit `tab` for a list).

The docstrings include educational information about each named matroid with the hopes that this class can be used as a reference. However, for a more comprehensive list of properties we refer to the appendix of [Oxl2011].

Todo: Add optional argument `groundset` to each method so users can customize the groundset of the matroid. We probably want some means of relabeling to accomplish that.

Add option to specify the field for represented matroids.

AUTHORS:

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

2.2.1 Functions

sage.matroids.catalog.**AG**(*n*, *q*, *x=None*)

Return the affine 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` – (default: `None`) a string. The name of the generator of a non-prime field, used for non-prime fields. If not supplied, `'x'` is used.

OUTPUT:

A linear matroid whose elements are the points of $AG(n, q)$.

The affine geometry can be obtained from the projective geometry by removing a hyperplane.

EXAMPLES:

```
sage: M = matroids.AG(2, 3) \ 8
sage: M.is_isomorphic(matroids.named_matroids.AG23minus())
True
sage: matroids.AG(5, 4, 'z').size() == ((4 ^ 6 - 1) / (4 - 1) -
....:                                   (4 ^ 5 - 1)/(4 - 1))
True
sage: M = matroids.AG(4, 2); M
AG(4, 2): Binary matroid of rank 5 on 16 elements, type (5, 0)
```

sage.matroids.catalog.**AG23minus**()

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. See [Oxl2011], p. 653.

EXAMPLES:

```
sage: M = matroids.named_matroids.AG23minus()
sage: M.is_valid()
True
```

sage.matroids.catalog.**AG32prime**()

Return the matroid $AG(3, 2)'$, represented as circuit closures.

The matroid $AG(3, 2)'$ is a 8-element matroid of rank-4. It is a smallest non-representable matroid. It is the unique relaxation of $AG(3, 2)$. See [Oxl2011], p. 646.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.AG32prime(); M
AG(3, 2)': Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'c', 'd', 'e', 'h'}, {'b', 'e', 'g', 'h'}, {'d', 'e', 'f', 'g'},
     {'a', 'b', 'd', 'e'}, {'b', 'c', 'd', 'g'}, {'c', 'f', 'g', 'h'},
     {'a', 'c', 'd', 'f'}, {'b', 'c', 'e', 'f'}, {'a', 'c', 'e', 'g'},
     {'a', 'b', 'f', 'g'}, {'a', 'b', 'c', 'h'}, {'a', 'e', 'f', 'h'},
     {'a', 'd', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: M.contract('c').is_isomorphic(matroids.named_matroids.Fano())
True
sage: setprint(M.noncospanning_cocircuits())
[{'b', 'd', 'f', 'h'}, {'a', 'd', 'g', 'h'}, {'c', 'd', 'e', 'h'},
 {'a', 'c', 'd', 'f'}, {'b', 'c', 'd', 'g'}, {'a', 'b', 'd', 'e'},
 {'d', 'e', 'f', 'g'}, {'c', 'f', 'g', 'h'}, {'b', 'c', 'e', 'f'},
 {'a', 'b', 'f', 'g'}, {'a', 'b', 'c', 'h'}, {'a', 'e', 'f', 'h'},
 {'b', 'e', 'g', 'h'}]
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**BetsyRoss**()

Return the Betsy Ross matroid, represented by circuit closures.

An extremal golden-mean matroid. That is, if M is simple, rank 3, has the Betsy Ross matroid as a restriction and is a Golden Mean matroid, then M is the Betsy Ross matroid.

EXAMPLES:

```
sage: M = matroids.named_matroids.BetsyRoss()
sage: len(M.circuit_closures()[2])
10
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**Block_10_5**()

Return the paving matroid whose non-spanning circuits form the blocks of a $3-(10,5,3)$ design.

EXAMPLES:

```
sage: M = matroids.named_matroids.Block_10_5()
sage: M.is_valid() # long time
True
sage: BD = BlockDesign(M.groundset(), M.nonspanning_circuits())
sage: BD.is_t_design(return_parameters=True)
(True, (3, 10, 5, 3))
```

sage.matroids.catalog.**Block_9_4**()

Return the paving matroid whose non-spanning circuits form the blocks of a $2-(9,4,3)$ design.

EXAMPLES:

```
sage: M = matroids.named_matroids.Block_9_4()
sage: M.is_valid() # long time
True
sage: BD = BlockDesign(M.groundset(), M.nonspanning_circuits())
sage: BD.is_t_design(return_parameters=True)
(True, (2, 9, 4, 3))
```

sage.matroids.catalog.**CompleteGraphic**(n)
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:

```
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
```

sage.matroids.catalog.**D16**()
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}. See [CMO2012].

EXAMPLES:

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

sage.matroids.catalog.**ExtendedBinaryGolayCode**()
Return the matroid of the extended binary Golay code.

See GolayCode documentation for more on this code.

EXAMPLES:

```
sage: M = matroids.named_matroids.ExtendedBinaryGolayCode()
sage: C = LinearCode(M.representation())
sage: C.is_permutation_equivalent(codes.GolayCode(GF(2))) # long time
True
sage: M.is_valid()
True
```

sage.matroids.catalog.**ExtendedTernaryGolayCode**()
Return the matroid of the extended ternary Golay code.

See GolayCode

EXAMPLES:

```
sage: M = matroids.named_matroids.ExtendedTernaryGolayCode()
sage: C = LinearCode(M.representation())
sage: C.is_permutation_equivalent(codes.GolayCode(GF(3))) # long time
True
sage: M.is_valid()
True
```

sage.matroids.catalog.**F8**()

Return the matroid F_8, represented as circuit closures.

The matroid F_8 is a 8-element matroid of rank-4. It is a smallest non-representable matroid. See [Oxl2011], p. 647.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.named_matroids.F8(); M
F8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'c', 'd', 'e', 'h'}, {'d', 'e', 'f', 'g'}, {'a', 'b', 'd', 'e'},
     {'b', 'c', 'd', 'g'}, {'c', 'f', 'g', 'h'}, {'a', 'c', 'd', 'f'},
     {'b', 'c', 'e', 'f'}, {'a', 'c', 'e', 'g'}, {'a', 'b', 'f', 'g'},
     {'a', 'b', 'c', 'h'}, {'a', 'e', 'f', 'h'}, {'a', 'd', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: D = get_nonisomorphic_matroids([M.contract(i)
....:                                 for i in M.groundset()])
sage: len(D)
3
sage: [N.is_isomorphic(matroids.named_matroids.Fano()) for N in D]
[...True...]
sage: [N.is_isomorphic(matroids.named_matroids.NonFano()) for N in D]
[...True...]
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**Fano**()

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. It is also the projective plane of order two, i.e. $\mathrm{PG}(2,2)$. See [Oxl2011], p. 643.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Fano(); M
Fano: Binary matroid of rank 3 on 7 elements, type (3, 0)
sage: setprint(sorted(M.nonspanning_circuits()))
[{'b', 'c', 'd'}, {'a', 'c', 'e'}, {'d', 'e', 'f'}, {'a', 'b', 'f'},
 {'c', 'f', 'g'}, {'b', 'e', 'g'}, {'a', 'd', 'g'}]
sage: M.delete(M.groundset_list()[randrange(0,
....:                  7)]).is_isomorphic(matroids.CompleteGraphic(4))
True
```

sage.matroids.catalog.**J**()

Return the matroid J, represented over $GF(3)$.

The matroid J is a 8-element matroid of rank-4. It is representable over a field if and only if that field has at least three elements. See [Oxl2011], p. 650.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.J(); M
J: Ternary matroid of rank 4 on 8 elements, type 0-
sage: setprint(M.truncation().nonbases())
[{'a', 'c', 'g'}, {'a', 'b', 'f'}, {'a', 'd', 'h'}]
sage: M.is_isomorphic(M.dual())
True
sage: M.has_minor(matroids.CompleteGraphic(4))
False
sage: M.is_valid()
True
```

sage.matroids.catalog.**K33dual**()

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

The matroid $M * (K_{3,3})$ is a 9-element matroid of rank-4. It is an excluded minor for the class of graphic matroids. It is the graft matroid of the 4-wheel with every vertex except the hub being coloured. See [Oxl2011], p. 652.

EXAMPLES:

```
sage: M = matroids.named_matroids.K33dual(); M
M*(K3, 3): Regular matroid of rank 4 on 9 elements with 81 bases
sage: any([N.is_3connected()
....:                      for N in M.linear_extensions(simple=True)])
False
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**L8**()

Return the matroid L_8, represented as circuit closures.

The matroid L_8 is a 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. See [Oxl2011], p. 648.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.L8(); M
L8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'b', 'd', 'f', 'h'}, {'c', 'd', 'e', 'h'}, {'d', 'e', 'f', 'g'},
     {'b', 'c', 'd', 'g'}, {'a', 'c', 'e', 'g'}, {'a', 'b', 'f', 'g'},
     {'a', 'b', 'c', 'h'}, {'a', 'e', 'f', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: M.equals(M.dual())
True
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**N1**()

Return the matroid N_1, represented over $\mathbf{F}_3$.

N_1 is an excluded minor for the dyadic matroids. See [Oxl2011], p. 554.

EXAMPLES:

```
sage: M = matroids.named_matroids.N1()
sage: M.is_field_isomorphic(M.dual())
True
```

```
sage: M.is_valid()
True
```

sage.matroids.catalog.**N2**()

Return the matroid N_2, represented over $\mathbf{F}_3$.

N_2 is an excluded minor for the dyadic matroids. See [Oxl2011], p. 554.

EXAMPLES:

```
sage: M = matroids.named_matroids.N2()
sage: M.is_field_isomorphic(M.dual())
True
sage: M.is_valid()
True
```

sage.matroids.catalog.**NonFano**()

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. See [Oxl2011], p. 643.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.NonFano(); M
NonFano: Ternary matroid of rank 3 on 7 elements, type 0-
sage: setprint(M.nonbases())
[{'b', 'c', 'd'}, {'a', 'c', 'e'}, {'a', 'b', 'f'}, {'c', 'f', 'g'},
 {'b', 'e', 'g'}, {'a', 'd', 'g'}]
sage: M.delete('f').is_isomorphic(matroids.CompleteGraphic(4))
True
sage: M.delete('g').is_isomorphic(matroids.CompleteGraphic(4))
False
```

sage.matroids.catalog.**NonPappus**()

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. See [Oxl2011], p. 655.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.NonPappus(); M
NonPappus: Matroid of rank 3 on 9 elements with circuit-closures
{2: {{'a', 'b', 'c'}, {'a', 'f', 'h'}, {'c', 'e', 'g'},
     {'b', 'f', 'g'}, {'c', 'd', 'h'}, {'b', 'd', 'i'},
     {'a', 'e', 'i'}, {'g', 'h', 'i'}},
 3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'}}}
sage: setprint(M.nonspanning_circuits())
[{'a', 'b', 'c'}, {'a', 'f', 'h'}, {'c', 'e', 'g'}, {'b', 'f', 'g'},
 {'c', 'd', 'h'}, {'b', 'd', 'i'}, {'a', 'e', 'i'}, {'g', 'h', 'i'}]
sage: M.is_dependent(['d', 'e', 'f'])
False
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**NonVamos**()

Return the non-Vamos matroid.

The non-Vamos matroid, or V_8^+ is an 8-element matroid of rank 4. It is a tightening of the Vamos matroid. It is representable over some field. See [Oxl2011], p. 72, 84.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.NonVamos(); M
NonVamos: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'g', 'h'}, {'a', 'b', 'c', 'd'}, {'e', 'f', 'g', 'h'},
    {'c', 'd', 'e', 'f'}, {'a', 'b', 'e', 'f'}, {'c', 'd', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: setprint(M.nonbases())
[{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
 {'c', 'd', 'e', 'f'}, {'c', 'd', 'g', 'h'}, {'e', 'f', 'g', 'h'}]
sage: M.is_dependent(['c', 'd', 'g', 'h'])
True
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**NotP8**()

Return the matroid `NotP8`.

This is a matroid that is not P_8, found on page 512 of [Oxl1992] (the first edition).

EXAMPLES:

```
sage: M = matroids.named_matroids.P8()
sage: N = matroids.named_matroids.NotP8()
sage: M.is_isomorphic(N)
False
sage: M.is_valid()
True
```

sage.matroids.catalog.**O7**()

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 $M(K_4)$. See [Oxl2011], p. 644

EXAMPLES:

```
sage: M = matroids.named_matroids.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
```

sage.matroids.catalog.**P6**()

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. See [Oxl2011], p. 641.

EXAMPLES:

```
sage: M = matroids.named_matroids.P6(); M
P6: Matroid of rank 3 on 6 elements with circuit-closures
{2: {{'a', 'b', 'c'}}, 3: {{'a', 'b', 'c', 'd', 'e', 'f'}}}
```

```
sage: len(set(M.nonspanning_circuits()).difference(M.nonbases())) == 0
True
sage: Matroid(matrix=random_matrix(GF(4, 'a'), ncols=5,
....:                                          nrows=5)).has_minor(M)
False
sage: M.is_valid()
True
```

sage.matroids.catalog.**P7**()

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 3 elements. It is one of two ternary 3-spikes, with the other being F_7^-. See [Oxl2011], p. 644.

EXAMPLES:

```
sage: M = matroids.named_matroids.P7(); M
P7: Ternary matroid of rank 3 on 7 elements, type 1+
sage: M.f_vector()
[1, 7, 11, 1]
sage: M.has_minor(matroids.CompleteGraphic(4))
False
sage: M.is_valid()
True
```

sage.matroids.catalog.**P8**()

Return the matroid P_8, represented over $GF(3)$.

The matroid P_8 is a 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. See [Oxl2011], p. 650.

EXAMPLES:

```
sage: M = matroids.named_matroids.P8(); M
P8: Ternary matroid of rank 4 on 8 elements, type 2+
sage: M.is_isomorphic(M.dual())
True
sage: Matroid(matrix=random_matrix(GF(4, 'a'), ncols=5,
....:                              nrows=5)).has_minor(M)
False
sage: M.bicycle_dimension()
2
```

sage.matroids.catalog.**P8pp**()

Return the matroid $P_8^=$, represented as circuit closures.

The matroid $P_8^=$ is a 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. See [Oxl2011], p. 651.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.named_matroids.P8pp(); M
P8'': Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'c', 'g', 'h'}, {'a', 'b', 'f', 'h'}, {'b', 'c', 'e', 'g'},
     {'a', 'd', 'e', 'g'}, {'c', 'd', 'f', 'h'}, {'b', 'd', 'f', 'g'},
     {'a', 'c', 'e', 'f'}, {'b', 'd', 'e', 'h'}},
```

```
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: M.is_isomorphic(M.dual())
True
sage: len(get_nonisomorphic_matroids([M.contract(i)
....:                                 for i in M.groundset()]))
1
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**P9**()

Return the matroid P_9.

This is the matroid referred to as P_9 by Oxley in his paper "The binary matroids with no 4-wheel minor"

EXAMPLES:

```
sage: M = matroids.named_matroids.P9()
sage: M
P9: Binary matroid of rank 4 on 9 elements, type (1, 1)
sage: M.is_valid()
True
```

sage.matroids.catalog.**PG**(*n*, *q*, *x=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` – (default: `None`) a string. The name of the generator of a non-prime field, used for non-prime fields. If not supplied, `'x'` is used.

OUTPUT:

A linear matroid whose elements are the points of $PG(n, q)$.

EXAMPLES:

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

sage.matroids.catalog.**Pappus**()

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. See [Oxl2011], p. 655.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Pappus(); M
```

```
Pappus: Matroid of rank 3 on 9 elements with circuit-closures
{2: {{'a', 'b', 'c'}, {'a', 'f', 'h'}, {'c', 'e', 'g'},
     {'b', 'f', 'g'}, {'c', 'd', 'h'}, {'d', 'e', 'f'},
     {'a', 'e', 'i'}, {'b', 'd', 'i'}, {'g', 'h', 'i'}},
 3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'}}}
sage: setprint(M.nonspanning_circuits())
[{'a', 'b', 'c'}, {'a', 'f', 'h'}, {'c', 'e', 'g'}, {'b', 'f', 'g'},
 {'c', 'd', 'h'}, {'b', 'd', 'i'}, {'a', 'e', 'i'}, {'d', 'e', 'f'},
 {'g', 'h', 'i'}]
sage: M.is_dependent(['d', 'e', 'f'])
True
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**Q10**()

Return the matroid Q_{10}, represented over $\mathbf{F}_4$.

Q_{10} is a 10-element, rank-5, self-dual matroid. It is representable over $\mathbf{F}_3$ and $\mathbf{F}_4$, and hence is a sixth-roots-of-unity matroid. Q_{10} is a splitter for the class of sixth-root-of-unity matroids.

EXAMPLES:

```
sage: M = matroids.named_matroids.Q10()
sage: M.is_isomorphic(M.dual())
True
sage: M.is_valid()
True
```

Check the splitter property. By Seymour's Theorem, and using self-duality, we only need to check that all 3-connected single-element extensions have an excluded minor for sixth-roots-of-unity. The only excluded minors that are quaternary are $U_{2,5}, U_{3,5}, F_7, F_7^*$. As it happens, it suffices to check for $U_{2,5}$:

sage: S = matroids.named_matroids.Q10().linear_extensions(simple=True) sage: [M for M in S if not M.has_line_minor(5)] # long time []

sage.matroids.catalog.**Q6**()

Return the matroid Q_6, represented over $GF(4)$.

The matroid Q_6 is a 6-element matroid of rank-3. It is representable over a field if and only if that field has at least four elements. It is the unique relaxation of the rank-3 whirl. See [Oxl2011], p. 641.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Q6(); M
Q6: Quaternary matroid of rank 3 on 6 elements
sage: setprint(M.hyperplanes())
[{'a', 'b', 'd'}, {'a', 'c'}, {'a', 'e'}, {'a', 'f'}, {'b', 'c', 'e'},
 {'b', 'f'}, {'c', 'd'}, {'c', 'f'}, {'d', 'e'}, {'d', 'f'},
 {'e', 'f'}]
sage: M.nonspanning_circuits() == M.noncospanning_cocircuits()
False
```

sage.matroids.catalog.**Q8**()

Return the matroid Q_8, represented as circuit closures.

The matroid Q_8 is a 8-element matroid of rank-4. It is a smallest non-representable matroid. See [Oxl2011], p. 647.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Q8(); M
Q8: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'c', 'd', 'e', 'h'}, {'d', 'e', 'f', 'g'}, {'a', 'b', 'd', 'e'},
     {'b', 'c', 'd', 'g'}, {'c', 'f', 'g', 'h'}, {'a', 'c', 'd', 'f'},
     {'b', 'c', 'e', 'f'}, {'a', 'b', 'f', 'g'}, {'a', 'b', 'c', 'h'},
     {'a', 'e', 'f', 'h'}, {'a', 'd', 'g', 'h'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: setprint(M.flats(3))
[{'a', 'b', 'c', 'h'}, {'a', 'b', 'd', 'e'}, {'a', 'b', 'f', 'g'},
 {'a', 'c', 'd', 'f'}, {'a', 'c', 'e'}, {'a', 'c', 'g'},
 {'a', 'd', 'g', 'h'}, {'a', 'e', 'f', 'h'}, {'a', 'e', 'g'},
 {'b', 'c', 'd', 'g'}, {'b', 'c', 'e', 'f'}, {'b', 'd', 'f'},
 {'b', 'd', 'h'}, {'b', 'e', 'g'}, {'b', 'e', 'h'}, {'b', 'f', 'h'},
 {'b', 'g', 'h'}, {'c', 'd', 'e', 'h'}, {'c', 'e', 'g'},
 {'c', 'f', 'g', 'h'}, {'d', 'e', 'f', 'g'}, {'d', 'f', 'h'},
 {'e', 'g', 'h'}]
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**R10**()

Return the matroid R_{10}, represented over the regular partial field.

The matroid R_{10} is a 10-element regular 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. See [Oxl2011], p. 656.

EXAMPLES:

```
sage: M = matroids.named_matroids.R10(); M
R10: Regular matroid of rank 5 on 10 elements with 162 bases
sage: cct = []
sage: for i in M.circuits():
....:     cct.append(len(i))
....:
sage: Set(cct)
{4, 6}
sage: M.equals(M.dual())
False
sage: M.is_isomorphic(M.dual())
True
sage: M.is_valid()
True
```

Check the splitter property:

```
sage: matroids.named_matroids.R10().linear_extensions(simple=True)
[]
```

sage.matroids.catalog.**R12**()

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. See [Oxl2011], p. 657.

EXAMPLES:

```
sage: M = matroids.named_matroids.R12(); M
R12: Regular matroid of rank 6 on 12 elements with 441 bases
```

```
sage: M.equals(M.dual())
False
sage: M.is_isomorphic(M.dual())
True
sage: M.is_valid()
True
```

sage.matroids.catalog.**R6**()
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 $U_{2,4}$. See [Oxl2011], p. 642.

EXAMPLES:

```
sage: M = matroids.named_matroids.R6(); M
R6: Ternary matroid of rank 3 on 6 elements, type 2+
sage: M.equals(M.dual())
True
sage: M.is_connected()
True
sage: M.is_3connected()
False
```

sage.matroids.catalog.**R8**()
Return the matroid R_8, represented over $GF(3)$.

The matroid R_8 is a 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. See [Oxl2011], p. 646.

EXAMPLES:

```
sage: M = matroids.named_matroids.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.named_matroids.NonFano())
True
sage: M.equals(M.dual())
True
sage: M.has_minor(matroids.named_matroids.Fano())
False
```

sage.matroids.catalog.**R9A**()
Return the matroid R_9^A.

The matroid R_9^A is not representable over any field, yet none of the cross-ratios in its Tuttegroup equal 1. It is one of the 4 matroids on at most 9 elements with this property, the others being $R_9^{A^*}$, R_9^B and $R_9^{B^*}$.

EXAMPLES:

```
sage: M = matroids.named_matroids.R9A()
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**R9B**()
Return the matroid R_9^B.

The matroid R_9^B is not representable over any field, yet none of the cross-ratios in its Tuttegroup equal 1. It is one of the 4 matroids on at most 9 elements with this property, the others being $R_9^{B^*}$, R_9^A and $R_9^{A^*}$.

EXAMPLES:

```
sage: M = matroids.named_matroids.R9B()
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**S8**()

Return the matroid S_8, represented over $GF(2)$.

The matroid S_8 is a 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. See [Oxl2011], p. 648.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.named_matroids.S8(); M
S8: Binary matroid of rank 4 on 8 elements, type (2, 0)
sage: M.contract('d').is_isomorphic(matroids.named_matroids.Fano())
True
sage: M.delete('d').is_isomorphic(
....:                           matroids.named_matroids.Fano().dual())
False
sage: M.is_graphic()
False
sage: D = get_nonisomorphic_matroids(
....:       list(matroids.named_matroids.Fano().linear_coextensions(
....:                                                 cosimple=True)))
sage: len(D)
2
sage: [N.is_isomorphic(M) for N in D]
[...True...]
```

sage.matroids.catalog.**T12**()

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. See [Oxl2011], p. 658.

EXAMPLES:

```
sage: M = matroids.named_matroids.T12()
sage: M
T12: Binary matroid of rank 6 on 12 elements, type (2, None)
sage: M.is_valid()
True
```

sage.matroids.catalog.**T8**()

Return the matroid T_8, represented over $GF(3)$.

The matroid T_8 is a 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. See [Oxl2011], p. 649.

EXAMPLES:

```
sage: M = matroids.named_matroids.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.named_matroids.P7())
```

```
True
sage: M.has_minor(matroids.Uniform(3, 8))
False
```

sage.matroids.catalog.**TernaryDowling3**()
Return the matroid $Q_3(GF(3)^imes)$, represented over $GF(3)$.

The matroid $Q_3(GF(3)^imes)$ 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. See [Oxl2011], p. 654.

EXAMPLES:

```
sage: M = matroids.named_matroids.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.matroids.catalog.**Terrahawk**()
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. See [CMO2011].

EXAMPLES:

```
sage: M = matroids.named_matroids.Terrahawk()
sage: M
Terrahawk: Binary matroid of rank 8 on 16 elements, type (0, 4)
sage: M.is_valid()
True
```

sage.matroids.catalog.**TicTacToe**()
Return the TicTacToe matroid.

The dual of the TicTacToe matroid is not algebraic; it is unknown whether the TicTacToe matroid itself is algebraic. See [Hoc].

EXAMPLES:

```
sage: M = matroids.named_matroids.TicTacToe()
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**Uniform**(*r*, *n*)
Return the uniform matroid of rank r on n elements.

INPUT:

- `r` – a nonnegative integer. The rank of the uniform matroid.
- `n` – a nonnegative integer. The number of elements of the uniform matroid.

OUTPUT:

The uniform matroid $U_{r,n}$.

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. See [Oxl2011], p. 660.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.Uniform(2, 5); M
U(2, 5): Matroid of rank 2 on 5 elements with circuit-closures
{2: {{0, 1, 2, 3, 4}}}
sage: M.dual().is_isomorphic(matroids.Uniform(3, 5))
True
sage: setprint(M.hyperplanes())
[{0}, {1}, {2}, {3}, {4}]
sage: M.has_line_minor(6)
False
sage: M.is_valid()
True
```

Check that bug trac ticket #15292 was fixed:

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

sage.matroids.catalog.**Vamos**()

Return the Vamos matroid, represented as circuit closures.

The Vamos matroid, or Vamos cube, or V_8 is a 8-element matroid of rank-4. It violates Ingleton's condition for representability over a division ring. It is not algebraic. See [Oxl2011], p. 649.

EXAMPLES:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Vamos(); M
Vamos: Matroid of rank 4 on 8 elements with circuit-closures
{3: {{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'e', 'f', 'g', 'h'},
     {'a', 'b', 'g', 'h'}, {'c', 'd', 'e', 'f'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
sage: setprint(M.nonbases())
[{'a', 'b', 'c', 'd'}, {'a', 'b', 'e', 'f'}, {'a', 'b', 'g', 'h'},
 {'c', 'd', 'e', 'f'}, {'e', 'f', 'g', 'h'}]
sage: M.is_dependent(['c', 'd', 'g', 'h'])
False
sage: M.is_valid() # long time
True
```

sage.matroids.catalog.**Wheel**(*n*, *field=None*, *ring=None*)

Return the rank-n wheel.

INPUT:

- `n` – 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 **Z**, then output will be a regular matroid.
- `field` – any field. Same as `ring`, but only fields are allowed.

OUTPUT:

The rank-n wheel matroid, represented as a regular matroid.

See [Oxl2011], p. 659.

EXAMPLES:

```
sage: M = matroids.Wheel(5); M
Wheel(5): Regular matroid of rank 5 on 10 elements with 121 bases
sage: M.tutte_polynomial()
x^5 + y^5 + 5*x^4 + 5*x^3*y + 5*x^2*y^2 + 5*x*y^3 + 5*y^4 + 10*x^3 +
15*x^2*y + 15*x*y^2 + 10*y^3 + 10*x^2 + 16*x*y + 10*y^2 + 4*x + 4*y
sage: M.is_valid()
True
sage: M = matroids.Wheel(3)
sage: M.is_isomorphic(matroids.CompleteGraphic(4))
True
sage: M.is_isomorphic(matroids.Wheel(3,field=GF(3)))
True
sage: M = matroids.Wheel(3,field=GF(3)); M
Wheel(3): Ternary matroid of rank 3 on 6 elements, type 0+
```

sage.matroids.catalog.**Whirl**(*n*)

Return the rank-n whirl.

INPUT:

- `n` – a positive integer. The rank of the desired matroid.

OUTPUT:

The rank-n whirl matroid, represented as a ternary matroid.

The whirl is the unique relaxation of the wheel. See [Oxl2011], p. 659.

EXAMPLES:

```
sage: M = matroids.Whirl(5); M
Whirl(5): Ternary matroid of rank 5 on 10 elements, type 0-
sage: M.is_valid()
True
sage: M.tutte_polynomial()
x^5 + y^5 + 5*x^4 + 5*x^3*y + 5*x^2*y^2 + 5*x*y^3 + 5*y^4 + 10*x^3 +
15*x^2*y + 15*x*y^2 + 10*y^3 + 10*x^2 + 15*x*y + 10*y^2 + 5*x + 5*y
sage: M.is_isomorphic(matroids.Wheel(5))
False
sage: M = matroids.Whirl(3)
sage: M.is_isomorphic(matroids.CompleteGraphic(4))
False
```

Todo: Optional arguments `ring` and `x`, such that the resulting matroid is represented over `ring` by a reduced matrix like `[-1 0 x] [ 1 -1 0] [ 0 1 -1]`

CHAPTER

THREE

CONCRETE IMPLEMENTATIONS

3.1 Basis matroids

In a matroid, a basis is an inclusionwise maximal independent set. The common cardinality of all bases is the rank of the matroid. Matroids are uniquely determined by their set of bases.

This module defines the class *BasisMatroid*, which internally represents a matroid as a set of bases. It is a subclass of *BasisExchangeMatroid*, and as such it inherits all method from that class and from the class *Matroid*. Additionally, it provides the following methods:

- *is_distinguished()*
- *relabel()*

3.1.1 Construction

A `BasisMatroid` can be created from another matroid, from a list of bases, or from a list of nonbases. For a full description of allowed inputs, see *below*. It is recommended to use the *Matroid()* function for easy construction of a `BasisMatroid`. For direct access to the `BasisMatroid` constructor, run:

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

See also *sage.matroids.advanced*.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M1 = BasisMatroid(groundset='abcd', bases=['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
sage: M2 = Matroid(['ab', 'ac', 'ad', 'bc', 'bd', 'cd'])
sage: M1 == M2
True
```

3.1.2 Implementation

The set of bases is compactly stored in a bitset which takes $O(binomial(N, R))$ bits of space, where N is the cardinality of the groundset and R is the rank. `BasisMatroid` inherits the matroid oracle from its parent class `BasisExchangeMatroid`, by providing the elementary functions for exploring the base exchange graph. In addition, `BasisMatroid` has methods for constructing minors, duals, single-element extensions, for testing matroid isomorphism and minor inclusion.

AUTHORS:

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

3.1.3 Methods

class sage.matroids.basis_matroid.**BasisMatroid**

Bases: *sage.matroids.basis_exchange_matroid.BasisExchangeMatroid*

Create general matroid, stored as a set of bases.

INPUT:

- `M` (optional) – a matroid.
- `groundset` (optional) – any iterable set.
- `bases` (optional) – a set of subsets of `groundset`.
- `nonbases` (optional) – a set of subsets of `groundset`.
- `rank` (optional) – a natural number

EXAMPLES:

The empty matroid:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid()
sage: M.groundset()
frozenset()
sage: M.full_rank()
0
```

Create a BasisMatroid instance out of any other matroid:

```
sage: from sage.matroids.advanced import *
sage: F = matroids.named_matroids.Fano()
sage: M = BasisMatroid(F)
sage: F.groundset() == M.groundset()
True
sage: len(set(F.bases()).difference(M.bases()))
0
```

It is possible to provide either bases or nonbases:

```
sage: from sage.matroids.advanced import *
sage: M1 = BasisMatroid(groundset='abc', bases=['ab', 'ac'] )
sage: M2 = BasisMatroid(groundset='abc', nonbases=['bc'])
sage: M1 == M2
True
```

Providing only groundset and rank creates a uniform matroid:

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

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

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

```
2
sage: M1.is_valid()
False
```

bases()

Return the list of bases of the matroid.

A *basis* is a maximal independent set.

OUTPUT:

An iterable containing all bases of the matroid.

EXAMPLES:

```
sage: M = Matroid(bases=matroids.named_matroids.Fano().bases())
sage: M
Matroid of rank 3 on 7 elements with 28 bases
sage: len(M.bases())
28
```

bases_count()

Return the number of bases of the matroid.

OUTPUT:

Integer.

EXAMPLES:

```
sage: M = Matroid(bases=matroids.named_matroids.Fano().bases())
sage: M
Matroid of rank 3 on 7 elements with 28 bases
sage: M.bases_count()
28
```

dual()

Return the dual of the matroid.

Let M be a matroid with ground set 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.

OUTPUT:

The dual matroid.

EXAMPLES:

```
sage: M = Matroid(bases=matroids.named_matroids.Pappus().bases())
sage: 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` – an element of the ground set

OUTPUT:

Boolean.

See also:

M.extensions(), M.linear_subclasses(), sage.matroids.extension

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid(matroids.named_matroids.N1())
sage: sorted([e for e in M.groundset() if M.is_distinguished(e)])
['c', 'g', 'h', 'j']
```

nonbases()

Return the list of nonbases of the matroid.

A *nonbasis* is a set with cardinality `self.full_rank()` that is not a basis.

OUTPUT:

An iterable containing the nonbases of the matroid.

See also:

Matroid.basis()

EXAMPLES:

```
sage: M = Matroid(bases=matroids.named_matroids.Fano().bases())
sage: M
Matroid of rank 3 on 7 elements with 28 bases
sage: len(M.nonbases())
7
```

relabel(l)

Return an isomorphic matroid with relabeled groundset.

The output is obtained by relabeling each element `e` by `l[e]`, where `l` is a given injective map. If `e not in l` then the identity map is assumed.

INPUT:

- `l` – a python object such that $l[e]$ is the new label of e.

OUTPUT:

A matroid.

Todo: Write abstract `RelabeledMatroid` class, and add `relabel()` method to the main `Matroid` class, together with `_relabel()` method that can be replaced by subclasses. Use the code from `is_isomorphism()` in `relabel()` to deal with a variety of input methods for the relabeling.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid(matroids.named_matroids.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']
```

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:

A matroid.

See also:

M.extension(), *M.contract()*

EXAMPLES:

```
sage: M = Matroid(bases=matroids.named_matroids.N2().bases())
sage: M.truncation()
Matroid of rank 5 on 12 elements with 702 bases
sage: M.f_vector()
[1, 12, 66, 190, 258, 99, 1]
sage: M.truncation().f_vector()
[1, 12, 66, 190, 258, 1]
```

3.2 Circuit closures matroids

Matroids are characterized by a list of all tuples (C, k), where C is the closure of a circuit, and k the rank of C. The CircuitClosuresMatroid class implements matroids using this information as data.

3.2.1 Construction

A `CircuitClosuresMatroid` can be created from another matroid or from a list of circuit-closures. For a full description of allowed inputs, see *below*. It is recommended to use the *Matroid()* function for a more flexible construction of a `CircuitClosuresMatroid`. For direct access to the `CircuitClosuresMatroid` constructor, run:

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

See also *sage.matroids.advanced*.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M1 = CircuitClosuresMatroid(groundset='abcdef',
....:                 circuit_closures={2: ['abc', 'ade'], 3: ['abcdef']})
```

```
sage: M2 = Matroid(circuit_closures={2: ['abc', 'ade'], 3: ['abcdef']})
sage: M3 = Matroid(circuit_closures={(2, 'abc'),
....:                                (3, 'abcdef'), (2, 'ade')})
sage: M1 == M2
True
sage: M1 == M3
True
```

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

```
sage: from sage.matroids.advanced import *
sage: M = CircuitClosuresMatroid(groundset='abcdef',
....:                 circuit_closures={2: ['abc', 'ade'], 3: ['abcdef']})
sage: isinstance(M.contract('a'), MinorMatroid)
True
sage: isinstance(M.dual(), DualMatroid)
True
```

AUTHORS:

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

3.2.2 Methods

class sage.matroids.circuit_closures_matroid.**CircuitClosuresMatroid**

Bases: *sage.matroids.matroid.Matroid*

A general matroid M is characterized by its rank $r(M)$ and the set of pairs

$(k, \{ \text{ closure } (C) : C \text{`circuit of `} M, r(C) = k\})$ for $k = 0, .., r(M) - 1$

As each independent set of size k is in at most one closure(C) of rank k, and each closure(C) of rank k contains at least $k + 1$ independent sets of size k, there are at most $\binom{n}{k}/(k + 1)$ such closures-of-circuits of rank k. Each closure(C) takes $O(n)$ bits to store, giving an upper bound of $O(2^n)$ on the space complexity of the entire matroid.

A subset X of the ground set is independent if and only if

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

So determining whether a set is independent takes time proportional to the space complexity of the matroid.

INPUT:

- `M` – (default: `None`) an arbitrary matroid.
- `groundset` – (default: `None`) the groundset of a matroid.
- `circuit_closures` – (default: `None`) the collection of circuit closures of a matroid, presented as a dictionary whose keys are ranks, and whose values are sets of circuit closures of the specified rank.

OUTPUT:

- If the input is a matroid `M`, return a `CircuitClosuresMatroid` instance representing `M`.
- Otherwise, return a `CircuitClosuresMatroid` instance based on `groundset` and `circuit_closures`.

Note: For a more flexible means of input, use the `Matroid()` function.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = CircuitClosuresMatroid(matroids.named_matroids.Fano())
sage: M
Matroid of rank 3 on 7 elements with circuit-closures
{2: {{'b', 'e', 'g'}, {'b', 'c', 'd'}, {'a', 'c', 'e'},
     {'c', 'f', 'g'}, {'d', 'e', 'f'}, {'a', 'd', 'g'},
     {'a', 'b', 'f'}}, 3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g'}}}
sage: M = CircuitClosuresMatroid(groundset='abcdefgh',
....:             circuit_closures={3: ['edfg', 'acdg', 'bcfg', 'cefh',
....:                  'afgh', 'abce', 'abdf', 'begh', 'bcdh', 'adeh'],
....:                                4: ['abcdefgh']})
sage: M.equals(matroids.named_matroids.P8())
True
```

circuit_closures()

Return the list of 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:

Matroid.circuit(), *Matroid.closure()*

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = CircuitClosuresMatroid(matroids.named_matroids.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']]
```

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.named_matroids.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:

A set.

EXAMPLES:

```
sage: M = matroids.named_matroids.Pappus()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

3.3 Linear matroids

When A is an r times E matrix, the linear matroid $M[A]$ has ground set E and, for independent sets, all F subset of E such that the columns of $M[A]$ indexed by F are linearly independent.

3.3.1 Construction

The recommended way to create a linear matroid is by using the *Matroid()* function, with a representation matrix A as input. This function will intelligently choose one of the dedicated classes *BinaryMatroid*, *TernaryMatroid*, *QuaternaryMatroid*, *RegularMatroid* when appropriate. However, invoking the classes directly is possible too. To get access to them, type:

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

See also *sage.matroids.advanced*. In both cases, it is possible to provide a reduced matrix B, to create the matroid induced by $A = [IB]$:

```
sage: from sage.matroids.advanced import *
sage: A = Matrix(GF(2), [[1, 0, 0, 1, 1, 0, 1], [0, 1, 0, 1, 0, 1, 1],
....:                    [0, 0, 1, 0, 1, 1, 1]])
sage: B = Matrix(GF(2), [[1, 1, 0, 1], [1, 0, 1, 1], [0, 1, 1, 1]])
sage: M1 = Matroid(A)
sage: M2 = LinearMatroid(A)
sage: M3 = BinaryMatroid(A)
sage: M4 = Matroid(reduced_matrix=B)
sage: M5 = LinearMatroid(reduced_matrix=B)
sage: isinstance(M1, BinaryMatroid)
True
sage: M1.equals(M2)
True
sage: M1.equals(M3)
True
sage: M1 == M4
True
sage: M1.is_field_isomorphic(M5)
True
sage: M2 == M3  # comparing LinearMatroid and BinaryMatroid always yields False
False
```

3.3.2 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()*
- *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

3.3.3 Methods

class sage.matroids.linear_matroid.**BinaryMatroid**

Bases: *sage.matroids.linear_matroid.LinearMatroid*

Binary matroids.

A binary matroid is a linear matroid represented over the finite field with two elements. See *LinearMatroid* for a definition.

The simplest way to create a `BinaryMatroid` is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object E can be given as a groundset. If E is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a *reduced* matrix B, to create the matroid induced by $A = [IB]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \quad B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.
- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) ignored.
- `keep_initial_representation` – (default: `True`) decides whether or not an internal copy of the input matrix should be preserved. This can help to see the structure of the matroid (e.g. in the case of graphic matroids), and makes it easier to look at extensions. However, the input matrix may have redundant rows, and sometimes it is desirable to store only a row-reduced copy.
- `basis` – (default: `None`) When provided, this is an ordered subset of `groundset`, such that the submatrix of `matrix` indexed by `basis` is an identity matrix. In this case, no row reduction takes place in the initialization phase.

OUTPUT:

A *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 *
```

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
```

base_ring()

Return the base ring of the matrix representing the matroid, in this case $\mathbf{F}_2$.

EXAMPLES:

```
sage: M = matroids.named_matroids.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:

```
sage: M = matroids.named_matroids.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.named_matroids.Fano()
sage: N.binary_matroid() is N
True
```

brown_invariant()

Return the value of Brown's invariant for the binary matroid.

For a binary space V, consider the sum $B(V) := \sum_{v \in V} i^{|v|}$, where $|v|$ denotes the number of nonzero entries of a binary vector v. The value of the Tutte Polynomial in the point $(-i, i)$ can be expressed in terms of $B(V)$, see [Pen2012]. If $|v|$ equals 2 modulo 4 for some $v \in V \cap V^{\perp}$, then $B(V) = 0$. In this case, Browns invariant is not defined. Otherwise, $B(V) = \sqrt{2}^k \exp(\sigma\pi i/4)$ for some integers k, σ. In that case, k equals the bycycle dimension of V, and Browns invariant for V is defined as σ modulo 8.

The Brown invariant of a binary matroid equals the Brown invariant of its cocycle-space.

OUTPUT:

Integer.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: M.brown_invariant()
0
sage: M = Matroid(Matrix(GF(2), 3, 8, [[1, 0, 0, 1, 1, 1, 1, 1],
....:                                  [0, 1, 0, 1, 1, 0, 0, 0],
....:                                  [0, 0, 1, 0, 0, 1, 1, 0]]))
sage: M.brown_invariant() is None
True
```

characteristic()

Return the characteristic of the base ring of the matrix representing the matroid, in this case 2.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: M.characteristic()
2
```

is_binary(*randomized_tests=1*)

Decide if `self` is a binary matroid.

INPUT:

- `randomized_tests` – Ignored.

OUTPUT:

A Boolean.

ALGORITHM:

`self` is a binary matroid, so just return `True`.

See also:

M.is_binary()

EXAMPLES:

```
sage: N = matroids.named_matroids.Fano()
sage: N.is_binary()
True
```

is_graphic()

Test if the binary matroid is graphic.

A matroid is *graphic* if there exists a graph whose edge set equals the groundset of the matroid, such that a subset of elements of the matroid is independent if and only if the corresponding subgraph is acyclic.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: R10 = matroids.named_matroids.R10()
sage: M = Matroid(ring=GF(2), reduced_matrix=R10.representation(
....:                                   reduced=True, labels=False))
sage: M.is_graphic()
False
sage: K5 = Matroid(graphs.CompleteGraph(5), regular = True)
sage: M = Matroid(ring=GF(2), reduced_matrix=K5.representation(
....:                                   reduced=True, labels=False))
sage: M.is_graphic()
True
sage: M.dual().is_graphic()
False
```

ALGORITHM:

In a recent paper, Geelen and Gerards [GG2012] reduced the problem to testing if a system of linear equations has a solution. While not the fastest method, and not necessarily constructive (in the presence of 2-separations especially), it is easy to implement.

is_valid()

Test if the data obey the matroid axioms.

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

OUTPUT:

`True`.

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(2), [[]]))
sage: M.is_valid()
True
```

class sage.matroids.linear_matroid.**LinearMatroid**

Bases: *sage.matroids.basis_exchange_matroid.BasisExchangeMatroid*

Linear matroids.

When A is an r times E matrix, the linear matroid $M[A]$ has ground set E and set of independent sets

$$I(A) = \{F \subseteq E : \text{the columns of } A \text{ indexed by } F \text{ are linearly independent } \}$$

The simplest way to create a LinearMatroid is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object `E` can be given as a groundset. If `E` is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a *reduced* matrix B, to create the matroid induced by $A = [IB]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \quad B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.

- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) the desired base ring of the matrix. If the base ring is different, an attempt will be made to create a new matrix with the correct base ring.
- `keep_initial_representation` – (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.

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 *
```

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
```

base_ring()

Return the base ring of the matrix representing the matroid.

EXAMPLES:

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

characteristic()

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

EXAMPLES:

```
sage: M = Matroid(matrix=Matrix(GF(5), [[1, 0, 1, 1, 1],
....:                                   [0, 1, 1, 2, 3]]))
sage: M.characteristic()
5
```

cross_ratio(*F*, *a*, *b*, *c*, *d*)

Return the cross ratio of the four ordered points `a, b, c, d` after contracting a flat `F` of codimension 2.

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

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & x & 1 \end{bmatrix}$$

The cross ratio of the ordered tuple (a, b, c, d) equals x. This method looks at such minors where `F` is a flat to be contracted, and all remaining elements other than `a, b, c, d` are deleted.

INPUT:

- `F` – A flat of codimension 2
- `a, b, c, d` – elements of the groundset

OUTPUT:

The cross ratio of the four points on the line obtained by contracting `F`.

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 0, 1, 1, 1],
....:                            [0, 1, 0, 1, 2, 4],
....:                            [0, 0, 1, 3, 2, 6]]))
sage: M.cross_ratio([0], 1, 2, 3, 5)
4

sage: M = Matroid(ring=GF(7), matrix=[[1, 0, 1, 1], [0, 1, 1, 1]])
sage: M.cross_ratio(set(), 0, 1, 2, 3)
Traceback (most recent call last):
...
ValueError: points a, b, c, d do not form a 4-point line in M/F
```

cross_ratios(*hyperlines=None*)

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

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

$$\begin{matrix} 1 & 0 & 1 & 1 \\ 0 & 1 & x & 1 \end{matrix}$$

The cross ratio of the ordered tuple (a, b, c, d) equals x. The set of all cross ratios of a matroid is the set of cross ratios of all such minors.

INPUT:

- `hyperlines` – (optional) a set of flats of the matroid, of rank $r - 2$, where r is the rank of the matroid. If not given, then `hyperlines` defaults to all such flats.

OUTPUT:

A list of all cross ratios of this linearly represented matroid that occur in rank-2 minors that arise by contracting a flat `F` in `hyperlines` (so by default, those are all cross ratios).

See also:

M.cross_ratio()

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 0, 1, 1, 1],
....:                            [0, 1, 0, 1, 2, 4],
....:                            [0, 0, 1, 3, 2, 5]]))
sage: sorted(M.cross_ratios())
[2, 3, 4, 5, 6]
sage: M = Matroid(graphs.CompleteGraph(5), regular = True)
sage: M.cross_ratios()
set()
```

dual()

Return the dual of the matroid.

Let M be a matroid with ground set 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:

```
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
```

fundamental_cocycle(B, e)

Return the fundamental cycle, relative to `B`, containing element `e`.

This is the *fundamental cocircuit* together with an appropriate signing from the field, such that $Av = 0$, where A is a representation matrix of the dual, and v the vector corresponding to the output.

INPUT:

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

OUTPUT:

A dictionary mapping elements of `M.fundamental_cocircuit(B, e)` to elements of the ring.

See also:

M.fundamental_cocircuit()

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 1, 1], [0, 1, 1, 4]]))
sage: v = M.fundamental_cocycle([0, 1], 0)
sage: [v[0], v[2], v[3]]
[1, 1, 1]
```

```
sage: frozenset(v.keys()) == M.fundamental_cocircuit([0, 1], 0)
True
```

fundamental_cycle(*B*, *e*)

Return the fundamental cycle, relative to `B`, containing element `e`.

This is the *fundamental circuit* together with an appropriate signing from the field, such that $Av = 0$, where A is the representation matrix, and v the vector corresponding to the output.

INPUT:

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

OUTPUT:

A dictionary mapping elements of `M.fundamental_circuit(B, e)` to elements of the ring.

See also:

M.fundamental_circuit()

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 1, 1], [0, 1, 1, 4]]))
sage: v = M.fundamental_cycle([0, 1], 3)
sage: [v[0], v[1], v[3]]
[6, 3, 1]
sage: frozenset(v.keys()) == M.fundamental_circuit([0, 1], 3)
True
```

has_field_minor(*N*)

Check if `self` has a minor field isomorphic to `N`.

INPUT:

- `N` – A 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.named_matroids.Fano().has_field_minor(M)
False
sage: matroids.named_matroids.NonFano().has_field_minor(M)
True
```

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

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

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

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

INPUT:

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

OUTPUT:

Boolean or tuple.

EXAMPLES:

```
sage: M = matroids.named_matroids.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'}))
```

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` – A matroid.

OUTPUT:

Boolean.

See also:

M.equals(), *M.is_field_isomorphism()*, *M.is_field_isomorphic()*

EXAMPLES:

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

```
sage: from sage.matroids.advanced import *
sage: M = matroids.named_matroids.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:

```
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
```

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` – A matroid.

OUTPUT:

Boolean.

See also:

M.is_isomorphic(), *M.is_field_isomorphism()*, *M.is_field_equivalent()*

EXAMPLES:

```
sage: M1 = matroids.Wheel(3)
sage: M2 = Matroid(graphs.CompleteGraph(4), regular = True)
sage: M1.is_field_isomorphic(M2)
True
sage: M3 = Matroid(bases=M1.bases())
sage: M1.is_field_isomorphic(M3)
Traceback (most recent call last):
...
AttributeError: 'sage.matroids.basis_matroid.BasisMatroid' object
has no attribute 'base_ring'
sage: from sage.matroids.advanced import *
sage: M4 = BinaryMatroid(Matrix(M1))
sage: M5 = LinearMatroid(reduced_matrix=Matrix(GF(2), [[-1, 0, 1],
....:                                    [1, -1, 0], [0, 1, -1]]))
sage: M4.is_field_isomorphic(M5)
True

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

is_field_isomorphism(*other*, *morphism*)

Test if a provided morphism induces a bijection between represented matroids.

Two represented matroids are *field isomorphic* if the bijection `morphism` between them induces a field equivalence between their representation matrices: the matrices are equal up to row operations and column scaling. This implies that the matroids are isomorphic, but the converse is false: two isomorphic matroids can be represented by matrices that are not field equivalent.

INPUT:

- `other` – A 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.named_matroids.Fano()
sage: N = matroids.named_matroids.NonFano()
sage: N.is_field_isomorphism(M, {e:e for e in M.groundset()})
False
```

```
sage: from sage.matroids.advanced import *
sage: M = matroids.named_matroids.Fano() \ ['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
```

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:

```
sage: M = Matroid(ring=QQ, reduced_matrix=Matrix(ZZ,
....:                           [[1, 0, 1], [1, 1, 0], [0, 1, 1]]))
sage: M.is_valid()
True
sage: from sage.matroids.advanced import *  # LinearMatroid
sage: M = LinearMatroid(ring=ZZ, reduced_matrix=Matrix(ZZ,
....:                           [[1, 0, 1], [1, 1, 0], [0, 1, 1]]))
sage: M.is_valid()
False
```

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 ground set to elements of the base ring.

OUTPUT:

A linear matroid $N = M([A0; -c1])$, where A is a matrix such that the current matroid is $M[A]$, and c is either given by `row` (relative to `self.representation()`) or has nonzero entries given by `cochain`.

Note: The minus sign is to ensure this method commutes with dualizing. See the last example.

See also:

M.coextension(), *M.linear_extension()*, *M.dual()*

EXAMPLES:

```
sage: M = Matroid(ring=GF(2), matrix=[[1, 1, 0, 1, 0, 0],
....:                                 [1, 0, 1, 0, 1, 0],
....:                                 [0, 1, 1, 0, 0, 1],
....:                                 [0, 0, 0, 1, 1, 1]])
sage: M.linear_coextension(6, {0:1, 5: 1}).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 0]
[0 1 1 0 0 1 0]
[0 0 0 1 1 1 0]
[1 0 0 0 0 1 1]
sage: M.linear_coextension(6, row=[0,1,1,1,0,1]).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 0]
[0 1 1 0 0 1 0]
[0 0 0 1 1 1 0]
[0 1 1 1 0 1 1]
```

Coextending commutes with dualizing:

```
sage: M = matroids.named_matroids.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
```

linear_coextension_cochains(*F=None*, *cosimple=False*, *fundamentals=None*)

Create a list of cochains that determine the single-element coextensions of this linear matroid representation.

A cochain is a dictionary, mapping elements from the groundset to elements of the base ring. If A represents the current matroid, then the coextension is given by $N = M([A0; -c1])$, with the entries of c given by the cochain. Note that the matroid does not change when row operations are carried out on A.

INPUT:

- `F` – (default: `self.groundset()`) a subset of the groundset.
- `cosimple` – (default: `False`) a boolean variable.
- `fundamentals` – (default: `None`) a set elements of the base ring.

OUTPUT:

A list of cochains, so each single-element coextension of this linear matroid representation is given by one of these cochains.

If one or more of the above inputs is given, the list is restricted to chains

- so that the support of each cochain lies in `F`, if given
- so that the cochain does not generate a series extension or coloop, if `cosimple = True`
- so that in the coextension generated by this cochain, the cross ratios are restricted to `fundamentals`, if given.

See also:

M.linear_coextension(), M.linear_coextensions(), M.cross_ratios()

EXAMPLES:

```
sage: M = Matroid(reduced_matrix=Matrix(GF(2),
....:                             [[1, 1, 0], [1, 0, 1], [0, 1, 1]]))
sage: len(M.linear_coextension_cochains())
8
sage: len(M.linear_coextension_cochains(F=[0, 1]))
4
sage: len(M.linear_coextension_cochains(F=[0, 1], cosimple=True))
0
sage: M.linear_coextension_cochains(F=[3, 4, 5], cosimple=True)
[{3: 1, 4: 1, 5: 1}]
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}]
```

linear_coextensions(*element=None*, *F=None*, *cosimple=False*, *fundamentals=None*)

Create a list of linear matroids represented by corank-preserving single-element coextensions of this linear matroid representation.

INPUT:

- `element` – (default: `None`) the name of the new element of the groundset.
- `F` – (default: `None`) a subset of the ground set.
- `cosimple` – (default: `False`) a boolean variable.
- `fundamentals` – (default: `None`) a set elements of the base ring.

OUTPUT:

A list of linear matroids represented by corank-preserving single-element coextensions of this linear matroid representation. In particular, the coextension by a loop is not generated.

If one or more of the above inputs is given, the list is restricted to coextensions

- so that the new element lies in the cospan of `F`, if given.
- so that the new element is no coloop and is not in series with another element, if `cosimple = True`.
- so that in the representation of the coextension, the cross ratios are restricted to `fundamentals`, if given. Note that it is assumed that the cross ratios of the input matroid already satisfy this condition.

See also:

M.linear_coextension(), *M.linear_coextension_cochains()*, *M.cross_ratios()*

EXAMPLES:

```
sage: M = Matroid(ring=GF(2),
....:         reduced_matrix=[[-1, 0, 1], [1, -1, 0], [0, 1, -1]])
sage: len(M.linear_coextensions())
8
sage: S = M.linear_coextensions(cosimple=True)
sage: S
[Binary matroid of rank 4 on 7 elements, type (3, 7)]
sage: F7 = matroids.named_matroids.Fano()
sage: S[0].is_field_isomorphic(F7.dual())
True
sage: M = Matroid(ring=QQ,
....:             reduced_matrix=[[1, 0, 1], [1, 1, 0], [0, 1, 1]])
sage: S = M.linear_coextensions(cosimple=True,
....:                           fundamentals=[1, -1, 1/2, 2])
sage: len(S)
7
sage: NF7 = matroids.named_matroids.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
```

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 ground set to elements of the base ring.

OUTPUT:

A linear matroid $N = M([A\ \ b])$, where A is a matrix such that the current matroid is $M[A]$, and b is either given by `col` or is a weighted combination of columns of A, the weights being given by `chain`.

See also:

M.extension().

EXAMPLES:

```
sage: M = Matroid(ring=GF(2), matrix=[[1, 1, 0, 1, 0, 0],
....:                                 [1, 0, 1, 0, 1, 0],
....:                                 [0, 1, 1, 0, 0, 1],
....:                                 [0, 0, 0, 1, 1, 1]])
sage: M.linear_extension(6, {0:1, 5: 1}).representation()
[1 1 0 1 0 0 1]
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
sage: M.linear_extension(6, col=[0, 1, 1, 1]).representation()
[1 1 0 1 0 0 0]
[1 0 1 0 1 0 1]
[0 1 1 0 0 1 1]
[0 0 0 1 1 1 1]
```

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`) a boolean variable.
- `fundamentals` – (default: `None`) a set elements of the base ring.

OUTPUT:

A list of chains, so each single-element extension of this linear matroid representation is given by one of these chains.

If one or more of the above inputs is given, the list is restricted to chains

- so that the support of each chain lies in `F`, if given
- so that the chain does not generate a parallel extension or loop, if `simple = True`
- so that in the extension generated by this chain, the cross ratios are restricted to `fundamentals`, if given.

See also:

M.linear_extension(), *M.linear_extensions()*, *M.cross_ratios()*

EXAMPLES:

```
sage: M = Matroid(reduced_matrix=Matrix(GF(2),
....:                           [[1, 1, 0], [1, 0, 1], [0, 1, 1]]))
sage: len(M.linear_extension_chains())
8
sage: len(M.linear_extension_chains(F=[0, 1]))
```

```
4
sage: len(M.linear_extension_chains(F=[0, 1], simple=True))
0
sage: M.linear_extension_chains(F=[0, 1, 2], simple=True)
[{0: 1, 1: 1, 2: 1}]
sage: N = Matroid(ring=QQ,
....:             reduced_matrix=[[-1, -1, 0], [1, 0, -1], [0, 1, 1]])
sage: N.linear_extension_chains(F=[0, 1], simple=True,
....:                           fundamentals=set([1, -1, 1/2, 2]))
[{0: 1, 1: 1}, {0: -1/2, 1: 1}, {0: -2, 1: 1}]
```

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 ground set.
- `simple` – (default: `False`) a boolean variable.
- `fundamentals` – (default: `None`) a set elements of the base ring.

OUTPUT:

A list of linear matroids represented by rank-preserving single-element extensions of this linear matroid representation. In particular, the extension by a coloop is not generated.

If one or more of the above inputs is given, the list is restricted to matroids

- so that the new element is spanned by `F`, if given
- so that the new element is not a loop or in a parallel pair, if `simple=True`
- so that in the representation of the extension, the cross ratios are restricted to `fundamentals`, if given. Note that it is assumed that the cross ratios of the input matroid already satisfy this condition.

See also:

M.linear_extension(), *M.linear_extension_chains()*, *M.cross_ratios()*

EXAMPLES:

```
sage: M = Matroid(ring=GF(2),
....:             reduced_matrix=[[-1, 0, 1], [1, -1, 0], [0, 1, -1]])
sage: len(M.linear_extensions())
8
sage: S = M.linear_extensions(simple=True)
sage: S
[Binary matroid of rank 3 on 7 elements, type (3, 0)]
sage: S[0].is_field_isomorphic(matroids.named_matroids.Fano())
True
sage: M = Matroid(ring=QQ,
....:                reduced_matrix=[[1, 0, 1], [1, 1, 0], [0, 1, 1]])
sage: S = M.linear_extensions(simple=True,
....:                         fundamentals=[1, -1, 1/2, 2])
sage: len(S)
7
sage: any(N.is_isomorphic(matroids.named_matroids.NonFano())
....:     for N in S)
True
```

```
sage: len(M.linear_extensions(simple=True,
....:                         fundamentals=[1, -1, 1/2, 2], F=[0, 1]))
1
```

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 $rimesn$ 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 $rimesr$ 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` – (default: `None`) an ordering of the groundset elements. If provided, the columns (and, in case of a reduced representation, rows) will be presented in the given order.
- `lift_map` – (default: `None`) a dictionary containing the cross ratios of the representing matrix in its domain. If provided, the representation will be transformed by mapping its cross ratios according to `lift_map`.

OUTPUT:

- `A` – a full or reduced representation matrix of `self`; or
- `(A, E)` – a full representation matrix `A` and a list `E` of column labels; or
- `(A, R, C)` – a reduced representation matrix and a list `R` of row labels and a list `C` of column labels.

If `B == None` and `reduced == False` and `order == None` then this method will always output the same matrix (except when `M._forget()` is called): either the matrix used as input to create the matroid, or a matrix in which the lexicographically least basis corresponds to an identity. If only `order` is not `None`, the columns of this matrix will be permuted accordingly.

If a `lift_map` is provided, then the resulting matrix will be lifted using the method *lift_cross_ratios()* See the docstring of this method for further details.

Note: A shortcut for `M.representation()` is `Matrix(M)`.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: M.representation()
[1 0 0 0 1 1 1]
```

```
[0 1 0 1 0 1 1]
[0 0 1 1 1 0 1]
sage: Matrix(M) == M.representation()
True
sage: M.representation(labels=True)
(
[1 0 0 0 1 1 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 0 1], ['a', 'b', 'c', 'd', 'e', 'f', 'g']
)
sage: M.representation(B='efg')
[1 1 0 1 1 0 0]
[1 0 1 1 0 1 0]
[1 1 1 0 0 0 1]
sage: M.representation(B='efg', order='efgabcd')
[1 0 0 1 1 0 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 1 0]
sage: M.representation(B='abc', reduced=True)
(
[0 1 1 1]
[1 0 1 1]
[1 1 0 1], ['a', 'b', 'c'], ['d', 'e', 'f', 'g']
)
sage: M.representation(B='efg', reduced=True, labels=False,
....:                  order='gfeabcd')
[1 1 1 0]
[1 0 1 1]
[1 1 0 1]
```

representation_vectors()

Return a dictionary that associates a column vector with each element of the matroid.

See also:

M.representation()

EXAMPLES:

```
sage: M = matroids.named_matroids.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: *sage.matroids.linear_matroid.LinearMatroid*

Quaternary matroids.

A quaternary matroid is a linear matroid represented over the finite field with four elements. See *LinearMatroid* for a definition.

The simplest way to create a `QuaternaryMatroid` is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object E can be given as a groundset. If E is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a 'reduced' matrix B, to create the matroid induced by $A = [I \ B]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \quad B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.
- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) must be a copy of $\mathbf{F}_4$.
- `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 *
```

EXAMPLES:

```
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
```

base_ring()

Return the base ring of the matrix representing the matroid, in this case $\mathbf{F}_4$.

EXAMPLES:

```
sage: M = Matroid(ring=GF(4, 'y'), reduced_matrix=[[1, 0, 1],
....:                                              [0, 1, 1]])
sage: M.base_ring()
Finite Field in y of size 2^2
```

bicycle_dimension()

Return the bicycle dimension of the quaternary matroid.

The bicycle dimension of a linear subspace V is $\dim(V \cap V^{\perp})$. We use the inner product $< v, w >= v_1 w_1^* + ... + v_n w_n^*$, where w_i^* is obtained from w_i by applying the unique nontrivial field automorphism of $\mathbf{F}_4$.

The bicycle dimension of a matroid equals the bicycle dimension of its rowspace, and is a matroid invariant. See [Pen2012].

OUTPUT:

Integer.

EXAMPLES:

```
sage: M = matroids.named_matroids.Q10()
sage: M.bicycle_dimension()
0
```

characteristic()

Return the characteristic of the base ring of the matrix representing the matroid, in this case 2.

EXAMPLES:

```
sage: M = Matroid(ring=GF(4, 'y'), reduced_matrix=[[1, 0, 1],
....:                                              [0, 1, 1]])
sage: M.characteristic()
2
```

is_valid()

Test if the data obey the matroid axioms.

Since this is a linear matroid over the field $\mathbf{F}_4$, this is always the case.

OUTPUT:

`True`.

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(4, 'x'), [[]]))
sage: M.is_valid()
True
```

class sage.matroids.linear_matroid.**RegularMatroid**

Bases: *sage.matroids.linear_matroid.LinearMatroid*

Regular matroids.

A regular matroid is a linear matroid represented over the integers by a totally unimodular matrix.

The simplest way to create a `RegularMatroid` is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object E can be given as a groundset. If E is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a 'reduced' matrix B, to create the matroid induced by $A = [IB]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \ \ B]$ 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: 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)
sage: M
Regular matroid of rank 2 on 4 elements with 5 bases
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', regular=True)
sage: sorted(M.groundset())
['a', 'b', 'c', 'd']
```

base_ring()

Return the base ring of the matrix representing the matroid, in this case **Z**.

EXAMPLES:

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

bases_count()

Count the number of bases.

EXAMPLES:

```
sage: M = Matroid(graphs.CompleteGraph(5), regular = True)
sage: M.bases_count()
125
```

ALGORITHM:

Since the matroid is regular, we use Kirchhoff's Matrix-Tree Theorem. See also Wikipedia article Kirchhoff's_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.named_matroids.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.named_matroids.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:

```
sage: M = matroids.named_matroids.R10()
sage: M.has_line_minor(4)
False
sage: M.has_line_minor(4, certificate=True)
(False, None)
sage: M.has_line_minor(3)
True
sage: M.has_line_minor(3, certificate=True)
(True, frozenset({'a', 'b', 'c', 'g'}))
sage: M.has_line_minor(k=3, hyperlines=[['a', 'b', 'c'],
....:                                   ['a', 'b', 'd' ]])
True
sage: M.has_line_minor(k=3, hyperlines=[['a', 'b', 'c'],
....:                                   ['a', 'b', 'd' ]], certificate=True)
(True, frozenset({'a', 'b', 'c'}))
```

is_binary(*randomized_tests=1*)

Decide if `self` is a binary matroid.

INPUT:

- `randomized_tests` – Ignored.

OUTPUT:

A Boolean.

ALGORITHM:

`self` is a regular matroid, so just return `True`.

See also:

M.is_binary()

EXAMPLES:

```
sage: N = matroids.named_matroids.R10()
sage: 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.named_matroids.R10()
sage: M.is_graphic()
False
sage: M = Matroid(graphs.CompleteGraph(5), regular = True)
sage: M.is_graphic()
True
sage: M.dual().is_graphic()
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_ternary(*randomized_tests=1*)

Decide if `self` is a ternary matroid.

INPUT:

- `randomized_tests` – Ignored.

OUTPUT:

A Boolean.

ALGORITHM:

`self` is a regular matroid, so just return `True`.

See also:

M.is_ternary()

EXAMPLES:

```
sage: N = matroids.named_matroids.R10()
sage: 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:

```
sage: M = Matroid(Matrix(ZZ, [[1, 0, 0, 1, 1, 0, 1],
....:                         [0, 1, 0, 1, 0, 1, 1],
....:                         [0, 0, 1, 0, 1, 1, 1]]),
....:             regular=True, check=False)
sage: M.is_valid()
False
```

```
sage: M = Matroid(graphs.PetersenGraph())
sage: M.is_valid()
True
```

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

Return a ternary matroid representing `self`.

INPUT:

- `randomized_tests` – Ignored.
- `verify` – Ignored

OUTPUT:

A ternary matroid.

ALGORITHM:

`self` is a regular matroid, so just cast `self` to a TernaryMatroid.

See also:

M.ternary_matroid()

EXAMPLES:

```
sage: N = matroids.named_matroids.R10()
sage: N.ternary_matroid()
Ternary matroid of rank 5 on 10 elements, type 4+
```

class sage.matroids.linear_matroid.**TernaryMatroid**

Bases: *sage.matroids.linear_matroid.LinearMatroid*

Ternary matroids.

A ternary matroid is a linear matroid represented over the finite field with three elements. See *LinearMatroid* for a definition.

The simplest way to create a `TernaryMatroid` is by giving only a matrix A. Then, the groundset defaults to `range(A.ncols())`. Any iterable object E can be given as a groundset. If E is a list, then `E[i]` will label the i-th column of A. Another possibility is to specify a 'reduced' matrix B, to create the matroid induced by $A = [I \ B]$.

INPUT:

- `matrix` – (default: `None`) a matrix whose column vectors represent the matroid.
- `reduced_matrix` – (default: `None`) a matrix B such that $[I \ B]$ represents the matroid, where I is an identity matrix with the same number of rows as B. Only one of `matrix` and `reduced_matrix` should be provided.
- `groundset` – (default: `None`) an iterable containing the element labels. When provided, must have the correct number of elements: the number of columns of `matrix` or the number of rows plus the number of columns of `reduced_matrix`.
- `ring` – (default: `None`) ignored.
- `keep_initial_representation` – (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 *
```

EXAMPLES:

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

base_ring()

Return the base ring of the matrix representing the matroid, in this case $\mathbf{F}_3$.

EXAMPLES:

```
sage: M = matroids.named_matroids.NonFano()
sage: M.base_ring()
Finite Field of size 3
```

bicycle_dimension()

Return the bicycle dimension of the ternary matroid.

The bicycle dimension of a linear subspace V is $\dim(V \cap V^{\perp})$. The bicycle dimension of a matroid equals the bicycle dimension of its rowspace, and is a matroid invariant. See [Pen2012].

OUTPUT:

Integer.

EXAMPLES:

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

character()

Return the character of the ternary matroid.

For a linear subspace V over $GF(3)$ with orthogonal basis $q_1, \ldots, q_k$ the character equals the product of $|q_i|$ modulo 3, where the product ranges over the i such that $|q_i|$ is not divisible by 3. The character does not depend on the choice of the orthogonal basis. The character of a ternary matroid equals the character of its cocycle-space, and is an invariant for ternary matroids. See [Pen2012].

OUTPUT:

Integer.

EXAMPLES:

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

characteristic()

Return the characteristic of the base ring of the matrix representing the matroid, in this case 3.

EXAMPLES:

```
sage: M = matroids.named_matroids.NonFano()
sage: M.characteristic()
3
```

is_ternary(*randomized_tests=1*)

Decide if `self` is a binary matroid.

INPUT:

- `randomized_tests` – Ignored.

OUTPUT:

A Boolean.

ALGORITHM:

`self` is a ternary matroid, so just return `True`.

See also:

M.is_ternary()

EXAMPLES:

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

is_valid()

Test if the data obey the matroid axioms.

Since this is a linear matroid over the field $\mathbf{F}_3$, this is always the case.

OUTPUT:

`True`.

EXAMPLES:

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

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

Return a ternary matroid representing `self`.

INPUT:

- `randomized_tests` – Ignored.
- `verify` – Ignored

OUTPUT:

A binary matroid.

ALGORITHM:

`self` is a ternary matroid, so just return `self`.

See also:

M.ternary_matroid()

EXAMPLES:

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

3.4 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.

3.4.1 Constructions

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

EXAMPLES:

```
sage: def f(X):
....:     return min(len(X), 3)
....:
sage: M = Matroid(groundset=range(6), rank_function=f)
sage: M.is_valid()
True
sage: M.is_isomorphic(matroids.Uniform(3, 6))
True

sage: def g(X):
....:     if len(X) >= 3:
....:         return 1
....:     else:
```

```
....:         return 0
....:
sage: N = Matroid(groundset='abc', rank_function=g)
sage: N.is_valid()
False
```

See *below* for more. It is recommended to use the *Matroid()* function for easy construction of a RankMatroid. For direct access to the RankMatroid constructor, run:

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

AUTHORS:

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

3.4.2 Methods

class sage.matroids.rank_matroid.**RankMatroid**(*groundset*, *rank_function*)

Bases: *sage.matroids.matroid.Matroid*

Matroid specified by its rank function.

INPUT:

- groundset – the groundset of a matroid.
- rank_function – a function mapping subsets of groundset to nonnegative integers.

OUTPUT:

A matroid on groundset whose rank function equals rank_function

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: def f(X):
....:     return min(len(X), 3)
....:
sage: M = RankMatroid(groundset=range(6), rank_function=f)
sage: M.is_valid()
True
sage: M.is_isomorphic(matroids.Uniform(3, 6))
True
```

groundset()

Return the groundset of self.

EXAMPLES:

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

3.5 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 ground set E and circuits C.

3.5.1 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 *
```

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
sage: 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 ground set. The user may wish to override this by specifying the ground set, as the vertex tuples will not be updated if the matroid is modified.

> sage: G = graphs.DiamondGraph() sage: M1 = Matroid(G) sage: N1 = M1.contract((0,1)) sage: N1.graph().edges_incident(0) [(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

3.5.2 Methods

class sage.matroids.graphic_matroid.**GraphicMatroid**(*G*, *groundset=None*)

Bases: *sage.matroids.matroid.Matroid*

The graphic matroid class.

INPUT:

- `G` – a Graph
- `groundset` – (optional) a list in 1-1 correspondence with `G.edge_iterator()`

OUTPUT:

A `GraphicMatroid` instance where the ground set elements are the edges of `G`.

..NOTE:

```
If a disconnected graph is given as input, the instance of
``GraphicMatroid`` will connect the graph components and store
this as its graph.
```

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = GraphicMatroid(graphs.BullGraph()); M
Graphic matroid of rank 4 on 5 elements
sage: N = GraphicMatroid(graphs.CompleteBipartiteGraph(3,3)); N
Graphic matroid of rank 5 on 9 elements
```

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

```
sage: G1 = graphs.CycleGraph(3); G2 = graphs.DiamondGraph()
sage: G = G1.disjoint_union(G2)
sage: len(G)
7
sage: G.is_connected()
False
sage: M = GraphicMatroid(G)
sage: M
Graphic matroid of rank 5 on 8 elements
sage: H = M.graph()
sage: H
Looped multi-graph on 6 vertices
sage: H.is_connected()
True
sage: M.is_connected()
False
```

You can still locate an edge using the vertices of the input graph:

```
sage: G1 = graphs.CycleGraph(3); G2 = graphs.DiamondGraph()
sage: G = G1.disjoint_union(G2)
sage: M = Matroid(G)
sage: H = M.graph()
sage: vm = M.vertex_map()
sage: (u, v, l) = G.random_edge()
sage: H.has_edge(vm[u], vm[v])
True
```

graph()

Return the graph that represents the matroid.

The graph will always have loops and multiedges enabled.

OUTPUT:

A Graph.

EXAMPLES:

```
sage: M = Matroid(Graph([(0, 1, 'a'), (0, 2, 'b'), (0, 3, 'c')]))
sage: M.graph().edges()
[(0, 1, 'a'), (0, 2, 'b'), (0, 3, 'c')]
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
- `element` – (optional) The name of the newly added element

OUTPUT:

An instance of GraphicMatroid coextended by the new element. If `X` is not specified, the new element will be a coloop.

Note: A loop on `u` will stay a loop unless it is in `X`.

EXAMPLES:

```
sage: G = Graph([(0, 1, 0), (0, 2, 1), (0, 3, 2), (0, 4, 3), (1, 2, 4), (1, 4,
↪ 5), (2, 3, 6), (3, 4, 7)])
sage: M = Matroid(G)
sage: M1 = M.graphic_coextension(0, X=[1,2], element='a')
sage: M1.graph().edges()
[(0, 1, 0),
 (0, 4, 3),
 (0, 5, 'a'),
 (1, 2, 4),
 (1, 4, 5),
 (2, 3, 6),
 (2, 5, 1),
 (3, 4, 7),
 (3, 5, 2)]
```

```
sage: M = Matroid(graphs.CycleGraph(3))
sage: M = M.graphic_coextension(u=2, element='a')
sage: M.graph()
Looped multi-graph on 4 vertices
sage: M.graph().loops()
[]
sage: M = M.graphic_coextension(u=2, element='a')
Traceback (most recent call last):
...
ValueError: cannot extend by element already in ground set
sage: M = M.graphic_coextension(u=4)
Traceback (most recent call last):
...
ValueError: u must be an existing vertex
```

```
sage: M = Matroid(range(5), graphs.DiamondGraph())
sage: N = M.graphic_coextension(u=3, v=5, element='a')
sage: N.graph().edges()
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4), (3, 5, 'a')]
sage: N = M.graphic_coextension(u=3, element='a')
sage: N.graph().edges()
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4), (3, 4, 'a')]
sage: N = M.graphic_coextension(u=3, v=3, element='a')
Traceback (most recent call last):
...
ValueError: u and v must be distinct
```

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: for N in I:
....:     N.graph().edges_incident(0)
[(0, 1, 0), (0, 2, 1), (0, 3, 2), (0, 4, 3), (0, 5, 'a')]
```

```
[(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, 3, 2), (0, 4, 3), (0, 5, 'a')]
[(0, 1, 0), (0, 2, 1), (0, 4, 3), (0, 5, 'a')]
[(0, 2, 1), (0, 3, 2), (0, 5, 'a')]
[(0, 1, 0), (0, 3, 2), (0, 5, 'a')]
[(0, 1, 0), (0, 2, 1), (0, 5, 'a')]
```

```
sage: N = Matroid(range(4), graphs.CycleGraph(4))
sage: I = N.graphic_coextensions(element='a')
sage: for N1 in I:
....:     N1.graph().edges()
[(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
```

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` – a 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 specifies 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: M1.graph().edges()
[(0, 1, 0), (0, 1, 'a'), (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
```

```
sage: M = Matroid(range(10), graphs.PetersenGraph())
sage: M.graphic_extension(0, 'b', 'c').graph().vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'b']
sage: M.graphic_extension('a', 'b', 'c').graph().vertices()
Traceback (most recent call last):
...
ValueError: u must be an existing vertex
```

graphic_extensions(*element=None*, *vertices=None*, *simple=False*)

Return an iterable containing the graphic extensions.

This method iterates over the vertices in the input. If `simple == False`, it first extends by a loop. It will then add an edge between every pair of vertices in the input, skipping pairs of vertices with an edge already between them if `simple == True`.

This method only considers the current graph presentation, and does not take 2-isomorphism into account. Use *twist* or *one_sum* if you wish to change the graph presentation.

INPUT:

- element – (optional) the name of the newly added element in each extension
- vertices – (optional) a set of vertices over which the extension may be taken
- simple – (default: False) if true, extensions by loops and parallel elements are not taken

OUTPUT:

An iterable containing instances of GraphicMatroid. If vertices is not specified, every vertex is used.

Note: The extension by a loop will always occur unless simple == True. The extension by a coloop will never occur.

EXAMPLES:

```
sage: M = Matroid(range(5), graphs.DiamondGraph())
sage: I = M.graphic_extensions('a')
sage: for N in I:
....:     N.graph().edges()
[(0, 0, 'a'), (0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 1, 'a'), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (0, 2, 'a'), (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, 2), (1, 2, 'a'), (1, 3, 3), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (1, 3, 'a'), (2, 3, 4)]
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 3, 4), (2, 3, 'a')]
```

```
sage: M = Matroid(graphs.CompleteBipartiteGraph(3,3))
sage: I = M.graphic_extensions(simple=True)
sage: sum (1 for i in I)
6
sage: I = M.graphic_extensions(vertices=[0,1,2])
sage: sum (1 for i in I)
4
```

groundset()

Return the ground set of the matroid as a frozenset.

EXAMPLES:

```
sage: M = Matroid(graphs.DiamondGraph())
sage: sorted(M.groundset())
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]
sage: G = graphs.CompleteGraph(3).disjoint_union(graphs.CompleteGraph(4))
sage: M = Matroid(range(G.num_edges()), G); sorted(M.groundset())
[0, 1, 2, 3, 4, 5, 6, 7, 8]
sage: M = Matroid(Graph([(0, 1, 'a'), (0, 2, 'b'), (0, 3, 'c')]))
sage: sorted(M.groundset())
['a', 'b', 'c']
```

groundset_to_edges(X)

Return a list of edges corresponding to a set of ground set elements.

INPUT:

- X – a subset of the ground set

OUTPUT:

A list of graph edges.

EXAMPLES:

```
sage: M = Matroid(range(5), graphs.DiamondGraph())
sage: M.groundset_to_edges([2,3,4])
[(1, 2, 2), (1, 3, 3), (2, 3, 4)]
sage: M.groundset_to_edges([2,3,4,5])
Traceback (most recent call last):
...
ValueError: input must be a subset of the ground set
```

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:

```
sage: M = matroids.CompleteGraphic(4); M
M(K4): Graphic matroid of rank 3 on 6 elements
sage: 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 – a subset of the ground set
- u – a vertex spanned by the edges of the elements in X
- v – a vertex spanned by the edges of the elements not in X

OUTPUT:

An instance of `GraphicMatroid` isomorphic to this matroid but with a graph that is not necessarily isomorphic.

EXAMPLES:

```
sage: edgedict = {0:[1, 2], 1:[2, 3], 2:[3], 3:[4, 5], 6:[4, 5]}
sage: M = Matroid(range(9), Graph(edgedict))
sage: M.graph().edges()
[(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()
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (1, 3, 3),
 (1, 4, 5),
 (1, 5, 6),
 (2, 3, 4),
 (4, 6, 7),
 (5, 6, 8)]
sage: M2 = M.one_sum(u=4, v=3, X=[5, 6, 7, 8])
sage: M2.graph().edges()
[(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)]
```

```
sage: M = Matroid(range(5), graphs.BullGraph())
sage: M.graph().edges()
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (1, 3, 3), (2, 4, 4)]
sage: M1 = M.one_sum(u=3, v=0, X=[3,4])
Traceback (most recent call last):
...
ValueError: too many vertices in the intersection

sage: M1 = M.one_sum(u=3, v=2, X=[3])
sage: M1.graph().edges()
[(0, 1, 0), (0, 2, 1), (1, 2, 2), (2, 4, 4), (2, 5, 3)]

sage: M2 = M1.one_sum(u=5, v=0, X=[3,4])
sage: M2.graph().edges()
[(0, 1, 0), (0, 2, 1), (0, 3, 3), (1, 2, 2), (3, 4, 4)]

sage: M = Matroid(range(5), graphs.BullGraph())
sage: M.one_sum(u=0, v=1, X=[3])
Traceback (most recent call last):
...
ValueError: first vertex must be spanned by the input

sage: M.one_sum(u=1, v=3, X=[3])
Traceback (most recent call last):
...
ValueError: second vertex must be spanned by the rest of the graph
```

regular_matroid()

Return an instance of RegularMatroid isomorphic to this GraphicMatroid.

EXAMPLES:

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

subgraph_from_set(X)

Return the subgraph corresponding to the matroid restricted to X.

INPUT:

- `X` – a subset of the ground set

OUTPUT:

A 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 ground set
```

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:

An instance of `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()
[(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
```

vertex_map()

Return a dictionary mapping the input vertices to the current vertices.

The graph for the matroid is alway 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:

A 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()
[(0, 1, 0),
 (0, 2, 1),
 (1, 2, 2),
 (2, 4, 3),
 (2, 5, 4),
 (4, 5, 5),
 (5, 7, 6),
 (5, 8, 7),
 (7, 8, 8),
 (7, 8, 9),
 (8, 8, 10)]
sage: M.vertex_map()
{0: 0, 1: 1, 2: 2, 3: 2, 4: 4, 5: 5, 6: 5, 7: 7, 8: 8}
```

CHAPTER

FOUR

ABSTRACT MATROID CLASSES

4.1 Dual matroids

4.1.1 Theory

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

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: N = M.dual()
sage: M.is_basis('abc')
True
sage: N.is_basis('defg')
True
sage: M.dual().dual() == M
True
```

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 default implementation of the method *M.dual()*. For direct access to the `DualMatroid` constructor, run:

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

See also *sage.matroids.advanced*.

AUTHORS:

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

4.1.3 Methods

class sage.matroids.dual_matroid.**DualMatroid**(*matroid*)

Bases: *sage.matroids.matroid.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 default implementation.

INPUT:

- `matroid` - a matroid.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.named_matroids.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'}, {'e', 'f', 'g', 'h'},
     {'a', 'b', 'g', 'h'}, {'c', 'd', 'e', 'f'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}'
```

dual()

Return the dual of the matroid.

Let M be a matroid with ground set 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.named_matroids.Pappus().dual()
sage: N = M.dual()
sage: N.rank()
3
sage: N
Pappus: Matroid of rank 3 on 9 elements with circuit-closures
{2: {{'a', 'b', 'c'}, {'a', 'f', 'h'}, {'c', 'e', 'g'},
     {'b', 'f', 'g'}, {'c', 'd', 'h'}, {'d', 'e', 'f'},
     {'a', 'e', 'i'}, {'b', 'd', 'i'}, {'g', 'h', 'i'}},
 3: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'}}}
```

groundset()

Return the groundset of the matroid.

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

OUTPUT:

A set.

EXAMPLES:

```
sage: M = matroids.named_matroids.Pappus().dual()
sage: sorted(M.groundset())
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

4.2 Minors of matroids

4.2.1 Theory

Let M be a matroid with ground set E. There are two standard ways to remove an element from E so that the result is again a matroid, *deletion* and *contraction*. Deletion is simply omitting the elements from a set D from E and keeping all remaining independent sets. This is denoted `M \ D` (this also works in Sage). Contraction is the dual operation: `M / C == (M.dual() \ C).dual()`.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: M \ ['a', 'c' ] == M.delete(['a', 'c'])
True
sage: M / 'a' == M.contract('a')
True
sage: M / 'c' \ 'ab' == M.minor(contractions='c', deletions='ab')
True
```

If a contraction set is not independent (or a deletion set not coindependent), this is taken care of:

```
sage: M = matroids.named_matroids.Fano()
sage: M.rank('abf')
2
sage: M / 'abf' == M / 'ab' \ 'f'
True
sage: M / 'abf' == M / 'af' \ '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:

```
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: *sage.matroids.matroid.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` – a 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.named_matroids.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'}, {'e', 'f', 'g', 'h'},
     {'a', 'b', 'g', 'h'}, {'c', 'd', 'e', 'f'}},
 4: {{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}}}
```

groundset()

Return the groundset of the matroid.

EXAMPLES:

```
sage: M = matroids.named_matroids.Pappus().contract(['c'])
sage: 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

4.3.1 Methods

class sage.matroids.basis_exchange_matroid.**BasisExchangeMatroid**

Bases: *sage.matroids.matroid.Matroid*

Class BasisExchangeMatroid is a virtual class that derives from Matroid. It implements each of the elementary matroid methods (*rank()*, *max_independent()*, *circuit()*, *closure()* etc.), essentially by crawling the base exchange graph of the matroid. This is the graph whose vertices are the bases of the matroid, two bases being adjacent in the graph if their symmetric difference has 2 members.

This base exchange graph is not stored as such, but should be provided implicitly by the child class in the form of two methods `__is_exchange_pair(x, y)` and `__exchange(x, y)`, as well as an initial basis. At any moment, BasisExchangeMatroid keeps a current basis B. The method `__is_exchange_pair(x, y)` should return a boolean indicating whether $B - x + y$ is a basis. The method `__exchange(x, y)` is called when the current basis B is replaced by said $B - x + y$. It is up to the child class to update its internal data structure to make information relative to the new basis more accessible. For instance, a linear matroid would perform a row reduction to make the column labeled by y a standard basis vector (and therefore the columns indexed by $B - x + y$ would form an identity matrix).

Each of the elementary matroid methods has a straightforward greedy-type implementation in terms of these two methods. For example, given a subset F of the groundset, one can step to a basis B over the edges of the base exchange graph which has maximal intersection with F, in each step increasing the intersection of the current B with F. Then one computes the rank of F as the cardinality of the intersection of F and B.

The following matroid classes can/will implement their oracle efficiently by deriving from `BasisExchangeMatroid`:

- *BasisMatroid*: keeps a list of all bases.
  - `__is_exchange_pair(x, y)` reduces to a query whether $B - x + y$ is a basis.
  - `__exchange(x, y)` has no work to do.
- *LinearMatroid*: keeps a matrix representation A of the matroid so that $A[B] = I$.
  - `__is_exchange_pair(x, y)` reduces to testing whether $A[r, y]$ is nonzero, where $A[r, x] = 1$.
  - `__exchange(x, y)` should modify the matrix so that $A[B - x + y]$ becomes I, which means pivoting on $A[r, y]$.
- `TransversalMatroid` (not yet implemented): If A is a set of subsets of E, then I is independent if it is a system of distinct representatives of A, i.e. if I is covered by a matching of an appropriate bipartite graph G, with color classes A and E and an edge (A_i, e) if e is in the subset A_i. At any time you keep a maximum matching M of G covering the current basis B.
  - `__is_exchange_pair(x, y)` checks for the existence of an M-alternating path P from y to x.
  - `__exchange(x, y)` replaces M by the symmetric difference of M and $E(P)$.
- `AlgebraicMatroid` (not yet implemented): keeps a list of polynomials in variables $E - B + e$ for each variable e in B.
  - `__is_exchange_pair(x, y)` checks whether the polynomial that relates y to $E - B$ uses x.
  - `__exchange(x, y)` make new list of polynomials by computing resultants.

All but the first of the above matroids are algebraic, and all implementations specializations of the algebraic one.

BasisExchangeMatroid internally renders subsets of the ground set as bitsets. It provides optimized methods for enumerating bases, nonbases, flats, circuits, etc.

bases()

Return the list of bases of the matroid.

A *basis* is a maximal independent set.

OUTPUT:

An iterable containing all bases of the matroid.

EXAMPLES:

```
sage: M = matroids.named_matroids.N1()
sage: M.bases_count()
184
sage: len([B for B in M.bases()])
184
```

bases_count()

Return the number of bases of the matroid.

A *basis* is an inclusionwise maximal independent set.

See also:

M.basis().

OUTPUT:

Integer.

EXAMPLES:

```
sage: M = matroids.named_matroids.N1()
sage: 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.named_matroids.Fano()
sage: sorted(M.basis())
['a', 'b', 'c']
sage: M.rank('cd')
2
sage: sorted(M.basis())
['a', 'c', 'd']
```

circuits()
Return the list of circuits of the matroid.

OUTPUT:

An iterable containing all circuits.

See also:

M.circuit()

EXAMPLES:

```
sage: M = Matroid(matroids.named_matroids.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']]
```

cocircuits()
Return the list of cocircuits of the matroid.

OUTPUT:

An iterable containing all cocircuits.

See also:

Matroid.cocircuit()

EXAMPLES:

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

coflats(*r*)
Return the collection of coflats of the matroid of specified corank.

A *coflat* is a coclosed set.

INPUT:

- `r` – A natural number.

OUTPUT:

An iterable containing all coflats of corank `r`.

See also:

Matroid.coclosure()

EXAMPLES:

```
sage: M = matroids.named_matroids.S8().dual()
sage: M.f_vector()
[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:

A list of subsets.

See also:

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

EXAMPLES:

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

dependent_r_sets(*r*)

Return the list of dependent subsets of fixed size.

INPUT:

- r – a nonnegative integer.

OUTPUT:

An iterable containing all dependent subsets of size r.

EXAMPLES:

```
sage: M = matroids.named_matroids.N1()
sage: len(M.nonbases())
68
sage: [len(M.dependent_r_sets(r)) for r in range(M.full_rank() + 1)]
[0, 0, 0, 0, 9, 68]
```

f_vector()

Return the f-vector of the matroid.

The *f-vector* is a vector $(f_0, ..., f_r)$, where f_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.named_matroids.S8()
sage: M.f_vector()
[1, 8, 22, 14, 1]
```

flats(*r*)

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

A *flat* is a closed set.

INPUT:

- `r` – A natural number.

OUTPUT:

An iterable containing all flats of rank `r`.

See also:

Matroid.closure()

EXAMPLES:

```
sage: M = matroids.named_matroids.S8()
sage: M.f_vector()
[1, 8, 22, 14, 1]
sage: len(M.flats(2))
22
sage: len(M.flats(8))
0
sage: len(M.flats(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.named_matroids.Fano()
sage: M.full_corank()
4
sage: 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.named_matroids.Fano()
sage: M.full_rank()
3
```

```
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:

A set.

EXAMPLES:

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

A list.

See also:

M.groundset()

EXAMPLES:

```
sage: M = matroids.named_matroids.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']
```

independent_r_sets(*r*)
Return the list of size-`r` independent subsets of the matroid.

INPUT:

- `r` – a nonnegative integer.

OUTPUT:

An iterable containing all independent subsets of the matroid of cardinality `r`.

EXAMPLES:

```
sage: M = matroids.named_matroids.N1()
sage: M.bases_count()
184
```

```
sage: [len(M.independent_r_sets(r)) for r in range(M.full_rank() + 1)]
[1, 10, 45, 120, 201, 184]
```

independent_sets()

Return the list of independent subsets of the matroid.

OUTPUT:

An iterable containing all independent subsets of the matroid.

EXAMPLES:

```
sage: M = matroids.named_matroids.Fano()
sage: I = M.independent_sets()
sage: len(I)
57
```

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:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid(matroids.named_matroids.Fano())
sage: M.is_valid()
True
sage: M = Matroid(groundset='abcd', bases=['ab', 'cd'])
sage: M.is_valid()
False
```

nonbases()

Return the list of nonbases of the matroid.

A *nonbasis* is a set with cardinality `self.full_rank()` that is not a basis.

OUTPUT:

An iterable containing the nonbases of the matroid.

See also:

Matroid.basis()

EXAMPLES:

```
sage: M = matroids.named_matroids.N1()
sage: binomial(M.size(), M.full_rank())-M.bases_count()
68
sage: len([B for B in M.nonbases()])
68
```

noncospanning_cocircuits()
Return the list of noncospanning cocircuits of the matroid.

A *noncospanning cocircuit* is a cocircuit whose corank is strictly smaller than the corank of the matroid.

OUTPUT:

An iterable containing all nonspanning circuits.

See also:

Matroid.cocircuit(), *Matroid.corank()*

EXAMPLES:

```
sage: M = matroids.named_matroids.N1()
sage: len(M.noncospanning_cocircuits())
23
```

nonspanning_circuits()
Return the list of nonspanning circuits of the matroid.

A *nonspanning circuit* is a circuit whose rank is strictly smaller than the rank of the matroid.

OUTPUT:

An iterable containing all nonspanning circuits.

See also:

Matroid.circuit(), *Matroid.rank()*

EXAMPLES:

```
sage: M = matroids.named_matroids.N1()
sage: len(M.nonspanning_circuits())
23
```

CHAPTER

FIVE

ADVANCED FUNCTIONALITY

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 *
```

This adds the following to the main namespace:

- **Matroid classes:**
  - *MinorMatroid*
  - *DualMatroid*
  - *RankMatroid*
  - *CircuitClosuresMatroid*
  - *BasisMatroid*
  - *LinearMatroid*
  - *RegularMatroid*
  - *BinaryMatroid*
  - *TernaryMatroid*
  - *QuaternaryMatroid*
  - GraphicMatroid

  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()

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:

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

See also *sage.matroids.advanced*.

AUTHORS:

- Stefan van Zwam (2011-06-24): initial version

sage.matroids.utilities.**get_nonisomorphic_matroids**(*MSet*)

Return non-isomorphic members of the matroids in set MSet.

For direct access to get_nonisomorphic_matroids, run:

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

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
```

sage.matroids.utilities.**lift_cross_ratios**(*A*, *lift_map=None*)

Return a matrix which arises from the given matrix by lifting cross ratios.

INPUT:

- A – a matrix over a ring source_ring.
- lift_map – a python dictionary, mapping each cross ratio of A to some element of a target ring, and such that lift_map[source_ring(1)] = target_ring(1).

OUTPUT:

- Z – a matrix over the ring target_ring.

The intended use of this method is to create a (reduced) matrix representation of a matroid `M` over a ring `target_ring`, given a (reduced) matrix representation of `A` of `M` over a ring `source_ring` and a map `lift_map` from `source_ring` to `target_ring`.

This method will create a unique candidate representation `Z`, but will not verify if `Z` is indeed a representation of `M`. However, this is guaranteed if the conditions of the lift theorem (see [PvZ2010]) hold for the lift map in combination with the matrix `A`.

For a lift map f and a matrix A these conditions are as follows. First of all $f : S \to T$, where S is a set of invertible elements of the source ring and T is a set of invertible elements of the target ring. The matrix A has entries from the source ring, and each cross ratio of A is contained in S. Moreover:

- $1 \in S, 1 \in T$;
- for all $x \in S$: $f(x) = 1$ if and only if $x = 1$;
- for all $x, y \in S$: if $x + y = 0$ then $f(x) + f(y) = 0$;
- for all $x, y \in S$: if $x + y = 1$ then $f(x) + f(y) = 1$;
- for all $x, y, z \in S$: if $xy = z$ then $f(x)f(y) = f(z)$.

Any ring homomorphism $h : P \to R$ induces a lift map from the set of units S of P to the set of units T of R. There exist lift maps which do not arise in this manner. Several such maps can be created by the function *lift_map()*.

See also:

lift_map()

EXAMPLES:

```
sage: from sage.matroids.advanced import lift_cross_ratios, lift_map,␣
↪LinearMatroid
sage: R = GF(7)
sage: to_sixth_root_of_unity = lift_map('sru')
sage: A = Matrix(R, [[1, 0, 6, 1, 2],[6, 1, 0, 0, 1],[0, 6, 3, 6, 0]])
sage: A
[1 0 6 1 2]
[6 1 0 0 1]
[0 6 3 6 0]
sage: Z = lift_cross_ratios(A, to_sixth_root_of_unity)
sage: Z
[ 1  0  1  1  1]
[ 1  1  0  0  z]
[ 0  z - 1  1  -z + 1  0]
sage: M = LinearMatroid(reduced_matrix = A)
sage: sorted(M.cross_ratios())
[3, 5]
sage: N = LinearMatroid(reduced_matrix = Z)
sage: sorted(N.cross_ratios())
[-z + 1, z]
sage: M.is_isomorphism(N, {e:e for e in M.groundset()})
True
```

sage.matroids.utilities.**lift_map**(*target*)

Create a lift map, to be used for lifting the cross ratios of a matroid representation.

See also:

lift_cross_ratios()

INPUT:

- `target` – 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 $\mathbf{F}_3$ to the regular partial field $(\mathbf{Z}, < -1 >)$.
- "sru": a lift map from $\mathbf{F}_7$ to the sixth-root-of-unity partial field $(\mathbf{Q}(z), < z >)$, where z is a sixth root of unity. The map sends 3 to z.
- "dyadic": a lift map from $\mathbf{F}_{11}$ to the dyadic partial field $(\mathbf{Q}, < -1, 2 >)$.
- "gm": a lift map from $\mathbf{F}_{19}$ to the golden mean partial field $(\mathbf{Q}(t), < -1, t >)$, where t is a root of $t^2 - t - 1$. The map sends 5 to t.

The example below shows that the latter map satisfies three necessary conditions stated in *lift_cross_ratios()*

EXAMPLES:

```
sage: from sage.matroids.utilities import lift_map
sage: lm = lift_map('gm')
sage: for x in lm:
....:     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:

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

ALGORITHM:

1. Create a set of all one-character alphanumeric strings.
2. Remove the string representation of each existing element from this list.
3. If the list is nonempty, return any element.
4. Otherwise, return the concatenation of the strings of each existing element, preceded by 'e'.

EXAMPLES:

```
sage: from sage.matroids.advanced import newlabel
sage: S = set(['a', 42, 'b'])
sage: newlabel(S) in S
False

sage: S = set('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789')
sage: t = newlabel(S)
sage: len(t)
63
sage: t[0]
'e'
```

sage.matroids.utilities.**sanitize_contractions_deletions**(*matroid*, *contractions*, *deletions*)

Return a fixed version of sets `contractions` and `deletions`.

INPUT:

- `matroid` – a *Matroid* instance.
- `contractions` – a subset of the groundset.
- `deletions` – a subset of the groundset.

OUTPUT:

An independent set `C` and a coindependent set `D` such that

```
matroid / contractions \ deletions == matroid / C \ D
```

Raise an error if either is not a subset of the groundset of `matroid` or if they are not disjoint.

This function is used by the *Matroid.minor()* method.

EXAMPLES:

```
sage: from sage.matroids.utilities import setprint
sage: from sage.matroids.utilities import sanitize_contractions_deletions
sage: M = matroids.named_matroids.Fano()
sage: setprint(sanitize_contractions_deletions(M, 'abc', 'defg'))
[{'a', 'b', 'c'}, {'d', 'e', 'f', 'g'}]
sage: setprint(sanitize_contractions_deletions(M, 'defg', 'abc'))
[{'d', 'e', 'g'}, {'a', 'b', 'c', 'f'}]
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.
```

sage.matroids.utilities.**setprint**(*X*)

Print nested data structures nicely.

Python's data structures `set` and `frozenset` do not print nicely. This function can be used as replacement for `print` to overcome this. For direct access to `setprint`, run:

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

Note: This function will be redundant when Sage moves to Python 3, since the default `print` will suffice then.

INPUT:

- `X` – Any Python object

OUTPUT:

`None`. However, the function prints a nice representation of `X`.

EXAMPLES:

Output looks much better:

```
sage: from sage.matroids.advanced import setprint
sage: L = [{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {4, 1, 3}]
sage: print(L)
[set([1, 2, 3]), set([1, 2, 4]), set([2, 3, 4]), set([1, 3, 4])]
sage: setprint(L)
[{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {1, 3, 4}]
```

Note that for iterables, the effect can be undesirable:

```
sage: from sage.matroids.advanced import setprint
sage: M = matroids.named_matroids.Fano().delete('efg')
sage: M.bases()
Iterator over a system of subsets
sage: setprint(M.bases())
[{'a', 'b', 'c'}, {'a', 'c', 'd'}, {'a', 'b', 'd'}]
```

An exception was made for subclasses of SageObject:

```
sage: from sage.matroids.advanced import setprint
sage: G = graphs.PetersenGraph()
sage: list(G)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: setprint(G)
Petersen graph: Graph on 10 vertices
```

sage.matroids.utilities.**setprint_s**(*X*, *toplevel=False*)
Create the string for use by `setprint()`.

INPUT:

- `X` – any Python object
- `toplevel` – (default: `False`) indicates whether this is a recursion or not.

OUTPUT:

A string representation of the object, with nice notation for sets and frozensets.

EXAMPLES:

```
sage: from sage.matroids.utilities import setprint_s
sage: L = [{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {4, 1, 3}]
sage: setprint_s(L)
'[{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {1, 3, 4}]'
```

The `toplevel` argument only affects strings, to mimic `print`'s behavior:

```
sage: X = 'abcd'
sage: setprint_s(X)
"'abcd'"
sage: setprint_s(X, toplevel=True)
'abcd'
```

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:

```
sage: len(sage.matroids.utilities.spanning_forest(matrix([[1,1,1],[1,1,1],[1,1,
↪1]])))
5
sage: len(sage.matroids.utilities.spanning_forest(matrix([[0,0,1],[0,1,0],[0,1,
↪0]])))
3
```

sage.matroids.utilities.**spanning_stars**(*M*)
Returns 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:

```
sage: edges = sage.matroids.utilities.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
```

sage.matroids.utilities.**split_vertex**(*G*, *u*, *v=None*, *edges=None*)

Split a vertex in a graph.

If an edge is inserted between the vertices after splitting, this corresponds to a graphic coextension of a matroid.

INPUT:

- `G` – A SageMath Graph.
- `u` – A vertex in `G`.
- `v` – (optional) The name of the new vertex after the splitting. If `v` is specified and already in the graph, it must be an isolated vertex.
- `edges` – (optional) 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:

```
sage: from sage.matroids.utilities import split_vertex
sage: G = graphs.BullGraph()
sage: split_vertex(G, u = 1, v = 'a', edges = [(1, 3)])
Traceback (most recent call last):
...
ValueError: the edges are not all incident with u
sage: split_vertex(G, u = 1, v = 'a', edges = [(1, 3, None)])
sage: G.edges()
[(0, 1, None), (0, 2, None), (1, 2, None), (2, 4, None), (3, 'a', 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 *
```

See also *sage.matroids.advanced*.

AUTHORS:

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

5.3.1 Methods

class sage.matroids.extension.**CutNode**

Bases: `object`

An internal class used for creating linear subclasses of a matroids in a depth-first manner.

A linear subclass is a set of hyperplanes mc with the property that certain sets of hyperplanes must either be fully contained in mc or intersect mc in at most 1 element. The way we generate them is by a depth-first seach. 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 seach tree.

The class has methods for selecting a hyperplane to insert, for inserting hyperplanes and closing the set to become a linear subclass again, and for adding a hyperplane to the set of *forbidden* hyperplanes, and similarly closing that set.

class sage.matroids.extension.**LinearSubclasses**

Bases: object

An iterable set of linear subclasses of a matroid.

Enumerate linear subclasses of a given matroid. A *linear subclass* is a collection of hyperplanes (flats of rank $r-1$ where r is the rank of the matroid) with the property that no modular triple of hyperplanes has exactly two members in the linear subclass. A triple of hyperplanes in a matroid of rank r is *modular* if its intersection has rank $r-2$.

INPUT:

- `M` – 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:

```
sage: from sage.matroids.extension import LinearSubclasses
sage: M = matroids.Uniform(3, 6)
sage: len([mc for mc in LinearSubclasses(M)])
83
sage: len([mc for mc in LinearSubclasses(M, line_length=5)])
22
sage: for mc in LinearSubclasses(M, subsets=[[0, 1], [2, 3], [4, 5]]):
....:     print(len(mc))
3
15
```

Note that this class is intended for runtime, internal use, so no loads/dumps mechanism was implemented.

class sage.matroids.extension.**LinearSubclassesIter**

Bases: object

An iterator for a set of linear subclass.

next()

x.next() -> the next value, or raise StopIteration

class sage.matroids.extension.**MatroidExtensions**

Bases: *sage.matroids.extension.LinearSubclasses*

An iterable set of single-element extensions of a given matroid.

INPUT:

- `M` – a matroid
- `e` – an element
- `line_length` (default: `None`) – an integer
- `subsets` (default: `None`) – a set of subsets of the groundset of `M`
- `splice` – a matroid N such that for some $f \in E(M)$, we have $N \setminus e = M \setminus f$.

OUTPUT:

An enumerator for the extensions of `M` to a matroid `N` so that $N \setminus e = M$. If `line_length` is not `None`, the enumeration is restricted to extensions N without $U(2, k)$-minors, where `k > line_length`.

If `subsets` is not `None`, the enumeration is restricted to extensions N of M by element e so that all hyperplanes of M which fully contain some set from `subsets`, will also span e.

If `splice` is not `None`, then the enumeration is restricted to extensions M' such that $M' \setminus f = N$, where $E(M) \setminus E(N) = \{f\}$.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = matroids.Uniform(3, 6)
sage: len([N for N in MatroidExtensions(M, 'x')])
83
sage: len([N for N in MatroidExtensions(M, 'x', line_length=5)])
22
sage: for N in MatroidExtensions(M, 'x', subsets=[[0, 1], [2, 3],
....:                                             [4, 5]]): print(N)
Matroid of rank 3 on 7 elements with 32 bases
Matroid of rank 3 on 7 elements with 20 bases
sage: M = BasisMatroid(matroids.named_matroids.BetsyRoss()); M
Matroid of rank 3 on 11 elements with 140 bases
sage: e = 'k'; f = 'h'; Me = M.delete(e); Mf=M.delete(f)
sage: for N in MatroidExtensions(Mf, f, splice=Me): print(N)
Matroid of rank 3 on 11 elements with 141 bases
Matroid of rank 3 on 11 elements with 140 bases
sage: for N in MatroidExtensions(Me, e, splice=Mf): print(N)
Matroid of rank 3 on 11 elements with 141 bases
Matroid of rank 3 on 11 elements with 140 bases
```

Note that this class is intended for runtime, internal use, so no loads/dumps mechanism was implemented.

CHAPTER

SIX

INTERNALS

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

class sage.matroids.lean_matrix.**BinaryMatrix**

Bases: *sage.matroids.lean_matrix.LeanMatrix*

Binary matrix class. Entries are stored bit-packed into integers.

INPUT:

- `m` – Number of rows.
- `n` – Number of columns.
- `M` – (default: `None`) Matrix or BinaryMatrix instance. Assumption: dimensions of `M` are at most `m` by `n`.
- `ring` – (default: `None`) ignored.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(2, 2, Matrix(GF(7), [[0, 0], [0, 0]]))
sage: B = BinaryMatrix(2, 2, ring=GF(5))
sage: C = BinaryMatrix(2, 2)
sage: A == B and A == C
True
```

base_ring()

Return $GF(2)$.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(4, 4)
sage: A.base_ring()
Finite Field of size 2
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(3, 4)
sage: A.characteristic()
2
```

class sage.matroids.lean_matrix.**GenericMatrix**

Bases: *sage.matroids.lean_matrix.LeanMatrix*

Matrix over arbitrary Sage ring.

INPUT:

- `nrows` – number of rows
- `ncols` – number of columns
- `M` – (default: `None`) a `Matrix` or `GenericMatrix` of dimensions at most `m*n`.
- `ring` – (default: `None`) a Sage ring.

Note: This class is intended for internal use by the LinearMatroid class only. Hence it does not derive from `SageObject`. If `A` is a LeanMatrix instance, and you need access from other parts of Sage, use `Matrix(A)` instead.

If the constructor is fed a GenericMatrix instance, the `ring` argument is ignored. Otherwise, the matrix entries will be converted to the appropriate ring.

EXAMPLES:

```
sage: M = Matroid(ring=GF(5), matrix=[[1, 0, 1, 1, 1], [0, 1, 1, 2, 3]])  #␣
↪indirect doctest
sage: M.is_isomorphic(matroids.Uniform(2, 5))
True
```

base_ring()

Return the base ring of `self`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import GenericMatrix
sage: A = GenericMatrix(3, 4, ring=GF(5))
sage: A.base_ring()
Finite Field of size 5
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import GenericMatrix
sage: A = GenericMatrix(3, 4, ring=GF(5))
sage: A.characteristic()
5
```

class sage.matroids.lean_matrix.**IntegerMatrix**

Bases: *sage.matroids.lean_matrix.LeanMatrix*

Matrix over the integers.

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 small integers. No overflow checking takes place!

EXAMPLES:

```
sage: M = Matroid(graphs.CompleteGraph(4).incidence_matrix(oriented=True),
....:             regular=True)  # indirect doctest
sage: M.is_isomorphic(matroids.Wheel(3))
True
```

base_ring()

Return the base ring of `self`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = IntegerMatrix(3, 4)
sage: A.base_ring()
Integer Ring
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = GenericMatrix(3, 4)
sage: A.characteristic()
0
```

class sage.matroids.lean_matrix.**LeanMatrix**

Bases: `object`

Lean matrices

Sage's matrix classes are powerful, versatile, and often very fast. However, their performance with regard to pivoting (pretty much the only task performed on them within the context of matroids) leaves something to be

desired. The LeanMatrix classes provide the LinearMatroid classes with a custom, light-weight data structure to store and manipulate matrices.

This is the abstract base class. Most methods are not implemented; this is only to fix the interface.

Note: This class is intended for internal use by the LinearMatroid class only. Hence it does not derive from `SageObject`. If `A` is a LeanMatrix instance, and you need access from other parts of Sage, use `Matrix(A)` instead.

EXAMPLES:

```
sage: M = Matroid(ring=GF(5), matrix=[[1, 0, 1, 1, 1], [0, 1, 1, 2, 3]])  #␣
↪indirect doctest
sage: M.is_isomorphic(matroids.Uniform(2, 5))
True
```

base_ring()
Return the base ring.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import LeanMatrix
sage: A = LeanMatrix(3, 4)
sage: A.base_ring()
Traceback (most recent call last):
...
NotImplementedError: subclasses need to implement this.
```

characteristic()
Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import GenericMatrix
sage: A = GenericMatrix(3, 4, ring=GF(5))
sage: A.characteristic()
5
```

ncols()
Return the number of columns.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import LeanMatrix
sage: A = LeanMatrix(3, 4)
sage: 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
```

class sage.matroids.lean_matrix.**QuaternaryMatrix**

Bases: *sage.matroids.lean_matrix.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:

```
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
```

base_ring()

Return copy of $GF(4)$ with appropriate generator.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = QuaternaryMatrix(2, 2, ring=GF(4, 'f'))
sage: A.base_ring()
Finite Field in f of size 2^2
```

characteristic()

Return the characteristic of `self.base_ring()`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = QuaternaryMatrix(200, 5000, ring=GF(4, 'x'))
sage: A.characteristic()
2
```

class sage.matroids.lean_matrix.**TernaryMatrix**

Bases: *sage.matroids.lean_matrix.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
```

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
```

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
```

sage.matroids.lean_matrix.**generic_identity**(*n*, *ring*)
Return a GenericMatrix instance containing the $nimesn$ identity matrix over `ring`.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = generic_identity(2, QQ)
sage: Matrix(A)
[1 0]
[0 1]
```

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 `createline` function to produce sequence of `100` points on a smooth curve containing the points in the specified line which inturn uses `scipy.interpolate.splprep` and `scipy.interpolate.splev`. Then one can use sage's graphics primitives `line`, `point`, `text` and `points` to produce graphics object containing points (ground

set elements) and lines (for a rank 3 matroid, these are flats of rank 2 of size greater than equal to 3) of the geometric representation of the matroid. Loops and parallel elements are added as per conventions in [Oxl2011] using function `addlp`. The priority order for point placement methods used inside plot() and show() is as follows:

1. User Specified points dictionary and lineorders
2. cached point placement dictionary and line orders (a list of ordered lists) in M._cached_info (a dictionary)
3. Internal point placement and orders deciding heuristics If a custom point placement and/or line orders is desired, then user can simply specify the custom points dictionary as:

```
M.cached info = {'plot_positions':<dictionary_of _points>,
                 'plot_lineorders':<list of lists>}
```

6.2.1 REFERENCES

- [Oxl2011] James Oxley, "Matroid Theory, Second Edition". Oxford University Press, 2011.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: M1=Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1,0,1],
....: [0, 1, 0, 1, 0, 1, 1,0,0,1,0], [0, 0, 1, 1, 1, 0, 1,0,0,0,0]])
sage: pos_dict= {0: (0, 0),  1: (2, 0),  2: (1, 2),  3: (1.5, 1.0),
....: 4: (0.5, 1.0),  5: (1.0, 0.0), 6: (1.0, 0.666666666666667),
....: 7: (3,3), 8: (4,0), 9: (-1,1), 10: (-2,-2)}
sage: M1._cached_info={'plot_positions': pos_dict, 'plot_lineorders': None}
sage: matroids_plot_helpers.geomrep(M1, sp=True)
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, 1,0,1],
....: [0, 1, 0, 1, 0, 1, 1,0,0],[0, 0, 1, 1, 1, 0, 1,0,0]])
sage: [M1,L,P]=matroids_plot_helpers.slp(M)
sage: G,lims=matroids_plot_helpers.addlp(M,M1,L,P,{0:(0,0)})
sage: G.show(axes=False)
```

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 allong with all keys and respective values in `ptsdict`.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: from sage.matroids.advanced import setprint
sage: ptsdict={'a':(0,0),'b':(1,2),'c':(2,0)}
sage: ptsdict_1=matroids_plot_helpers.addnontripts(['a','b','c'],
....:         ['d','e','f'],ptsdict)
sage: setprint(ptsdict_1)
{'a': [0, 0], 'b': [1, 2], 'c': [0, 2], 'd': [0.666666666666666, 1.0],
'e': [0.666666666666666, 0.8888888888888888],
'f': [0.8888888888888888, 1.3333333333333333]}
sage: ptsdict_2=matroids_plot_helpers.addnontripts(['a','b','c'],
....:         ['d','e','f','g','h'],ptsdict)
sage: setprint(ptsdict_2)
{'a': [0, 0], 'b': [1, 2], 'c': [0, 2], 'd': [0.666666666666666, 1.0],
'e': [0.666666666666666, 0.8888888888888888],
'f': [0.8888888888888888, 1.3333333333333333],
'g': [0.2222222222222222, 1.0],
'h': [0.5185185185185185, 0.5555555555555555]}
```

Note: This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**createline**(*ptsdict*, *ll*, *lineorders2=None*)

Return ordered lists of co-ordinates of points to be traversed to draw a 2D line.

INPUT:

- `ptsdict` – A dictionary containing keys and their (x,y) position as values.
- `ll` – A list of keys in `ptsdict` through which a line is to be drawn.

- `lineorders2`– (optional) A list of ordered lists of keys in `ptsdict` such that if ll is setwise same as any of these then points corresponding to values of the keys will be traversed in that order thus overriding internal order deciding heuristic.

OUTPUT:

A tuple containing 4 elements in this order:

1. Ordered list of x-coordinates of values of keys in `ll` specified in ptsdict.
2. Ordered list of y-coordinates of values of keys in `ll` specified in ptsdict.
3. Ordered list of interpolated x-coordinates of points through which a line can be drawn.
4. Ordered list of interpolated y-coordinates of points through which a line can be drawn.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: ptsdict={'a':(1,3),'b':(2,1),'c':(4,5),'d':(5,2)}
sage: x,y,x_i,y_i=matroids_plot_helpers.createline(ptsdict,
....: ['a','b','c','d'])
sage: [len(x), len(y), len(x_i), len(y_i)]
[4, 4, 100, 100]
sage: G = line(zip(x_i, y_i),color='black',thickness=3,zorder=1)
sage: G+=points(zip(x, y), color='black', size=300,zorder=2)
sage: G.show()
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)
sage: G+=points(zip(x, y), color='black', size=300,zorder=2)
sage: G.show()
```

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.grondset()` 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:

```
sage: from sage.matroids import matroids_plot_helpers
sage: M=matroids.named_matroids.P7()
sage: G=matroids_plot_helpers.geomrep(M)
sage: G.show(xmin=-2, xmax=3, ymin=-2, ymax=3)
sage: M=matroids.named_matroids.P7()
sage: G=matroids_plot_helpers.geomrep(M,lineorders1=[['f','e','d']])
sage: G.show(xmin=-2, xmax=3, ymin=-2, ymax=3)
```

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) co-ordinates with `M.simplify.groundset()` elements that can be placed on the sides of the triangle as keys.
2. A list of 3 lists of elements of `M.simplify.groundset()` that can be placed on the 3 sides of the triangle.
3. A list of elements of $M.simplify.groundset()$‘ that cane be placed inside the triangle in the geometric representation.
4. A list of lists of elements of `M.simplify.groundset()` that correspond to lines in the geometric representation other than the sides of the triangle.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers as mph
sage: M=Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0],
....: [0, 1, 0, 1, 0, 1, 1,0],[0, 0, 1, 1, 1, 0, 1,0]])
sage: N=M.simplify()
sage: B1=list(N.basis())
sage: nB1=list(set(M.simplify().groundset())-set(B1))
sage: pts,trilines,nontripts,curvedlines=mph.it(M,
....: B1,nB1,M.loops())
sage: print(pts)
{1: (1.0, 0.0), 2: (1.5, 1.0), 3: (0.5, 1.0), 4: (0, 0), 5: (1, 2),
6: (2, 0)}
sage: print(trilines)
[[3, 4, 5], [2, 5, 6], [1, 4, 6]]
sage: print(nontripts)
[0]
sage: print(curvedlines)
```

```
[[0, 1, 5], [0, 2, 4], [0, 3, 6], [1, 2, 3], [1, 4, 6], [2, 5, 6],
 [3, 4, 5]]
```

Note: This method does NOT do any checks.

sage.matroids.matroids_plot_helpers.**line_hasorder**(*l*, *lodrs=None*)

Determine if an order is specified for a line

INPUT:

- `l` – A line specified as a list of ground set elements.
- `lordrs` – (optional) 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 `lordrs` that is setwise equal to `l`.
2. A list specifying an order on `set(l)` if 1. is True, otherwise an empty list.

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: matroids_plot_helpers.line_hasorder(['a','b','c','d'],
....: [['a','c','d','b'],['p','q','r']])
(True, ['a', 'c', 'd', 'b'])
sage: matroids_plot_helpers.line_hasorder(['a','b','c','d'],
....: [['p','q','r'],['l','m','n','o']])
(False, [])
```

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:

```
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']]
```

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:

```
sage: from sage.matroids import matroids_plot_helpers
sage: M1=Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1,0,1],
....: [0, 1, 0, 1, 0, 1, 1,0,0,1,0],[0, 0, 1, 1, 1, 0, 1,0,0,0,0]])
sage: pos_dict= {0: (0, 0),  1: (2, 0),  2: (1, 2),  3: (1.5, 1.0),
....: 4: (0.5, 1.0),  5: (1.0, 0.0), 6: (1.0, 0.6666666666666666)}
sage: matroids_plot_helpers.posdict_is_sane(M1,pos_dict)
True
sage: pos_dict= {1: (2, 0),  2: (1, 2),  3: (1.5, 1.0),
....: 4: (0.5, 1.0), 5: (1.0, 0.0), 6: (1.0, 0.6666666666666666)}
sage: matroids_plot_helpers.posdict_is_sane(M1,pos_dict)
False
```

Note: This method does NOT do any checks. M1 is assumed to be a matroid and posdict is assumed to be a dictionary.

---

sage.matroids.matroids_plot_helpers.**slp**(*M1*, *pos_dict=None*, *B=None*)

Return simple matroid, loops and parallel elements of given matroid.

INPUT:

- M1 – A matroid.
- pos_dict – (optional) A dictionary containing non loopy elements of M as keys and their (x,y) positions. as keys. While simplifying the matroid, all except one element in a parallel class that is also specified in pos_dict will be retained.
- B – (optional) A basis of M1 that has been chosen for placement on vertices of triangle.

OUTPUT:

A tuple containing 3 elements in this order:

1. Simple matroid corresponding to M1.
2. Loops of matroid M1.
3. Elements that are in $M1.groundset()$ but not in ground set of 1 or in 2

EXAMPLES:

```
sage: from sage.matroids import matroids_plot_helpers
sage: from sage.matroids.advanced import setprint
sage: M1=Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1,0,1],
....: [0, 1, 0, 1, 0, 1, 1,0,0,1,0],[0, 0, 1, 1, 1, 0, 1,0,0,0,0]])
sage: [M,L,P]=matroids_plot_helpers.slp(M1)
sage: M.is_simple()
True
sage: setprint([L,P])
[{7}, {8, 9, 10}]
```

```
sage: M1=Matroid(ring=GF(2), matrix=[[1, 0, 0, 0, 1, 1, 1,0,1,0,1],
....: [0, 1, 0, 1, 0, 1, 1,0,0,1,0],[0, 0, 1, 1, 1, 0, 1,0,0,0,0]])
sage: posdict= {8: (0, 0),  1: (2, 0),  2: (1, 2),  3: (1.5, 1.0),
....: 4: (0.5, 1.0),  5: (1.0, 0.0), 6: (1.0, 0.6666666666666666)}
sage: [M,L,P]=matroids_plot_helpers.slp(M1,pos_dict=posdict)
sage: M.is_simple()
True
sage: setprint([L,P])
[{7}, {0, 9, 10}]
```

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]
```

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 co-ordinates 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:

```
sage: from sage.matroids import matroids_plot_helpers
sage: points=matroids_plot_helpers.trigrid([[2,1],[4,5],[5,2]])
sage: print(points)
[[3.6666666666666665, 2.6666666666666665],
 [3.222222222222222, 2.888888888888889],
```

```
 [4.222222222222222, 3.222222222222222],
 [3.5555555555555554, 1.8888888888888886]]
```

Note: This method does NOT do any checks.

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

6.3.1 Methods

class sage.matroids.set_system.**SetSystem**
Bases: object

A SetSystem is an enumerator of a collection of subsets of a given fixed and finite ground set. It offers the possibility to enumerate its contents. One is most likely to encounter these as output from some Matroid methods:

```
sage: M = matroids.named_matroids.Fano()
sage: M.circuits()
Iterator over a system of subsets
```

To access the sets in this structure, simply iterate over them. The simplest way must be:

```
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
sage: T = list(S)
```

Or immediately use it to iterate:

```
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
sage: [min(X) for X in S]
[1, 3, 1]
```

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 ground set so the each subset is either contained in X or disjoint from X.

EXAMPLES:

```
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
sage: S.is_connected()
True
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4]])
sage: S.is_connected()
False
sage: S = SetSystem([1], [])
sage: S.is_connected()
True
```

class sage.matroids.set_system.**SetSystemIterator**

Bases: object

Create an iterator for a SetSystem.

Called internally when iterating over the contents of a SetSystem.

EXAMPLES:

```
sage: from sage.matroids.set_system import SetSystem
sage: S = SetSystem([1, 2, 3, 4], [[1, 2], [3, 4], [1, 2, 4]])
sage: type(S.__iter__())
<... 'sage.matroids.set_system.SetSystemIterator'>
```

next()

x.next() -> the next value, or raise StopIteration

6.4 Unpickling methods

Python saves objects by providing a pair (f, data) such that f(data) reconstructs the object. This module collects the loading (_unpickling_ in Python terminology) functions for Sage's matroids.

Note: The reason this code was separated out from the classes was to make it play nice with lazy importing of the Matroid() and matroids keywords.

AUTHORS:

- Rudi Pendavingh, Stefan van Zwam (2013-07-01): initial version

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, name, BB) in which E is the groundset of the matroid, R is the rank, name is a custom name, and BB is the bitpacked list of bases, as pickled by Sage's bitset_pickle.

OUTPUT:

A matroid.

Warning: Users should never call this function directly.

EXAMPLES:

```
sage: from sage.matroids.advanced import *
sage: M = BasisMatroid(matroids.named_matroids.Vamos())
sage: M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_binary_matrix**(*version*, *data*)

Reconstruct a BinaryMatrix object (internal Sage data structure).

Warning: Users should not call this method directly.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = BinaryMatrix(2, 5)
sage: A == loads(dumps(A))  # indirect doctest
True
sage: C = BinaryMatrix(2, 2, Matrix(GF(2), [[1, 1], [0, 1]]))
sage: C == loads(dumps(C))
True
```

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).
- 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 BinaryMatroid instance.

Warning: Users should never call this function directly.

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(2), [[1, 0, 0, 1], [0, 1, 0, 1],
....:                            [0, 0, 1, 1]]))
sage: M == loads(dumps(M))  # indirect doctest
True
sage: M.rename("U34")
```

```
sage: loads(dumps(M))
U34
```

sage.matroids.unpickling.**unpickle_circuit_closures_matroid**(*version*, *data*)
Unpickle a CircuitClosuresMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – an integer, expected to be 0
- `data` – a tuple `(E, CC, name)` in which `E` is the groundset of the matroid, `CC` is the dictionary of circuit closures, and `name` is a custom name.

OUTPUT:

A matroid.

Warning: Users should never call this function directly.

EXAMPLES:

```
sage: M = matroids.named_matroids.Vamos()
sage: M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_dual_matroid**(*version*, *data*)
Unpickle a DualMatroid.

Pickling is Python's term for the loading and saving of objects. Functions like these serve to reconstruct a saved object. This all happens transparently through the `load` and `save` commands, and you should never have to call this function directly.

INPUT:

- `version` – an integer, expected to be 0
- `data` – a tuple `(M, name)` in which `M` is the internal matroid, and `name` is a custom name.

OUTPUT:

A matroid.

Warning: Users should not call this function directly. Instead, use load/save.

EXAMPLES:

```
sage: M = matroids.named_matroids.Vamos().dual()
sage: M == loads(dumps(M))  # indirect doctest
True
```

sage.matroids.unpickling.**unpickle_generic_matrix**(*version*, *data*)
Reconstruct a `GenericMatrix` object (internal Sage data structure).

Warning: Users should not call this method directly.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = GenericMatrix(2, 5, ring=QQ)
sage: A == loads(dumps(A))  # indirect doctest
True
```

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())
sage: M == loads(dumps(M))
True
```

sage.matroids.unpickling.**unpickle_integer_matrix**(*version*, *data*)

Reconstruct an `IntegerMatrix` object (internal Sage data structure).

Warning: Users should not call this method directly.

EXAMPLES:

```
sage: from sage.matroids.lean_matrix import *
sage: A = IntegerMatrix(2, 5)
sage: A == loads(dumps(A))  # indirect doctest
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, 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 `LinearMatroid` instance.

Warning: Users should never call this function directly.

EXAMPLES:

```
sage: M = Matroid(Matrix(GF(7), [[1, 0, 0, 1, 1], [0, 1, 0, 1, 2],
....:                            [0, 1, 1, 1, 3]]))
sage: M == loads(dumps(M))  # indirect doctest
True
sage: M.rename("U35")
sage: loads(dumps(M))
U35
```

`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:

```
sage: M = matroids.named_matroids.Vamos().minor('abc', 'g')
sage: M == loads(dumps(M))  # indirect doctest
True
```

`sage.matroids.unpickling.`**`unpickle_quaternary_matrix`**(*version*, *data*)

Reconstruct a `QuaternaryMatrix` object (internal Sage data structure).

Warning: Users should not call this method directly.

EXAMPLES:

```
sage: 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
```

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, 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 = QuaternaryMatroid(Matrix(GF(3), [[1, 0, 0, 1], [0, 1, 0, 1],
....:           [0, 0, 1, 1]]))
sage: M == loads(dumps(M))  # indirect doctest
True
sage: M.rename("U34")
sage: loads(dumps(M))
U34
sage: M = QuaternaryMatroid(Matrix(GF(4, 'x'), [[1, 0, 1],
....:                                           [1, 0, 1]]))
sage: loads(dumps(M)).representation()
[1 0 1]
[1 0 1]
```

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.named_matroids.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
```

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
```

CHAPTER

SEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

m

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W