CONTENTS

1 Set Constructions 1
2 Sets of Numbers 75
3 Indices and Tables 99
Python Module Index 101
Index 103
1.1 Cartesian products

AUTHORS:

• Nicolas Thiery (2010-03): initial version

```python
class sage.sets.cartesian_product.CartesianProduct(sets, category, flatten=False):
    Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A class implementing a raw data structure for Cartesian products of sets (and elements thereof). See cartesian_product for how to construct full fledged Cartesian products.

EXAMPLES:
```
```
sage: S1 = Sets().example()
sage: S2 = InfiniteEnumeratedSets().example()
sage: C = cartesian_product([S2, S1, S2])
sage: C._cartesian_product_of_elements([S2.an_element(), S1.an_element(), S2.an_element()])
(42, 47, 42)

class Element
Bases: sage.structure.element_wrapper.ElementWrapperCheckWrappedClass

cartesian_factors()

Return the tuple of elements that compose this element.

EXAMPLES:

sage: A = cartesian_product([ZZ, RR])
sage: A((1, 1.23)).cartesian_factors()
(1, 1.23000000000000)
sage: type(_)
<... 'tuple'>

cartesian_projection(i)

Return the projection of self on the $i$-th factor of the Cartesian product, as per Sets.CartesianProducts.ElementMethods.cartesian_projection().

INPUT:

• $i$ – the index of a factor of the Cartesian product

EXAMPLES:

sage: C = Sets().CartesianProducts().example(); C
The Cartesian product of (Set of prime numbers (basic implementation), An example of an infinite enumerated set: the non negative integers, An example of a finite enumerated set: {1,2,3})
sage: x = C.an_element(); x
(47, 42, 1)
sage: x.cartesian_projection(1)
42

wrapped_class

alias of builtins.tuple

an_element()

EXAMPLES:

sage: C = Sets().CartesianProducts().example(); C
The Cartesian product of (Set of prime numbers (basic implementation), An example of an infinite enumerated set: the non negative integers, An example of a finite enumerated set: {1,2,3})
sage: C.an_element()
(47, 42, 1)

cartesian_factors()

Return the Cartesian factors of self.

See also:
Sets.CartesianProducts.ParentMethods.cartesian_factors().

EXAMPLES:
**cartesian_projection** *(i)*  
Return the natural projection onto the *i*-th Cartesian factor of *self* as per *Sets.CartesianProducts.ParentMethods.cartesian_projection()*.

**INPUT:**  
- *i* – the index of a Cartesian factor of *self*

**EXAMPLES:**

```python  
sage: C = Sets().CartesianProducts().example(); C  
The Cartesian product of (Set of prime numbers (basic implementation), An→example of an infinite enumerated set: the non negative integers, An→example of a finite enumerated set: {1,2,3})  
sage: x = C.an_element(); x  
(47, 42, 1)  
sage: pi = C.cartesian_projection(1)  
sage: pi(x)  
42  
sage: C.cartesian_projection('hey')  
Traceback (most recent call last):  
...  
ValueError: i (=hey) must be in {0, 1, 2}
```

**construction()**  
Return the construction functor and its arguments for this Cartesian product.

**OUTPUT:**  
A pair whose first entry is a Cartesian product functor and its second entry is a list of the Cartesian factors.

**EXAMPLES:**

```python  
sage: cartesian_product([ZZ, QQ]).construction()  
(The cartesian_product functorial construction, (Integer Ring, Rational Field))
```

### 1.2 Families

A Family is an associative container which models a family \((f_i)_{i \in I}\). Then, \(f[i]\) returns the element of the family indexed by *i*. Whenever available, set and combinatorial class operations (counting, iteration, listing) on the family are induced from those of the index set. Families should be created through the *Family()* function.

**AUTHORS:**  
- Nicolas Thiery (2008-02): initial release  

```python  
class sage.sets.family.AbstractFamily  
    Bases: sage.structure.parent.Parent
    The abstract class for family  
    Any family belongs to a class which inherits from AbstractFamily.
```
hidden_keys()
Returns the hidden keys of the family, if any.

EXAMPLES:

```python
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: f.hidden_keys()
[]
```

inverse_family()
Returns the inverse family, with keys and values exchanged. This presumes that there are no duplicate values in self.

This default implementation is not lazy and therefore will only work with not too big finite families. It is also cached for the same reason:

```python
sage: Family({3: 'a', 4: 'b', 7: 'd'}).inverse_family()
Finite family {'a': 3, 'b': 4, 'd': 7}
sage: Family((3,4,7)).inverse_family()
Finite family {3: 0, 4: 1, 7: 2}
```

map(f, name=None)
Returns the family \( \{ f(self[i]) \}_{i \in I} \), where \( I \) is the index set of self.

Todo: good name?

EXAMPLES:

```python
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: g = f.map(lambda x: x+'1')
sage: list(g)
['a1', 'b1', 'd1']
```

zip(f, other, name=None)
Given two families with same index set \( I \) (and same hidden keys if relevant), returns the family \( \{ f(self[i], other[i]) \}_{i \in I} \).

Todo: generalize to any number of families and merge with map?

EXAMPLES:

```python
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: g = Family({3: '1', 4: '2', 7: '3'})
sage: h = f.zip(lambda x, y: x+y, g)
sage: list(h)
['a1', 'b2', 'd3']
```

class sage.sets.family.EnumeratedFamily(enumset)
Bases: sage.sets.family.LazyFamily

EnumeratedFamily turns an enumerated set \( c \) into a family indexed by the set \( \{0, \ldots, |c| - 1\} \).

Instances should be created via the Family() factory. See its documentation for examples and tests.

cardinality()
Return the number of elements in self.
EXAMPLES:

```python
sage: from sage.sets.family import EnumeratedFamily
sage: f = EnumeratedFamily(Permutations(3))
sage: f.cardinality()
6
sage: f = Family(NonNegativeIntegers())
sage: f.cardinality()
+Infinity
```

```
sage.sets.family.Family(indices=None, function=None, hidden_keys=[], hidden_function=None, lazy=False, name=None)
```

A Family is an associative container which models a family \( (f_i)_{i \in I} \). Then, \( f[i] \) returns the element of the family indexed by \( i \). Whenever available, set and combinatorial class operations (counting, iteration, listing) on the family are induced from those of the index set.

There are several available implementations (classes) for different usages; Family serves as a factory, and will create instances of the appropriate classes depending on its arguments.

**INPUT:**

- `indices` – the indices for the family
- `function` – (optional) the function \( f \) applied to all visible indices; the default is the identity function
- `hidden_keys` – (optional) a list of hidden indices that can be accessed through \( my \_\_\_family[i] \)
- `hidden_function` – (optional) a function for the hidden indices
- `lazy` – boolean (default: False); whether the family is lazily created or not; if the indices are infinite, then this is automatically made True
- `name` – (optional) the name of the function; only used when the family is lazily created via a function

**EXAMPLES:**

In its simplest form, a list \( l = [l_0, l_1, \ldots, l_\ell] \) or a tuple by itself is considered as the family \( (l_i)_{i \in I} \) where \( I \) is the set \( \{0, \ldots, \ell\} \) where \( \ell = \text{len}(l) - 1 \). So `Family(l)` returns the corresponding family:

```
sage: f = Family([1,2,3])
sage: f
Family (1, 2, 3)
sage: f = Family((1,2,3))
sage: f
Family (1, 2, 3)
```

Instead of a list you can as well pass any iterable object:

```
sage: f = Family(2*i+1 for i in [1,2,3])
sage: f
Family (3, 5, 7)
```

A family can also be constructed from a dictionary \( t \). The resulting family is very close to \( t \), except that the elements of the family are the values of \( t \). Here, we define the family \( (f_i)_{i \in \{3,4,7\}} \) with \( f_3 = a \), \( f_4 = b \), and \( f_7 = d \):

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: f
Finite family {3: 'a', 4: 'b', 7: 'd'}
sage: f[7]
'd'
```

(continues on next page)
A family can also be constructed by its index set $I$ and a function $f$, as in $(f(i))_{i \in I}$:

```
sage: f = Family([3,4,7], lambda i: 2*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f[7]
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

By default, all images are computed right away, and stored in an internal dictionary:

```
sage: f = Family((3,4,7), lambda i: 2*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
```

Note that this requires all the elements of the list to be hashable. One can ask instead for the images $f(i)$ to be computed lazily, when needed:

```
sage: f = Family([3,4,7], lambda i: 2*i, lazy=True)
sage: f
Lazy family (<lambda>(i))_{i in [3, 4, 7]}
sage: f[7]
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
```

This allows in particular for modeling infinite families:

```
sage: f = Family(ZZ, lambda i: 2*i, lazy=True)
sage: f
Lazy family (<lambda>(i))_{i in Integer Ring}
sage: f.keys()
Integer Ring
```

(continues on next page)
sage: f[1]
2
sage: f[-5]
-10
sage: i = iter(f)
sage: next(i), next(i), next(i), next(i), next(i)
(0, 2, -2, 4, -4)

Note that the lazy keyword parameter is only needed to force laziness. Usually it is automatically set to a correct default value (ie: False for finite data structures and True for enumerated sets:

sage: f == Family(ZZ, lambda i: 2*i)
True

Beware that for those kind of families len(f) is not supposed to work. As a replacement, use the .cardinality() method:

sage: f = Family(Permutations(3), attrcall("to_lehmer_code")); f
Lazy family (<lambda>(i))_{i in Standard permutations of 3}
sage: f == loads(dumps(f))
True
sage: f = Family(Permutations(3), attrcall("to_lehmer_code")); f
Lazy family {i.to_lehmer_code() for i in Standard permutations of 3}
sage: f == loads(dumps(f))
True

Caveat: Only certain families with lazy behavior can be pickled. In particular, only functions that work with Sage's pickle_function and unpickle_function (in sage.misc.fpickle) will correctly unpickle. The following two work:

sage: def plus_n(n):
    return lambda x: x+n
sage: f = Family([1,2,3], plus_n(3), lazy=True); f
Lazy family (<lambda>(i))_{i in [1, 2, 3]}
sage: f == loads(dumps(f))
Traceback (most recent call last):
  ... ValueError: Cannot pickle code objects from closures

Finally, it can occasionally be useful to add some hidden elements in a family, which are accessible as f[i], but do not appear in the keys or the container operations:

sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2])
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]

(continues on next page)

1.2. Families
The following example illustrates when the function is actually called:

```python
sage: def compute_value(i):
    ...:     print('computing 2*'+str(i))
    ...:     return 2*i
sage: f = Family([3,4,7], compute_value, hidden_keys=[2])
computing 2*3
computing 2*4
computing 2*7
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]
sage: f[7]
14
sage: f[2]
computing 2*2
4
sage: f[2]
4
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

Here is a close variant where the function for the hidden keys is different from that for the other keys:

```python
sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2], hidden_function = lambda i: 3*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]
sage: f[7]
14
sage: f[2]
6
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
```

sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3

Family accept finite and infinite EnumeratedSets as input:

sage: f = Family(FiniteEnumeratedSet([1,2,3]))
sage: f
Family (1, 2, 3)
sage: f = Family(NonNegativeIntegers())
sage: f
Family (Non negative integers)
sage: f = Family(FiniteEnumeratedSet([3,4,7]), lambda i: 2*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
{3, 4, 7}
sage: f[7]
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3

class sage.sets.family.FiniteFamily(dictionary, keys=None)
Bases: sage.sets.family.AbstractFamily

A FiniteFamily is an associative container which models a finite family $(f_i)_{i \in I}$. Its elements $f_i$ are therefore its values. Instances should be created via the Family() factory. See its documentation for examples and tests.

EXAMPLES:

We define the family $(f_i)_{i \in \{3,4,7\}}$ with $f_3 = a$, $f_4 = b$, and $f_7 = d$:

sage: from sage.sets.family import FiniteFamily
sage: f = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})

Individual elements are accessible as in a usual dictionary:

sage: f[7]
'd'

And the other usual dictionary operations are also available:

sage: len(f)
3
sage: f.keys()
[3, 4, 7]

However f behaves as a container for the $f_i$’s:
The order of the elements can be specified using the `keys` optional argument:

```python
sage: f = FiniteFamily({"a": "aa", "b": "bb", "c": "cc" }, keys = ["c", "a", "b" →])
sage: list(f)
['cc', 'aa', 'bb']
```

**cardinality()**

Returns the number of elements in self.

**EXAMPLES:**

```python
sage: from sage.sets.family import FiniteFamily
dsage: f = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})
sage: f.cardinality()
3
```

**has_key(k)**

Returns whether k is a key of self

**EXAMPLES:**

```python
dsage: Family({"a":1, "b":2, "c":3}).has_key("a")
True
dsage: Family({"a":1, "b":2, "c":3}).has_key("d")
False
```

**keys()**

Returns the index set of this family

**EXAMPLES:**

```python
dsage: f = Family(["c", "a", "b"], lambda x: x+x)
dsage: f.keys()
['c', 'a', 'b']
```

**values()**

Returns the elements of this family

**EXAMPLES:**

```python
dsage: f = Family(["c", "a", "b"], lambda x: x+x)
dsage: f.values()
['cc', 'aa', 'bb']
```

**class** `sage.sets.family.FiniteFamilyWithHiddenKeys` *(dictionary, hidden_keys, hidden_function, keys=None)*

**Bases:** `sage.sets.family.FiniteFamily`

A close variant of `FiniteFamily` where the family contains some hidden keys whose corresponding values are computed lazily (and remembered). Instances should be created via the `Family()` factory. See its documentation for examples and tests.

**Caveat:** Only instances of this class whose functions are compatible with `sage.misc.fpickle` can be pickled.
hidden_keys()  
Returns self’s hidden keys.

EXAMPLES:

```python
sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2])
sage: f.hidden_keys()
[2]
```

class sage.sets.family.LazyFamily(set, function, name=None)

Bases: sage.sets.family.AbstractFamily

A LazyFamily(I, f) is an associative container which models the (possibly infinite) family $(f(i))_{i \in I}$.

Instances should be created via the `Family()` factory. See its documentation for examples and tests.

cardinality()  
Return the number of elements in self.

EXAMPLES:

```python
sage: from sage.sets.family import LazyFamily
sage: f = LazyFamily([3,4,7], lambda i: 2*i)
sage: f.cardinality()
3
sage: l = LazyFamily(NonNegativeIntegers(), lambda i: 2*i)
sage: l.cardinality()
+Infinity
```

keys()  
Returns self’s keys.

EXAMPLES:

```python
sage: from sage.sets.family import LazyFamily
sage: f = LazyFamily([3,4,7], lambda i: 2*i)
sage: f.keys()
[3, 4, 7]
```

class sage.sets.family.TrivialFamily(enumeration)

Bases: sage.sets.family.AbstractFamily

TrivialFamily turns a list/tuple $c$ into a family indexed by the set \{0,...,|c|−1\}.

Instances should be created via the `Family()` factory. See its documentation for examples and tests.

cardinality()  
Return the number of elements in self.

EXAMPLES:

```python
sage: from sage.sets.family import TrivialFamily
sage: f = TrivialFamily([3,4,7])
sage: f.cardinality()
3
```

keys()  
Returns self’s keys.

EXAMPLES:
1.3 Sets

AUTHORS:

- William Stein (2005) - first version
- William Stein (2006-02-16) - large number of documentation and examples; improved code
- Mike Hansen (2007-3-25) - added differences and symmetric differences; fixed operators
- Florent Hivert (2010-06-17) - Adapted to categories
- Nicolas M. Thiery (2011-03-15) - Added subset and superset methods
- Julian Rueth (2013-04-09) - Collected common code in `Set_object_binary`, fixed `__hash__`

`sage.sets.set.Set(X=[])`
Create the underlying set of X.

If X is a list, tuple, Python set, or X.is_finite() is True, this returns a wrapper around Python’s enumerated immutable `frozenset` type with extra functionality. Otherwise it returns a more formal wrapper.

If you need the functionality of mutable sets, use Python’s builtin set type.

EXAMPLES:

```python
code=
```

```python
code=
```

```python
code=
```

Usually sets can be used as dictionary keys.

```python
code=
```

```python
code=
```

The original object is often forgotten.

```python
code=
```

(continues on next page)

Set also accepts iterators, but be careful to only give finite sets:

```sage
sage: sorted(Set(range(1,6)))
[1, 2, 3, 4, 5]
sage: sorted(Set(list(range(1,6))))
[1, 2, 3, 4, 5]
sage: sorted(Set(iter(range(1,6))))
[1, 2, 3, 4, 5]
```

We can also create sets from different types:

```sage
sage: sorted(Set([Sequence([3,1], immutable=True), 5, QQ, Partition([3,1,1])]), key=str)
[5, Rational Field, [3, 1, 1], [3, 1]]
```

Sets with unhashable objects work, but with less functionality:

```sage
sage: A = Set([QQ, (3, 1), 5])  # hashable
sage: sorted(A.list(), key=repr)
[(3, 1), 5, Rational Field]
sage: type(A)
<class 'sage.sets.set.Set_object_enumerated_with_category'>
sage: B = Set([QQ, [3, 1], 5])  # unhashable
sage: sorted(B.list(), key=repr)
Traceback (most recent call last):
  ... AttributeError: 'Set_object_with_category' object has no attribute 'list'
sage: type(B)
<class 'sage.sets.set.Set_object_with_category'>
```

class `sage.sets.set.Set_object` (`X, category=None`)

Bases: `sage.structure.parent.Set_generic`

A set attached to an almost arbitrary object.

EXAMPLES:

```sage
sage: K = GF(19)
sage: Set(K)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
sage: S = Set(K)
sage: latex(S)
\(\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}\)
sage: TestSuite(S).run()
sage: latex(Set(ZZ))
\Bold{Z}

an_element()

Return the first element of `self` returned by `__iter__()`

1.3. Sets 13
If `self` is empty, the exception `EmptySetError` is raised instead.

This provides a generic implementation of the method `_an_element_()` for all parents in `EnumeratedSets`.

**EXAMPLES:**

```python
sage: C = FiniteEnumeratedSets().example(); C
An example of a finite enumerated set: {1,2,3}
sage: C.an_element()  # indirect doctest
1
sage: S = Set([])
sage: S.an_element()
Traceback (most recent call last):
  ... EmptySetError
```

**cardinality()**

Return the cardinality of this set, which is either an integer or `Infinity`.

**EXAMPLES:**

```python
sage: Set(ZZ).cardinality()
+Infinity
sage: Primes().cardinality()
+Infinity
sage: Set(GF(5)).cardinality()
5
sage: Set(GF(5^2,'a')).cardinality()
25
```

**difference(X)**

Return the set difference `self - X`.

**EXAMPLES:**

```python
sage: X = Set(ZZ).difference(Primes())
sage: 4 in X
True
sage: 3 in X
False
sage: 4/1 in X
True
sage: X = Set(GF(9,'b')).difference(Set(GF(27,'c')))
sage: X
{0, 1, 2, b, b + 1, b + 2, 2*b, 2*b + 1, 2*b + 2}
sage: X = Set(GF(9,'b')).difference(Set(GF(27,'b')))
sage: X
{0, 1, 2, b, b + 1, b + 2, 2*b, 2*b + 1, 2*b + 2}
```

**intersection(X)**

Return the intersection of `self` and `X`.

**EXAMPLES:**
sage: X = Set(ZZ).intersection(Primes())
sage: 4 in X
False
sage: 3 in X
True
sage: 2/1 in X
True
sage: X = Set(GF(9,'b')).intersection(Set(GF(27,'c')))
sage: X
{}
sage: X = Set(GF(9,'b')).intersection(Set(GF(27,'b')))
sage: X
{}

**is_empty()**

Return boolean representing emptiness of the set.

**OUTPUT:**

True if the set is empty. False if otherwise.

**EXAMPLES:**

```sage
sage: Set([]).is_empty()
True
sage: Set([0]).is_empty()
False
sage: Set([1..100]).is_empty()
False
sage: Set(SymmetricGroup(2).list()).is_empty()
False
sage: Set(ZZ).is_empty()
False
```

**is_finite()**

Return True if self is finite.

**EXAMPLES:**

```sage
sage: Set(QQ).is_finite()
False
sage: Set(GF(250037)).is_finite()
True
sage: Set(Integers(2^1000000)).is_finite()
True
sage: Set([1,'a',ZZ]).is_finite()
True
```

**object()**

Return underlying object.

**EXAMPLES:**

```sage
sage: X = Set(QQ)
sage: X.object()
Rational Field
```

(continues on next page)
\begin{verbatim}
sage: X = Primes()
sage: X.object()
Set of all prime numbers: 2, 3, 5, 7, ...
\end{verbatim}

\textbf{subsets (size=None)}

Return the \texttt{Subsets} object representing the subsets of a set. If size is specified, return the subsets of that size.

**EXAMPLES:**

\begin{verbatim}
sage: X = Set([1,2,3])
sage: list(X.subsets())
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
sage: list(X.subsets(2))
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}]
\end{verbatim}

\textbf{subsets_lattice()}

Return the lattice of subsets ordered by containment.

**EXAMPLES:**

\begin{verbatim}
sage: X = Set([1,2,3])
sage: X.subsets_lattice()
Finite lattice containing 8 elements
sage: Y = Set()
sage: Y.subsets_lattice()
Finite lattice containing 1 elements
\end{verbatim}

\textbf{symmetric_difference (X)}

Returns the symmetric difference of \texttt{self} and \texttt{X}.

**EXAMPLES:**

\begin{verbatim}
sage: X = Set([1,2,3]).symmetric_difference(Set([3,4]))
sage: X
{1, 2, 4}
\end{verbatim}

\textbf{union (X)}

Return the union of \texttt{self} and \texttt{X}.

**EXAMPLES:**

\begin{verbatim}
sage: Set(QQ).union(Set(ZZ))
Set-theoretic union of Set of elements of Rational Field and Set of elements of Integer Ring
sage: Set(QQ) + Set(ZZ)
Set-theoretic union of Set of elements of Rational Field and Set of elements of Integer Ring
sage: X = Set(QQ).union(Set(GF(3))); X
Set-theoretic union of Set of elements of Rational Field and \{0, 1, 2\}
sage: 2/3 in X
True
sage: GF(3)(2) in X
True
sage: GF(5)(2) in X
False
sage: sorted(Set(GF(7)) + Set(GF(3)), key=int)
[0, 0, 1, 1, 2, 2, 3, 4, 5, 6]
\end{verbatim}
class sage.sets.set.Set_object_binary (X, Y, op, latex_op)
    Bases: sage.sets.set.Set_object

    An abstract common base class for sets defined by a binary operation (ex.
    Set_object_union, Set_object_intersection, Set_object_difference, and
    Set_object_symmetric_difference).

    INPUT:

    - X, Y – sets, the operands to op
    - op – a string describing the binary operation
    - latex_op – a string used for rendering this object in LaTeX

    EXAMPLES:

    sage: X = Set(QQ^2)
sage: Y = Set(ZZ)
sage: from sage.sets.set import Set_object_binary
sage: S = Set_object_binary(X, Y, "union", "\cup"); S
Set-theoretic union of Set of elements of Vector space of dimension 2
over Rational Field and Set of elements of Integer Ring

class sage.sets.set.Set_object_difference (X, Y)
    Bases: sage.sets.set.Set_object_binary

    Formal difference of two sets.

    is_finite ()
    Return whether this set is finite.

    EXAMPLES:

    sage: X = Set(range(10))
sage: Y = Set(range(-10,5))
sage: Z = Set(QQ)
sage: X.difference(Y).is_finite()
True
sage: X.difference(Z).is_finite()
True
sage: Z.difference(X).is_finite()
False
sage: Z.difference(Set(ZZ)).is_finite()
Traceback (most recent call last):
... NotImplementedError

class sage.sets.set.Set_object Enumerated (X)
    Bases: sage.sets.set.Set_object

    A finite enumerated set.

    cardinality ()
    Return the cardinality of self.

    EXAMPLES:

    sage: Set([1,1]).cardinality()
1
**difference (other)**

Return the set difference \( \text{self} - \text{other} \).

**EXAMPLES:**

```
sage: X = Set([1,2,3,4])
sage: Y = Set([1,2])
sage: X.difference(Y)
{3, 4}
sage: Z = Set(ZZ)
sage: W = Set([2.5, 4, 5, 6])
sage: W.difference(Z)
{2.50000000000000}
```

**frozenset ()**

Return the Python frozenset object associated to this set, which is an immutable set (hence hashable).

**EXAMPLES:**

```
sage: X = Set(GF(8,'c'))
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: s = X.set(); s
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: hash(s)
Traceback (most recent call last):
... TypeError: unhashable type: 'set'
sage: s = X.frozenset(); s
frozenset({0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1})
sage: hash(s) != hash(tuple(X.set()))
True
sage: type(s)
<... 'frozenset'>
```

**intersection (other)**

Return the intersection of \( \text{self} \) and \( \text{other} \).

**EXAMPLES:**

```
sage: X = Set(GF(8,'c'))
sage: Y = Set([GF(8,'c').0, 1, 2, 3])
sage: X.intersection(Y)
{1, c}
```

**is_finite ()**

Return True as this is a finite set.

**EXAMPLES:**

```
sage: Set(GF(19)).is_finite()
True
```

**issubset (other)**

Return whether \( \text{self} \) is a subset of \( \text{other} \).

**INPUT:**

- \( \text{self} \)
- \( \text{other} \)
• other – a finite Set

EXAMPLES:

```python
sage: X = Set([1,3,5])
sage: Y = Set([0,1,2,3,5,7])
sage: X.issubset(Y)  # True
sage: Y.issubset(X)  # False
sage: X.issubset(X)  # True
```

`issuperset (other)`

Return whether `self` is a superset of `other`.

INPUT:

• other – a finite Set

EXAMPLES:

```python
sage: X = Set([1,3,5])
sage: Y = Set([0,1,2,3,5,7])
sage: X.issuperset(Y)  # False
sage: Y.issuperset(X)  # True
sage: X.issuperset(X)  # True
```

`list()`

Return the elements of `self`, as a list.

EXAMPLES:

```python
sage: X = Set(GF(8,'c'))
sage: X  # {0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: X.list()  # [0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1]
sage: type(X.list())  # <... 'list'>
```

Todo: FIXME: What should be the order of the result? That of `self.object()`? Or the order given by `set(self.object())`? Note that `__getitem__()` is currently implemented in term of this list method, which is really inefficient ...

`random_element()`

Return a random element in this set.

EXAMPLES:

```python
sage: Set([1,2,3]).random_element()  # random
2
```

`set()`

Return the Python set object associated to this set.

1.3. Sets
Python has a notion of finite set, and often Sage sets have an associated Python set. This function returns that set.

EXAMPLES:

```python
sage: X = Set(GF(8,'c'))
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: X.set()
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: type(X.set())
<... 'set'>
sage: type(X)
<class 'sage.sets.set.Set_object_enumerated_with_category'>
```

**symmetric_difference**(other)

Return the symmetric difference of self and other.

EXAMPLES:

```python
sage: X = Set([1,2,3,4])
sage: Y = Set([1,2])
sage: X.symmetric_difference(Y)
{3, 4}
sage: Z = Set(ZZ)
sage: W = Set([2.5, 4, 5, 6])
sage: U = W.symmetric_difference(Z)
sage: 2.5 in U
True
sage: 4 in U
False
sage: V = 2.5.symmetric_difference(W)
sage: V == U
True
sage: 2.5 in V
True
sage: 6 in V
False
```

**union**(other)

Return the union of self and other.

EXAMPLES:

```python
sage: X = Set(GF(8,'c'))
sage: Y = Set([GF(8,'c').0, 1, 2, 3])
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: sorted(Y)
[1, 2, 3, c]
sage: sorted(X.union(Y), key=str)
[0, 1, 2, 3, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1]
```

**class** **sage.sets.set.Set_object_intersection**(X, Y)

**Bases:** **sage.sets.set.Set_object_binary**

Formal intersection of two sets.

**is_finite**()

Return whether this set is finite.
EXERCISE:
```python
sage: X = Set(IntegerRange(100))
sage: Y = Set(ZZ)
sage: X.intersection(Y).is_finite()
True
sage: Y.intersection(X).is_finite()
True
sage: Y.intersection(Set(QQ)).is_finite()
Traceback (most recent call last):
  ... 
NotImplementedError
```

```python
class sage.sets.set.Set_object_symmetric_difference(X, Y)
    Bases: sage.sets.set.Set_object_binary

    Formal symmetric difference of two sets.

    is_finite()
    Return whether this set is finite.

    EXAMPLES:
```python
sage: X = Set(range(10))
sage: Y = Set(range(-10, 5))
sage: Z = Set(QQ)
sage: X.symmetric_difference(Y).is_finite()
True
sage: X.symmetric_difference(Z).is_finite()
False
sage: Z.symmetric_difference(X).is_finite()
False
sage: Z.symmetric_difference(Set(ZZ)).is_finite()
Traceback (most recent call last):
  ... 
NotImplementedError
```

```python
class sage.sets.set.Set_object_union(X, Y)
    Bases: sage.sets.set.Set_object_binary

    A formal union of two sets.

    cardinality()
    Return the cardinality of this set.

    EXAMPLES:
```python
sage: X = Set(GF(3)).union(Set(GF(2)))
sage: X
{0, 1, 2, 0, 1}
sage: X.cardinality()
5
sage: X = Set(GF(3)).union(Set(ZZ))
sage: X.cardinality()
+Infinity
```

is_finite()
Return whether this set is finite.

EXAMPLES:
sage: X = Set(range(10))
sage: Y = Set(range(-10,0))
sage: Z = Set(Primes())
sage: X.union(Y).is_finite()
True
sage: X.union(Z).is_finite()
False

sage.sets.set.has_finite_length(obj)
Return True if obj is known to have finite length.
This is mainly meant for pure Python types, so we do not call any Sage-specific methods.

EXAMPLES:

sage: from sage.sets.set import has_finite_length
sage: has_finite_length(tuple(range(10)))
True
sage: has_finite_length(list(range(10)))
True
sage: has_finite_length(set(range(10)))
True
sage: has_finite_length(iter(range(10)))
False
sage: has_finite_length(GF(17^127))
True
sage: has_finite_length(ZZ)
False

1.4 Disjoint-set data structure

The main entry point is DisjointSet() which chooses the appropriate type to return. For more on the data structure, see DisjointSet().

This module defines a class for mutable partitioning of a set, which can not be used as a key of a dictionary, vertex of a graph etc. For immutable partitioning see SetPartition.

AUTHORS:
• Sébastien Labbé (2008) - Initial version.
• Sébastien Labbé (2009-11-24) - Pickling support
• Sébastien Labbé (2010-01) - Inclusion into sage (trac ticket #6775).

EXAMPLES:
Disjoint set of integers from 0 to n - 1:

sage: s = DisjointSet(6)
sage: s
{(0), {1}, {2}, {3}, {4}, {5}}
sage: s.union(2, 4)
sage: s.union(1, 3)
sage: s.union(5, 1)
sage: s
{(0), {1, 3, 5}, {2, 4}}
Disjoint set of hashables objects:

```python
dsage: d = DisjointSet('abcde')
d sage: d
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}}
sage: d.union('a','b')
sage: d.union('b','c')
sage: d.union('c','d')
sage: d
{{'a', 'b', 'c', 'd'}, {'e'}}
sage: d.find('c')
'a'
```

**sage.sets.disjoint_set.DisjointSet** *(arg)*

Constructs a disjoint set where each element of `arg` is in its own set. If `arg` is an integer, then the disjoint set returned is made of the integers from 0 to `arg - 1`.

A disjoint-set data structure (sometimes called union-find data structure) is a data structure that keeps track of a partitioning of a set into a number of separate, nonoverlapping sets. It performs two operations:

- **find()** – Determine which set a particular element is in.
- **union()** – Combine or merge two sets into a single set.

**REFERENCES:**

- Wikipedia article Disjoint-set_data_structure

**INPUT:**

- `arg` – non negative integer or an iterable of hashable objects.

**EXAMPLES:**

From a non-negative integer:

```python
sage: DisjointSet(5)
{{0}, {1}, {2}, {3}, {4}}
```

From an iterable:

```python
sage: DisjointSet('abcde')
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}}
sage: DisjointSet(range(6))
{{0}, {1}, {2}, {3}, {4}, {5}}
sage: DisjointSet(['yi', '45', 'cheval'])
{{'cheval'}, {'yi'}, {45}}
```

**class** sage.sets.disjoint_set.DisjointSet_class

Bases: sage.structure.sage_object.SageObject


**cardinality()**

Return the number of elements in `self`, not the number of subsets.
EXAMPLES:

```python
sage: d = DisjointSet(5)
sage: d.cardinality()
5
sage: d.union(2, 4)
sage: d.cardinality()
5
sage: d = DisjointSet(range(5))
sage: d.cardinality()
5
sage: d.union(2, 4)
sage: d.cardinality()
5
```

**number_of_subsets()**

Return the number of subsets in `self`.

EXAMPLES:

```python
sage: d = DisjointSet(5)
sage: d.number_of_subsets()
5
sage: d.union(2, 4)
sage: d.number_of_subsets()
4
sage: d = DisjointSet(range(5))
sage: d.number_of_subsets()
5
sage: d.union(2, 4)
sage: d.number_of_subsets()
4
```

**class sage.sets.disjoint_set.DisjointSet_of_hashables**

Bases: `sage.sets.disjoint_set.DisjointSet_class`

Disjoint set of hashables.

EXAMPLES:

```python
sage: d = DisjointSet('abcde')
sage: d
{{'a'}, {'b'}, {'c'}, {'d'}, {'e'}}
sage: d.union('a', 'c')
sage: d
{{'a', 'c'}, {'b'}, {'d'}, {'e'}}
sage: d.find('a')
'a'
```

**element_to_root_dict()**

Return the dictionary where the keys are the elements of `self` and the values are their representative inside a list.

EXAMPLES:

```python
sage: d = DisjointSet(range(5))
sage: d.union(2, 3)
sage: d.union(4, 1)
sage: e = d.element_to_root_dict()
```
find(e)
Return the representative of the set that e currently belongs to.

INPUT:
• e – element in self

EXAMPLES:

```python
sage: e = DisjointSet(range(5))
sage: e.union(4,2)
sage: e
{{0}, {1}, {2, 4}, {3}}
sage: e.find(2)
4
sage: e.find(4)
4
sage: e.union(1,3)
sage: e
{{0}, {1, 3}, {2, 4}}
sage: e.find(1)
1
sage: e.find(3)
1
sage: e.union(3,2)
sage: e
{{0}, {1, 2, 3, 4}}
sage: [e.find(i) for i in range(5)]
[0, 1, 1, 1, 1]
sage: e.find(5)
Traceback (most recent call last):
  ... KeyError: 5
```

root_to_elements_dict()
Return the dictionary where the keys are the roots of self and the values are the elements in the same set.

EXAMPLES:

```python
sage: d = DisjointSet(range(5))
sage: d.union(2,3)
sage: e = d.root_to_elements_dict()
sage: sorted(e.items())
[[(0, [0]), (2, [2, 3]), (4, [1, 4])]
```

to_digraph()
Return the current digraph of self where (a, b) is an oriented edge if b is the parent of a.

EXAMPLES:

```python
sage: d = DisjointSet(range(5))
sage: d.union(2,3)
```

(continues on next page)
The result depends on the ordering of the union:

```python
sage: d = DisjointSet(range(5))
sage: d.union(1,2)
sage: d.union(1,3)
sage: d.union(1,4)
sage: d.union(0,2)
sage: d.union(0,3)
sage: d.union(0,4)
sage: d
{{0}, {1}, {2}, {3}, {4}}
```
{{0, 2, 4}, {1}, {3}}
sage: d.find(2)
2

\texttt{element\_to\_root\_dict()}
Return the dictionary where the keys are the elements of self and the values are their representative inside a list.

\textbf{EXAMPLES:}

```
sage: d = DisjointSet(5)
sage: d.union(2,3)
sage: d.union(4,1)
sage: e = d.element_to_root_dict(); e
{0: 0, 1: 4, 2: 2, 3: 2, 4: 4}
sage: WordMorphism(e)
WordMorphism: 0->0, 1->4, 2->2, 3->2, 4->4
```

\texttt{find(i)}
Return the representative of the set that \texttt{i} currently belongs to.

\textbf{INPUT:}

- \texttt{i} – element in self

\textbf{EXAMPLES:}

```
sage: e = DisjointSet(5)
sage: e.union(4,2)
sage: e
{{0}, {1}, {2, 4}, {3}}
sage: e.find(2)
4
sage: e.find(4)
4
sage: e.union(1,3)
sage: e
{{0}, {1, 3}, {2, 4}}
sage: e.find(1)
1
sage: e.find(3)
1
sage: e.union(3,2)
sage: e
{{0}, {1, 2, 3, 4}}
sage: [e.find(i) for i in range(5)]
[0, 1, 1, 1, 1]
sage: e.find(5)
Traceback (most recent call last):
...  
ValueError: i(=5) must be between 0 and 4
```

\texttt{root\_to\_elements\_dict()}
Return the dictionary where the keys are the roots of self and the values are the elements in the same set as the root.

\textbf{EXAMPLES:}
sage: d = DisjointSet(5)
sage: sorted(d.root_to_elements_dict().items())
[(0, [0]), (1, [1]), (2, [2]), (3, [3]), (4, [4])]
sage: d.union(2,3)
sage: sorted(d.root_to_elements_dict().items())
[(0, [0]), (1, [1]), (2, [2, 3]), (4, [4])]
sage: d.union(3,0)
sage: sorted(d.root_to_elements_dict().items())
[(1, [1]), (2, [0, 2, 3]), (4, [4])]
sage: d
{{0, 2, 3}, {1}, {4}}

to_digraph()
Return the current digraph of self where (a, b) is an oriented edge if b is the parent of a.

EXAMPLES:

sage: d = DisjointSet(5)
sage: d.union(2,3)
sage: d.union(4,1)
sage: d.union(3,4)
sage: d
{{0}, {1, 2, 3, 4}}
sage: g = d.to_digraph(); g
Looped digraph on 5 vertices
sage: g.edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None), (4, 4, None)]

The result depends on the ordering of the union:

sage: d = DisjointSet(5)
sage: d.union(1,2)
sage: d.union(1,3)
sage: d.union(1,4)
sage: d
{{0}, {1, 2, 3, 4}}
sage: d.to_digraph().edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None), (4, 4, None)]

union(i, j)
Combine the set of i and the set of j into one.

All elements in those two sets will share the same representative that can be gotten using find.

INPUT:

- i - element in self
- j - element in self

EXAMPLES:

sage: d = DisjointSet(5)
sage: d
{{0, 1, 2, 3, 4}}
sage: d.union(0,1)
sage: d
{{0, 1, 2, 3, 4}}
sage: d.union(2,4)
(continues on next page)
1.5 Disjoint union of enumerated sets

AUTHORS:

• Florent Hivert (2009-07/09): initial implementation.
• Florent Hivert (2010-03): classcall related stuff.
• Florent Hivert (2010-12): fixed facade element construction.

```python
d = {{0, 1}, {2, 4}, {3}}
d.union(1,4)
d.union(1,5)
```

```
Traceback (most recent call last):
... ValueError: j(=5) must be between 0 and 4
```

A class for disjoint unions of enumerated sets.

INPUT:

- `family` – a list (or iterable or family) of enumerated sets
- `keepkey` – a boolean
- `facade` – a boolean

This models the enumerated set obtained by concatenating together the specified ordered sets. The latter are supposed to be pairwise disjoint; otherwise, a multiset is created.

The argument `family` can be a list, a tuple, a dictionary, or a family. If it is not a family it is first converted into a family (see `sage.sets.family.Family()`).

Experimental options:

By default, there is no way to tell from which set of the union an element is generated. The option `keepkey=True` keeps track of those by returning pairs `(key, el)` where `key` is the index of the set to which `el` belongs. When this option is specified, the enumerated sets need not be disjoint anymore.

With the option `facade=False` the elements are wrapped in an object whose parent is the disjoint union itself. The wrapped object can then be recovered using the `value` attribute.

The two options can be combined.

The names of those options is imperfect, and subject to change in future versions. Feedback welcome.
EXAMPLES:

The input can be a list or a tuple of FiniteEnumeratedSets:

```python
sage: U1 = DisjointUnionEnumeratedSets((
.....:   FiniteEnumeratedSet([1,2,3]),
.....:   FiniteEnumeratedSet([4,5,6])))
sage: U1
Disjoint union of Family ({1, 2, 3}, {4, 5, 6})
sage: U1.list()
[1, 2, 3, 4, 5, 6]
sage: U1.cardinality()
6
```

The input can also be a dictionary:

```python
sage: U2 = DisjointUnionEnumeratedSets({1: FiniteEnumeratedSet([1,2,3]),
.....:   2: FiniteEnumeratedSet([4,5,6])})
sage: U2
Disjoint union of Finite family {1: {1, 2, 3}, 2: {4, 5, 6}}
sage: U2.list()
[1, 2, 3, 4, 5, 6]
sage: U2.cardinality()
6
```

However in that case the enumeration order is not specified.

In general the input can be any family:

```python
sage: U3 = DisjointUnionEnumeratedSets(
.....:   Family([2,3,4], Permutations, lazy=True))
sage: U3
Disjoint union of Lazy family (<class 'sage.combinat.permutation.Permutations'>
˓→(i))_{i in [2, 3, 4]}
sage: U3.cardinality()
32
sage: it = iter(U3)
sage: [next(it), next(it), next(it), next(it), next(it), next(it)]
[[1, 2], [2, 1], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1]]
sage: U3.unrank(18)
[2, 4, 1, 3]
```

This allows for infinite unions:

```python
sage: U4 = DisjointUnionEnumeratedSets(
.....:   Family(NonNegativeIntegers(), Permutations))
sage: U4
Disjoint union of Lazy family (<class 'sage.combinat.permutation.Permutations'>
˓→(i))_{i in Non negative integers}
sage: U4.cardinality()
+Infinity
sage: it = iter(U4)
sage: [next(it), next(it), next(it), next(it), next(it), next(it)]
[[], [1], [2, 1], [1, 2, 3], [1, 2, 3], [2, 1, 3], [2, 3, 1]]
sage: U4.unrank(18)
[2, 4, 1, 3]
```
Warning: Beware that some of the operations assume in that case that infinitely many of the enumerated sets are non empty.

Experimental options

We demonstrate the keepkey option:

```
sage: Ukeep = DisjointUnionEnumeratedSets(
    ....:     Family(list(range(4)), Permutations), keepkey=True)
sage: it = iter(Ukeep)
sage: [(next(it) for i in range(6))
[(0, []), (1, [1]), (2, [1, 2]), (2, [2, 1]), (3, [1, 2, 3]), (3, [1, 3, 2])]
sage: type(next(it)[1])
<class 'sage.combinat.permutation.StandardPermutations_n_with_category.element_class'>
```

We now demonstrate the facade option:

```
sage: UNoFacade = DisjointUnionEnumeratedSets(
    ....:     Family(list(range(4)), Permutations), facade=False)
sage: it = iter(UNoFacade)
sage: [(next(it) for i in range(6))
[[], [1], [1, 2], [2, 1], [1, 2, 3], [1, 3, 2]]
sage: el = next(it); el
[2, 1, 3]
sage: type(el)
<... 'sage.structure.element_wrapper.ElementWrapper'>
sage: el.parent() == UNoFacade
True
sage: elv = el.value; elv
[2, 1, 3]
sage: type(elv)
<class 'sage.combinat.permutation.StandardPermutations_n_with_category.element_class'>
```

The elements `el` of the disjoint union are simple wrapped elements. So to access the methods, you need to do `el.value`:

```
sage: el[0] # py2
Traceback (most recent call last):
...  TypeError: 'sage.structure.element_wrapper.ElementWrapper' object has no attribute '__getitem__'
sage: el[0] # py3
Traceback (most recent call last):
...  TypeError: 'sage.structure.element_wrapper.ElementWrapper' object is not subscriptable
sage: el.value[0]
2
```

Possible extensions: the current enumeration order is not suitable for unions of infinite enumerated sets (except possibly for the last one). One could add options to specify alternative enumeration orders (anti-diagonal, round robin, ...) to handle this case.

1.5. Disjoint union of enumerated sets 31
Inheriting from `DisjointUnionEnumeratedSets`

There are two different use cases for inheriting from `DisjointUnionEnumeratedSets`: writing a parent which happens to be a disjoint union of some known parents, or writing generic disjoint unions for some particular classes of `sage.categories.enumerated_setsEnumeratedSets`.

- In the first use case, the input of the `__init__` method is most likely different from that of `DisjointUnionEnumeratedSets`. Then, one simply writes the `__init__` method as usual:

  ```python
  sage: class MyUnion(DisjointUnionEnumeratedSets):
  ....:     def __init__(self):
  ....:         DisjointUnionEnumeratedSets.__init__(self, Family([1, 2], Permutations))
  sage: pp = MyUnion()
  sage: pp.list()
  [[1], [1, 2], [2, 1]]
  ```

  In case the `__init__` method takes optional arguments, or does some normalization on them, a specific method `__classcall_private__` is required (see the documentation of `UniqueRepresentation`).

- In the second use case, the input of the `__init__` method is the same as that of `DisjointUnionEnumeratedSets`; one therefore wants to inherit the `__classcall_private__` method as well, which can be achieved as follows:

  ```python
  sage: class UnionOfSpecialSets(DisjointUnionEnumeratedSets):
  ....:     __classcall_private__ = staticmethod(DisjointUnionEnumeratedSets.__classcall_private__)
  sage: psp = UnionOfSpecialSets(Family([1, 2], Permutations))
  sage: psp.list()
  [[1], [1, 2], [2, 1]]
  ```

`Element`()  

`an_element()`  

Return an element of this disjoint union, as per `Sets.ParentMethods.an_element()`.

**EXAMPLES:**

```python
sage: U4 = DisjointUnionEnumeratedSets(Family([3, 5, 7], Permutations))
sage: U4.an_element()
[1, 2, 3]
```

`cardinality()`  

Returns the cardinality of this disjoint union.

**EXAMPLES:**

For finite disjoint unions, the cardinality is computed by summing the cardinalities of the enumerated sets:

```python
sage: U = DisjointUnionEnumeratedSets(Family([0, 1, 2, 3], Permutations))
sage: U.cardinality()
10
```

For infinite disjoint unions, this makes the assumption that the result is infinite:

```python
sage: U = DisjointUnionEnumeratedSets(Family(NonNegativeIntegers(), Permutations))
```
Warning: As pointed out in the main documentation, it is possible to construct examples where this is incorrect:

```
sage: U = DisjointUnionEnumeratedSets(
    ....:     Family(NonNegativeIntegers(), lambda x: []))
sage: U.cardinality()  # Should be 0!
+Infinity
```

### 1.6 Enumerated set from iterator

**EXAMPLES:**

We build a set from the iterator `graphs` that returns a canonical representative for each isomorphism class of graphs:

```
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: E = EnumeratedSetFromIterator(
    ....:     graphs,
    ....:     name = "Graphs",
    ....:     category = InfiniteEnumeratedSets(),
    ....:     cache = True)
sage: E
Graphs
sage: E.unrank(0)
Graph on 0 vertices
sage: E.unrank(4)
Graph on 3 vertices
sage: E.cardinality()
+Infinity
sage: E.category()
Category of facade infinite enumerated sets
```

The module also provides decorator for functions and methods:

```
sage: from sage.sets.set_from_iterator import set_from_function
sage: @set_from_function
    ....: def f(n):
    ....:     return xsrange(n)
sage: f(3)
{0, 1, 2}
sage: f(5)
{0, 1, 2, 3, 4}
sage: f(100)
{0, 1, 2, 3, 4, ...}

sage: from sage.sets.set_from_iterator import set_from_method
sage: class A:
    ....:     @set_from_method
    ....:     def f(self, n):
    ....:         return xsrange(n)
```

(continues on next page)
class sage.sets.set_from_iterator.Decorator
    Bases: object

    Abstract class that manage documentation and sources of the wrapped object.

    The method needs to be stored in the attribute self.f

class sage.sets.set_from_iterator.DummyExampleForPicklingTest
    Bases: object

    Class example to test pickling with the decorator set_from_method.

    Warning: This class is intended to be used in doctest only.

EXAMPLES:

sage: from sage.sets.set_from_iterator import DummyExampleForPicklingTest
sage: DummyExampleForPicklingTest().f()
{10, 11, 12, 13, 14, ...}

f()
    Returns the set between self.start and self.stop.

    EXAMPLES:

    sage: from sage.sets.set_from_iterator import DummyExampleForPicklingTest
    sage: d = DummyExampleForPicklingTest()
    sage: d.f()
    {10, 11, 12, 13, 14, ...}
    sage: d.start = 4
    sage: d.stop = 200
    sage: d.f()
    {4, 5, 6, 7, 8, ...}

class sage.sets.set_from_iterator.EnumeratedSetFromIterator (f, args=None, kwds=None, name=None, category=None, cache=False)

    Bases: sage.structure.parent.Parent

    A class for enumerated set built from an iterator.

    INPUT:
    
    • f – a function that returns an iterable from which the set is built from
    • args – tuple – arguments to be sent to the function f
    • kwds – dictionary – keywords to be sent to the function f
    • name – an optional name for the set
• **category** – (default: None) an optional category for that enumerated set. If you know that your iterator will stop after a finite number of steps you should set it as `FiniteEnumeratedSets`, conversely if you know that your iterator will run over and over you should set it as `InfiniteEnumeratedSets`.

• **cache** – boolean (default: False) – Whether or not use a cache mechanism for the iterator. If True, then the function \( f \) is called only once.

**EXAMPLES:**

```python
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: E = EnumeratedSetFromIterator(graphs, args = (7,))
sage: E
{Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, ...}
sage: E.category()
Category of facade enumerated sets
```

The same example with a cache and a custom name:

```python
sage: E = EnumeratedSetFromIterator(graphs, args = (8,),
                                 category = FiniteEnumeratedSets(),
                                 name = "Graphs with 8 vertices",
                                 cache = True)
sage: E
Graphs with 8 vertices
data: E.unrank(3)
Graph on 8 vertices
sage: E.category()
Category of facade finite enumerated sets
```

**Note:** In order to make the `TestSuite` works, the elements of the set should have parents.

**clear_cache()**

Clear the cache.

**EXAMPLES:**

```python
sage: from itertools import count
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: E = EnumeratedSetFromIterator(count, args=(1,), cache=True)
sage: e1 = E._cache
sage: e1
lazy list [1, 2, 3, ...]
sage: E.clear_cache()
sage: E._cache
lazy list [1, 2, 3, ...]
sage: e1 is E._cache
False
```

**is_parent_of(x)**

Test whether \( x \) is in \( self \).

If the set is infinite, only the answer `True` should be expected in finite time.

**EXAMPLES:**
sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: P = Partitions(12,min_part=2,max_part=5)
    sage: E = EnumeratedSetFromIterator(P.__iter__)  
    sage: P([5,5,2]) in E
    True

unrank(i)
Returns the element at position i.

EXAMPLES:

sage: from sage.sets.set_from_iterator import EnumeratedSetFromIterator
sage: E = EnumeratedSetFromIterator(graphs, args=(8,), cache=True)
    sage: F = EnumeratedSetFromIterator(graphs, args=(8,), cache=False)
    sage: E.unrank(2)
    Graph on 8 vertices
    sage: E.unrank(2) == F.unrank(2)
    True

class sage.sets.set_from_iterator.EnumeratedSetFromIterator_function_decorator (f=None,
name=None,**options)

Bases: sage.sets.set_from_iterator.Decorator

Decorator for EnumeratedSetFromIterator.
Name could be string or a function (args,kwds) -> string.

Warning: If you are going to use this with the decorator cached_function, you must place the
cached_function first. See the example below.

EXAMPLES:

sage: from sage.sets.set_from_iterator import set_from_function
sage: @set_from_function
    ....: def f(n):
    ....:     for i in range(n):
    ....:         yield i**2 + i + 1
sage: f(3)
{1, 3, 7}
    sage: f(100)
{1, 3, 7, 13, 21, ...}

To avoid ambiguity, it is always better to use it with a call which provides optional global initialization for the
call to EnumeratedSetFromIterator:

sage: @set_from_function(category=InfiniteEnumeratedSets())
    ....: def Fibonacci():
    ....:     a = 1; b = 2
    ....:     while True:
    ....:         yield a
    ....:         a, b = b, a + b
sage: F = Fibonacci()
    sage: F
{1, 2, 3, 5, 8, ...}

(continues on next page)
A simple example with many options:

```python
sage: @set_from_function
    ...:     def f(m, n):
    ...:         return xsrange(m,n+1)
sage: E = f(3,10); E
From 3 to 10
sage: E.list()
[3, 4, 5, 6, 7, 8, 9, 10]
sage: E = f(1,100); E
From 1 to 100
sage: E.cardinality()
100
sage: f(n=100,m=1) == E
True
```

An example which mixes together `set_from_function` and `cached_method`:

```python
sage: @cached_function @set_from_function
    ...:     def Graphs(n):
    ...:         return graphs(n)
sage: Graphs(10)
Graphs on 10 vertices
sage: Graphs(10).unrank(0)
Graph on 10 vertices
sage: Graphs(10) is Graphs(10)
True
```

The `cached_function` must go first:

```python
sage: @set_from_function
    ...:     def Graphs(n):
    ...:         return graphs(n)
sage: Graphs(10)
Graphs on 10 vertices
sage: Graphs(10).unrank(0)
Graph on 10 vertices
sage: Graphs(10) is Graphs(10)
False
```

```python
class sage.sets.set_from_iterator.EnumeratedSetFromIterator_method_caller (inst, f, name=None, **options)
Bases: sage.sets.set_from_iterator.Decorator
Caller for decorated method in class.
```

1.6. Enumerated set from iterator 37
INPUT:

- \texttt{inst} – an instance of a class
- \texttt{f} – a method of a class of \texttt{inst} (and not of the instance itself)
- \texttt{name} – optional – either a string (which may contains substitution rules from argument or a function \texttt{args,kwds} \rightarrow \texttt{string}.
- \texttt{options} – any option accepted by \texttt{EnumeratedSetFromIterator}

\begin{verbatim}
class sage.sets.set_from_iterator.EnumeratedSetFromIterator_method_decorator(f=None, **options):
    # Decorator for enumerated set built from a method.
    INPUT:
    - \texttt{f} – Optional function from which are built the enumerated sets at each call
    - \texttt{name} – Optional string (which may contains substitution rules from argument) or a function \texttt{(args, kwds) \rightarrow string}.
    - any option accepted by \texttt{EnumeratedSetFromIterator}.

    EXAMPLES:
    sage: from sage.sets.set_from_iterator import set_from_method
    sage: class A:
    ....:     def n(self): return 12
    ....:     @set_from_method
    ....:     def f(self): return xsrange(self.n())
    sage: a = A()
    sage: print(a.f.__class__)
    <class 'sage.sets.set_from_iterator.EnumeratedSetFromIterator_method_caller'>
    sage: a.f()
    {0, 1, 2, 3, 4, ...}
    sage: A.f(a)
    {0, 1, 2, 3, 4, ...}

    A more complicated example with a parametrized name:
    sage: class B:
    ....:     @set_from_method(name = "Graphs(%(n)d)",
    ....:                        category = FiniteEnumeratedSets())
    ....:     def graphs(self, n):
    ....:         return graphs(n)
    sage: b = B()
    sage: G3 = b.graphs(3)
    sage: G3
    Graphs(3)
    sage: G3.cardinality()
    4
    sage: G3.category()
    Category of facade finite enumerated sets
    sage: B.graphs(b,3)
    Graphs(3)

    And a last example with a name parametrized by a function:
    
    sage: class C:
    ....:     @set_from_method(name = "Graphs\text{\ {\tt\%}(\%(n)\d)\}s\}",
    ....:                        category = FiniteEnumeratedSets())
    ....:     def graphs(self, n):
    ....:         return graphs(n)
    sage: c = C()
    sage: G4 = c.graphs(4)
    sage: G4
    Graphs\text{\ {\tt\%}(\%(n)\d)\}s\}s(4)
    sage: G4.cardinality()
    20
    sage: G4.category()
    Category of facade finite enumerated sets
    sage: C.graphs(c,4)
    Graphs\text{\ {\tt\%}(\%(n)\d)\}s\}s(4)

\end{verbatim}
```
sage: class D():
.....:    def __init__(self, name): self.name = str(name)
.....:    def __str__(self): return self.name
.....:    @set_from_method()
.....:        name = lambda self,n: str(self)*n,
.....:        category = FiniteEnumeratedSets())
.....:    def subset(self, n):
.....:        return xsrange(n)

sage: d = D('a')

sage: E = d.subset(3); E
[0, 1, 2]

sage: F = d.subset(n=10); F
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Todo: It is not yet possible to use set_from_method in conjunction with cached_method.
```

sage.sets.set_from_iterator.set_from_function
alias of sage.sets.set_from_iterator.EnumeratedSetFromFunction_decorator

sage.sets.set_from_iterator.set_from_method
alias of sage.sets.set_from_iterator.EnumeratedSetFromMethod_decorator

### 1.7 Finite Enumerated Sets

**class** sage.sets.finite_enumerated_set.FiniteEnumeratedSet(elements)**

**Bases:** sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent Parent

A class for finite enumerated set.

Returns the finite enumerated set with elements in elements where element is any (finite) iterable object.

The main purpose is to provide a variant of list or tuple, which is a parent with an interface consistent with EnumeratedSets and has unique representation. The list of the elements is expanded in memory.

**EXAMPLES:**

```
sage: S = FiniteEnumeratedSet([1, 2, 3])
sage: S
{1, 2, 3}
sage: S.list()  
[1, 2, 3]
sage: S.cardinality()  
3
sage: S.random_element()  
1
sage: S.first()  
1
sage: S.category()  
Category of facade finite enumerated sets
sage: TestSuite(S).run()```
Note that being an enumerated set, the result depends on the order:

```python
sage: S1 = FiniteEnumeratedSet((1, 2, 3))
sage: S1
{1, 2, 3}
sage: S1.list()
[1, 2, 3]
sage: S1 == S
True
sage: S2 = FiniteEnumeratedSet((2, 1, 3))
sage: S2 == S
False
```

As an abuse, repeated entries in elements are allowed to model multisets:

```python
sage: S1 = FiniteEnumeratedSet((1, 2, 1, 2, 2, 3))
sage: S1
{1, 2, 1, 2, 2, 3}
```

Finally, the elements are not aware of their parent:

```python
sage: S.first().parent()
Integer Ring
```

- **an_element()**
  - Returns the first element of the enumeration or raise an EmptySetError if the set is empty.
  - **EXAMPLES:**
    ```python
    sage: S = FiniteEnumeratedSet('abc')
    sage: S.first()
    'a'
    ```

- **cardinality()**
  - Returns the index of x in this finite enumerated set.
  - **EXAMPLES:**
    ```python
    sage: S = FiniteEnumeratedSet(['a','b','c'])
    sage: S.index('b')
    1
    ```

- **is_parent_of(x)**
  - Returns the last element of the iteration or raise an EmptySetError if the set is empty.
  - **EXAMPLES:**
    ```python
    sage: S = FiniteEnumeratedSet([0,'a',1.23, 'd'])
    sage: S.last()
    'd'
    ```

- **list()**
  - Return a random element.
EXAMPLES:

```
sage: S = FiniteEnumeratedSet('abc')
sage: S.random_element()  # random
'b'
```

`rank (x)`

Returns the index of `x` in this finite enumerated set.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet(['a','b','c'])
sage: S.index('b')
1
```

`unrank (i)`

Return the element at position `i`.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet([1,'a',-51])
sage: S[0], S[1], S[2]
(1, 'a', -51)
sage: S[3]
Traceback (most recent call last):
  ... IndexError: tuple index out of range
sage: S[-1], S[-2], S[-3]
(-51, 'a', 1)
sage: S[-4]
Traceback (most recent call last):
  ... IndexError: list index out of range
```

### 1.8 Recursively enumerated set

A set `S` is called recursively enumerable if there is an algorithm that enumerates the members of `S`. We consider here the recursively enumerated sets that are described by some `seeds` and a successor function `successors`. The successor function may have some structure (symmetric, graded, forest) or not. The elements of a set having a symmetric, graded or forest structure can be enumerated uniquely without keeping all of them in memory. Many kinds of iterators are provided in this module: depth first search, breadth first search or elements of given depth.


See documentation of `RecursivelyEnumeratedSet()` below for the description of the inputs.

AUTHORS:

- Sébastien Labbé, April 2014, at Sage Days 57, Cernay-la-ville

EXAMPLES:
1.8.1 No hypothesis on the structure

What we mean by “no hypothesis” is that the set is not known to be a forest, symmetric, or graded. However, it may have other structure, like not containing an oriented cycle, that does not help with the enumeration.

In this example, the seed is 0 and the successor function is either $+2$ or $+3$. This is the set of non negative linear combinations of 2 and 3:

```python
sage: succ = lambda a: [a+2, a+3]
sage: C = RecursivelyEnumeratedSet([0], succ)
sage: C
A recursively enumerated set (breadth first search)
```

Breadth first search:

```python
sage: it = C.breadth_first_search_iterator()
sage: [next(it) for _ in range(10)]
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Depth first search:

```python
sage: it = C.depth_first_search_iterator()
sage: [next(it) for _ in range(10)]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

1.8.2 Symmetric structure

The origin $(0, 0)$ as seed and the upper, lower, left and right lattice point as successor function. This function is symmetric since $p$ is a successor of $q$ if and only if $q$ is a successor or $p$:

```python
sage: succ = lambda a: [(a[0]-1,a[1]), (a[0],a[1]-1), (a[0]+1,a[1]), (a[0],a[1]+1)]
sage: seeds = [(0,0)]
sage: C = RecursivelyEnumeratedSet(seeds, succ, structure='symmetric', enumeration='depth')
sage: C
A recursively enumerated set with a symmetric structure (depth first search)
```

In this case, depth first search is the default enumeration for iteration:

```python
sage: it_depth = iter(C)
sage: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9)]
```

Breadth first search:

```python
sage: it_breadth = C.breadth_first_search_iterator()
sage: [next(it_breadth) for _ in range(13)]
[(0, 0), (-1, 0), (0, -1), (1, 0), (0, 1), (-2, 0), (-1, -1), (-1, 1), (0, -2), (1, -1), (2, 0), (1, 1), (0, 2)]
```

Levels (elements of given depth):

```python
sage: sorted(C.graded_component(0))
[(0, 0)]
sage: sorted(C.graded_component(1))
(continues on next page)
1.8.3 Graded structure

Identity permutation as seed and \texttt{permutohedron\_succ} as successor function:

\begin{verbatim}
sage: succ = attrcall("permutohedron\_succ")
sage: seed = [Permutation([1..5])]
sage: R = RecursivelyEnumeratedSet(seed, succ, structure='graded')
sage: R
A recursively enumerated set with a graded structure (breadth first search)
\end{verbatim}

Depth first search iterator:

\begin{verbatim}
sage: it_depth = R.depth_first_search_iterator()
sage: [next(it_depth) for _ in range(5)]
[[1, 2, 3, 4, 5],
 [1, 2, 3, 5, 4],
 [1, 2, 5, 3, 4],
 [1, 2, 5, 4, 3],
 [1, 5, 2, 4, 3]]
\end{verbatim}

Breadth first search iterator:

\begin{verbatim}
sage: it_breadth = R.breadth_first_search_iterator()
sage: [next(it_breadth) for _ in range(5)]
[[1, 2, 3, 4, 5],
 [2, 1, 3, 4, 5],
 [1, 3, 2, 4, 5],
 [1, 2, 4, 3, 5],
 [1, 2, 3, 5, 4]]
\end{verbatim}

Elements of given depth iterator:

\begin{verbatim}
sage: sorted(R.elements_of_depth_iterator(9))
[[4, 5, 3, 2, 1], [5, 3, 4, 2, 1], [5, 4, 2, 3, 1], [5, 4, 3, 1, 2]]
sage: list(R.elements_of_depth_iterator(10))
[[5, 4, 3, 2, 1]]
\end{verbatim}

Graded components (set of elements of the same depth):

\begin{verbatim}
sage: sorted(R.graded_component(0))
[[1, 2, 3, 4, 5]]
sage: sorted(R.graded_component(1))
[[1, 2, 3, 5, 4], [1, 2, 4, 3, 5], [1, 3, 2, 4, 5], [2, 1, 3, 4, 5]]
sage: sorted(R.graded_component(9))
[[4, 5, 3, 2, 1], [5, 3, 4, 2, 1], [5, 4, 2, 3, 1], [5, 4, 3, 1, 2]]
sage: sorted(R.graded_component(10))
[[5, 4, 3, 2, 1]]
\end{verbatim}
1.8.4 Forest structure

The set of words over the alphabet \( \{a, b\} \) can be generated from the empty word by appending letter \( a \) or \( b \) as a successor function. This set has a forest structure:

```python
sage: seeds = ['']
sage: succ = lambda w: [w+'a', w+'b']
sage: C = RecursivelyEnumeratedSet(seeds, succ, structure='forest')
sage: C
An enumerated set with a forest structure
```

Depth first search iterator:

```python
sage: it = C.depth_first_search_iterator()
sage: [next(it) for _ in range(6)]
['', 'a', 'aa', 'aaa', 'aaaa', 'aaaaa']
```

Breadth first search iterator:

```python
sage: it = C.breadth_first_search_iterator()
sage: [next(it) for _ in range(6)]
['', 'a', 'b', 'aa', 'ab', 'ba']
```

1.8.5 Example: Forest structure

This example was provided by Florent Hivert.

How to define a set using those classes?

Only two things are necessary to define a set using a `RecursivelyEnumeratedSet` object (the other classes being very similar):

For the previous example, the two necessary pieces of information are:

- the initial element ":
- the function:

  ```python
  lambda x: [x + letter for letter in ['a', 'b', 'c']]
  ```

This would actually describe an infinite set, as such rules describes “all words” on 3 letters. Hence, it is a good idea to replace the function by:

```python
lambda x: [x + letter for letter in ['a', 'b', 'c']] if len(x) < 2 else []
```

or even:

```python
sage: def children(x):
.....: if len(x) < 2:
.....: for letter in ['a', 'b', 'c']:
.....: yield x+letter
```

We can then create the `RecursivelyEnumeratedSet` object with either:
1.8.6 Example: Forest structure 2

This example was provided by Florent Hivert.

Here is a little more involved example. We want to iterate through all permutations of a given set $S$. One solution is to take elements of $S$ one by one and insert them at every position. So a node of the generating tree contains two pieces of information:

- the list $\text{lst}$ of already inserted element;
- the set $\text{st}$ of the yet to be inserted element.

We want to generate a permutation only if $\text{st}$ is empty (leaves on the tree). Also suppose for the sake of the example, that instead of list we want to generate tuples. This selection of some nodes and final mapping of a function to the element is done by the $\text{post_process} = f$ argument. The convention is that the generated elements are the $s := f(n)$, except when $s$ not $\text{None}$ when no element is generated at all. Here is the code:

```python
sage: def children(node):
    (lst, st) = node
    st = set(st) # make a copy
    if st:
        el = st.pop()
        for i in range(0, len(lst)+1):
            yield (lst[0:i]+[el]+lst[i:], st)
sage: list(children(([1,2], {3,7,9})))
[[[9, 1, 2], {3, 7}), ([1, 9, 2], {3, 7}), ([1, 2, 9], {3, 7})]
sage: def post_process(node):
    (l, s) = node
    return tuple(l) if not s else None
sage: S = RecursivelyEnumeratedSet( [([], {1,3,6,8})],
    children, post_process=post_process,
    structure='forest', enumeration='depth',
    category=FiniteEnumeratedSets())
sage: S.list()
[(6, 3, 1, 8), (3, 6, 1, 8), (3, 1, 6, 8), (6, 3, 1, 8), (6, 1, 3, 8),
(1, 6, 3, 8), (1, 3, 6, 8), (1, 3, 8, 6), (6, 1, 8, 3), (1, 6, 8, 3),
(1, 8, 6, 3), (1, 8, 3, 6), (6, 3, 8, 1), (3, 6, 8, 1), (3, 8, 6, 1),
(3, 8, 1, 6), (6, 8, 3, 1), (8, 6, 3, 1), (8, 3, 6, 1), (8, 3, 1, 6),
(6, 8, 1, 3), (8, 6, 1, 3), (8, 1, 6, 3), (8, 1, 3, 6)]
sage: S.cardinality()
24
```
Return a recursively enumerated set.

A set $S$ is called recursively enumerable if there is an algorithm that enumerates the members of $S$. We consider here the recursively enumerated set that are described by some seeds and a successor function successors.

Let $U$ be a set and $\text{successors} : U \rightarrow 2^U$ be a successor function associating to each element of $U$ a subset of $U$. Let seeds be a subset of $U$. Let $S \subseteq U$ be the set of elements of $U$ that can be reached from a seed by applying recursively the successors function. This class provides different kinds of iterators (breadth first, depth first, elements of given depth, etc.) for the elements of $S$.

See Wikipedia article Recursively_enumerable_set.

INPUT:

- **seeds** – list (or iterable) of hashable objects
- **successors** – function (or callable) returning a list (or iterable) of hashable objects
- **structure** – string (optional, default: None), structure of the set, possible values are:
  - None – nothing is known about the structure of the set.
  - 'forest' – if the successors function generates a forest, that is, each element can be reached uniquely from a seed.
  - 'graded' – if the successors function is graded, that is, all paths from a seed to a given element have equal length.
  - 'symmetric' – if the relation is symmetric, that is, $y \in \text{successors}(x)$ if and only if $x \in \text{successors}(y)$
- **enumeration** – 'depth', 'breadth', 'naive' or None (optional, default: None). The default enumeration for the \texttt{__iter__} function.
- **max_depth** – integer (optional, default: float("inf")), limit the search to a certain depth, currently works only for breadth first search
- **post_process** – (optional, default: None), for forest only
- **facade** – (optional, default: None)
- **category** – (optional, default: None)

EXAMPLES:

A recursive set with no other information:

```python
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: C
A recursively enumerated set (breadth first search)
sage: it = iter(C)
sage: [next(it) for _ in range(10)]
[0, 3, 5, 6, 8, 10, 9, 11, 13, 15]
```

A recursive set with a forest structure:
A recursive set given by a symmetric relation:

```python
sage: f = lambda a: [a-1,a+1]
sage: C = RecursivelyEnumeratedSet([10, 15], f, structure='symmetric')
sage: C
A recursively enumerated set with a symmetric structure (breadth first search)
sage: it = iter(C)
sage: [next(it) for _ in range(7)]
[10, 15, 9, 11, 14, 16, 8]
```

A recursive set given by a graded relation:

```python
sage: f = lambda a: [a+1, a+I]
sage: C = RecursivelyEnumeratedSet([0], f, structure='graded')
sage: C
A recursively enumerated set with a graded structure (breadth first search)
sage: it = iter(C)
sage: [next(it) for _ in range(7)]
[0, 1, I, 2, I + 1, 2*I, 3]
```

**Warning:** If you do not set the good structure, you might obtain bad results, like elements generated twice:

```python
sage: f = lambda a: [a-1,a+1]
sage: C = RecursivelyEnumeratedSet([0], f, structure='graded')
sage: it = iter(C)
sage: [next(it) for _ in range(7)]
[0, -1, 1, -2, 0, 2, -3]
```

```python
class sage.sets.recursivelyEnumeratedSet.RecursivelyEnumeratedSet_forest(roots=None, children=None, post_process=None, algorithm='depth', facade=None, category=None)

Bases: sage.structure.parent.Parent

The enumerated set of the nodes of the forest having the given roots, and where children(x) returns the children of the node x of the forest.
```
See also \texttt{sage.combinat.backtrack.GenericBacktracker, RecursivelyEnumeratedSet\_graded}, and \texttt{RecursivelyEnumeratedSet\_symmetric}.

**INPUT:**

- \texttt{roots} – a list (or iterable)
- \texttt{children} – a function returning a list (or iterable, or iterator)
- \texttt{post\_process} – a function defined over the nodes of the forest (default: no post processing)
- \texttt{algorithm} – 'depth' or 'breadth' (default: 'depth')
- \texttt{category} – a category (default: EnumeratedSets)

The option \texttt{post\_process} allows for customizing the nodes that are actually produced. Furthermore, if \(f(x)\) returns \texttt{None}, then \(x\) won’t be output at all.

**EXAMPLES:**

We construct the set of all binary sequences of length at most three, and list them:

```python
sage: from sage.sets.recursivelyEnumeratedSet import RecursivelyEnumeratedSet_forest
sage: S = RecursivelyEnumeratedSet_forest([[]],
....: lambda l: [l+[0], l+[1]] if len(l) < 3 else [],
....: category=FiniteEnumeratedSets())
sage: S.list()
[[],
 [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0], [0, 1, 1],
 [1], [1, 0], [1, 0, 0], [1, 0, 1], [1, 1], [1, 1, 0], [1, 1, 1]]
```

\texttt{RecursivelyEnumeratedSet\_forest} needs to be explicitly told that the set is finite for the following to work:

```python
sage: S.category()
Category of finite enumerated sets
sage: S.cardinality()
15
```

We proceed with the set of all lists of letters in \(0, 1, 2\) without repetitions, ordered by increasing length (i.e. using a breadth first search through the tree):

```python
sage: from sage.sets.recursivelyEnumeratedSet import RecursivelyEnumeratedSet_forest
sage: tb = RecursivelyEnumeratedSet_forest([[]],
....: lambda l: [l + [i] for i in range(3) if i not in l],
....: algorithm = 'breadth',
....: category=FiniteEnumeratedSets())
sage: tb[0]
[]
sage: tb.cardinality()
16
sage: list(tb)
[[],
 [0], [1], [2],
 [0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1],
 [0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
```

For infinite sets, this option should be set carefully to ensure that all elements are actually generated. The following example builds the set of all ordered pairs \((i, j)\) of nonnegative integers such that \(j \leq 1\):
With a depth first search, only the elements of the form \((i, 0)\) are generated:

```sage
sage: depth_search = I.depth_first_search_iterator()
sage: [next(depth_search) for i in range(7)]
[(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0)]
```

Using instead breadth first search gives the usual anti-diagonal iterator:

```sage
sage: breadth_search = I.breadth_first_search_iterator()
sage: [next(breadth_search) for i in range(15)]
[(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), (3, 0), (2, 1), (1, 2), (0, 3), (4, 0), (3, 1), (2, 2), (1, 3), (0, 4)]
```

### Deriving subclasses

The class of a parent \(A\) may derive from `RecursivelyEnumeratedSet_forest` so that \(A\) can benefit from enumeration tools. As a running example, we consider the problem of enumerating integers whose binary expansion have at most three nonzero digits. For example, \(3 = 2^1 + 2^0\) has two nonzero digits. \(15 = 2^3 + 2^2 + 2^1 + 2^0\) has four nonzero digits. In fact, 15 is the smallest integer which is not in the enumerated set.

To achieve this, we use `RecursivelyEnumeratedSet_forest` to enumerate binary tuples with at most three nonzero digits, apply a post processing to recover the corresponding integers, and discard tuples finishing by zero.

A first approach is to pass the roots and children functions as arguments to `RecursivelyEnumeratedSet_forest.__init__()`:

```sage
sage: from sage.sets.recursivelyEnumeratedSet import RecursivelyEnumeratedSet_forest
sage: class A(UniqueRepresentation, RecursivelyEnumeratedSet_forest):
    ....:    def __init__(self):
    ....:        RecursivelyEnumeratedSet_forest.__init__(self, [],
    ....:            lambda x : [x+(0,), x+(1,)] if sum(x) < 3 else [],
    ....:            lambda x : sum(x[i]*2^i for i in range(len(x))) if sum(x) != 0
    ....:                and x[-1] != 0 else None,
    ....:            algorithm = 'breadth',
    ....:            category=InfiniteEnumeratedSets())
```

An alternative approach is to implement roots and children as methods of the subclass (in fact they could also be attributes of \(A\)). Namely, \(A.roots()\) must return an iterable containing the enumeration generators,
and `A.children(x)` must return an iterable over the children of `x`. Optionally, `A` can have a method or attribute such that `A.post_process(x)` returns the desired output for the node `x` of the tree:

```python
sage: from sage.sets.recursively_enumerated_set import RecursivelyEnumeratedSet_forest
sage: class A(UniqueRepresentation, RecursivelyEnumeratedSet_forest):
    ....:    def __init__(self):
    ....:        RecursivelyEnumeratedSet_forest.__init__(self, algorithm = 'breadth_first_search_iterator',
    ....:        category=InfiniteEnumeratedSets())
    ....:    def roots(self):
    ....:        return [()]
    ....:    def children(self, x):
    ....:        if sum(x) < 3:
    ....:            return [x+(0,), x+(1,)]
    ....:        else:
    ....:            return []
    ....:    def post_process(self, x):
    ....:        if sum(x) == 0 or x[-1] == 0:
    ....:            return None
    ....:        else:
    ....:            return sum(x[i]*2^i for i in range(len(x)))
```

```python
sage: MyForest = A(); MyForest
An enumerated set with a forest structure
sage: MyForest.category()
Category of infinite enumerated sets
sage: p = iter(MyForest)
sage: [next(p) for i in range(30)]
[1, 2, 3, 4, 6, 5, 7, 8, 12, 10, 14, 9, 13, 11, 16, 24, 20, 28, 18, 26, 22, 17, 25, 21, 19, 32, 48, 40, 56, 36]
```

**Warning:** A `RecursivelyEnumeratedSet_forest` instance is picklable if and only if the input functions are themselves picklable. This excludes anonymous or interactively defined functions:

```python
sage: def children(x):
    ....:    return [x+1]
sage: S = RecursivelyEnumeratedSet_forest( [1], children, category=InfiniteEnumeratedSets())
sage: dumps(S)
Traceback (most recent call last):
  ... PicklingError: Can't pickle <...function...>: attribute lookup ... failed
```

Let us now fake `children` being defined in a Python module:

```python
sage: import __main__
sage: __main__.children = children
sage: S = RecursivelyEnumeratedSet_forest( [1], children, category=InfiniteEnumeratedSets())
sage: loads(dumps(S))
An enumerated set with a forest structure
```

`breadth_first_search_iterator()`

Return a breadth first search iterator over the elements of `self`
EXAMPLES:

```python
sage: from sage.sets.recursively enumerated_set import *
...
RecursivelyEnumeratedSet_forest
sage: f = RecursivelyEnumeratedSet_forest([],
....: lambda l: [l+[0], l+[1]] if len(l) < 3 else [])
```

```python
sage: list(f.breadth_first_search_iterator())
[[], [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0], [0, 1, 1], [0, 1, 2], [1], [1, 0], [1, 0, 0], [1, 0, 1], [1, 1], [1, 1, 0], [1, 1, 1]]
```

```python
sage: S = RecursivelyEnumeratedSet_forest([(0,0)],
....: lambda x : [(x[0], x[1]+1)] if x[1] != 0 else [(x[0]+1,0), (x[0],1)],
....: post_process = lambda x: x if ((is_prime(x[0]) and is_prime(x[1])) and
....: ((x[0] - x[1]) == 2)) else None)
```

```python
sage: p = S.breadth_first_search_iterator()
```

```python
sage: [next(p), next(p), next(p), next(p), next(p), next(p), next(p)]
[(5, 3), (7, 5), (13, 11), (19, 17), (31, 29), (43, 41), (61, 59)]
```

children(x)

Return the children of the element x

The result can be a list, an iterable, an iterator, or even a generator.

EXAMPLES:

```python
sage: from sage.sets.recursively enumerated_set import *
...
RecursivelyEnumeratedSet_forest
sage: I = RecursivelyEnumeratedSet_forest([(0,0)], lambda l: [(l[0]+1, l[1]), (l[0], 1)] if l[1] == 0 else [(l[0]+1, l[1])])
```

```python
sage: [i for i in I.children((0,0))]
[(1, 0), (0, 1)]
```

```python
sage: [i for i in I.children((1,0))]
[(2, 0), (1, 1)]
```

```python
sage: [i for i in I.children((1,1))]
[(1, 2)]
```

```python
sage: [i for i in I.children((4,1))]
[(4, 2)]
```

```python
sage: [i for i in I.children((4,0))]
[(5, 0), (4, 1)]
```

depth_first_search_iterator()

Return a depth first search iterator over the elements of self

EXAMPLES:

```python
sage: from sage.sets.recursively enumerated_set import *
...
RecursivelyEnumeratedSet_forest
sage: f = RecursivelyEnumeratedSet_forest([],
....: lambda l: [l+[0], l+[1]] if len(l) < 3 else [])
```

```python
sage: list(f.depth_first_search_iterator())
[[], [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0], [0, 1, 1], [1], [1, 0], [1, 0, 0], [1, 0, 1], [1, 1], [1, 1, 0], [1, 1, 1]]
```

elements_of_depth_iterator(depth=0)

Return an iterator over the elements of self of given depth. An element of depth n can be obtained applying n times the children function from a root.

EXAMPLES:
map_reduce (map_function=None, reduce_function=None, reduce_init=None)  
Apply a Map/Reduce algorithm on self

INPUT:

• map_function – a function from the element of self to some set with a reduce operation (e.g.: a monoid). The default value is the constant function 1.
• reduce_function – the reduce function (e.g.: the addition of a monoid). The default value is +.
• reduce_init – the initialisation of the reduction (e.g.: the neutral element of the monoid). The default value is 0.

Note: the effect of the default values is to compute the cardinality of self.

EXAMPLES:

sage: seeds = [([i], i) for i in range(1, 10)]  
sage: def succ(t):  
....:     list, sum, last = t  
....:     return [(list + [i], sum + i, i) for i in range(1, last)]  
sage: F = RecursivelyEnumeratedSet(seeds, succ,  
....:     structure='forest', enumeration='depth')

sage: y = var('y')  
sage: def map_function(t):  
....:     li, sum, _ = t  
....:     return y ** sum  
sage: reduce_function = lambda x, y: x + y  
sage: F.map_reduce(map_function, reduce_function, 0)

```
y^45 + y^44 + y^43 + 2*y^42 + 2*y^41 + 3*y^40 + 4*y^39 + 5*y^38 + 6*y^37 +  
+ 8*y^36 + 9*y^35 + 10*y^34 + 12*y^33 + 13*y^32 + 15*y^31 + 17*y^30 + 18*y^29 +  
+ 19*y^28 + 21*y^27 + 21*y^26 + 22*y^25 + 23*y^24 + 23*y^23 + 23*y^22 +  
+ 23*y^21 + 22*y^20 + 21*y^19 + 21*y^18 + 19*y^17 + 18*y^16 + 17*y^15 + 15*y^14 +  
+ 13*y^13 + 12*y^12 + 10*y^11 + 9*y^10 + 8*y^9 + 6*y^8 + 5*y^7 + 4*y^6 +  
+ 3*y^5 + 2*y^4 + 2*y^3 + y^2 + y
```

Here is an example with the default values:
sage: F.map_reduce()
511

See also:
sage.parallel.map_reduce

roots()
Return an iterable over the roots of self.

EXAMPLES:

```
sage: from sage.sets.recursively_enumerated_set import RecursivelyEnumeratedSet_forest
sage: I = RecursivelyEnumeratedSet_forest([(0,0)], lambda l: [ (l[0]+1, l[1]), (l[0], 1) ] if l[1] == 0 else [ (l[0], l[1]+1) ])
sage: [i for i in I.roots()]
[(0, 0)]
sage: I = RecursivelyEnumeratedSet_forest([(0,0),(1,1)], lambda l: [ (l[0]+1, l[1]), (l[0], 1) ] if l[1] == 0 else [ (l[0], l[1]+1) ])
sage: [i for i in I.roots()]
[(0, 0), (1, 1)]
```

class sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic
Bases: sage.structure.parent.Parent
A generic recursively enumerated set.

For more information, see `RecursivelyEnumeratedSet()`.

EXAMPLES:

```
sage: f = lambda a: [a+1]
```

Different structure for the sets:

```
sage: R = RecursivelyEnumeratedSet([0], f, structure=None)
A recursively enumerated set (breadth first search)
sage: R = RecursivelyEnumeratedSet([0], f, structure='graded')
A recursively enumerated set with a graded structure (breadth first search)
sage: R = RecursivelyEnumeratedSet([0], f, structure='symmetric')
A recursively enumerated set with a symmetric structure (breadth first search)
sage: R = RecursivelyEnumeratedSet([0], f, structure='forest')
An enumerated set with a forest structure
```

Different default enumeration algorithms:

```
sage: R = RecursivelyEnumeratedSet([0], f, enumeration='breadth')
A recursively enumerated set (breadth first search)
sage: R = RecursivelyEnumeratedSet([0], f, enumeration='naive')
A recursively enumerated set (naive search)
sage: R = RecursivelyEnumeratedSet([0], f, enumeration='depth')
A recursively enumerated set (depth first search)
```

```
breadth_first_search_iterator(max_depth=None)
Iterate on the elements of self (breadth first).
This code remembers every element generated.
The elements are guaranteed to be enumerated in the order in which they are first visited (left-to-right traversal).
```

1.8. Recursively enumerated set
INPUT:

- **max_depth** – (default: self._max_depth) specifies the maximal depth to which elements are computed

EXAMPLES:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: it = C.breadth_first_search_iterator()
sage: [next(it) for _ in range(10)]
[0, 3, 5, 6, 8, 10, 9, 11, 13, 15]
```

**depth_first_search_iterator()**

Iterate on the elements of self (depth first).

This code remembers every elements generated.

The elements are traversed right-to-left, so the last element returned by the successor function is visited first.

See Wikipedia article Depth-first_search.

EXAMPLES:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: it = C.depth_first_search_iterator()
sage: [next(it) for _ in range(10)]
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

**elements_of_depth_iterator(depth)**

Iterate over the elements of self of given depth.

An element of depth $n$ can be obtained applying $n$ times the successor function to a seed.

INPUT:

- **depth** – integer

OUTPUT:

An iterator.

EXAMPLES:

```
sage: f = lambda a: [a-1, a+1]
sage: S = RecursivelyEnumeratedSet([5, 10], f, structure='symmetric')
sage: it = S.elements_of_depth_iterator(2)
sage: sorted(it)
[3, 7, 8, 12]
```

**graded_component(depth)**

Return the graded component of given depth.

This method caches each lower graded component.

A graded component is a set of elements of the same depth where the depth of an element is its minimal distance to a root.

It is currently implemented only for graded or symmetric structure.

INPUT:
• depth – integer

OUTPUT:
A set.

EXAMPLES:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: C.graded_component(0)
Traceback (most recent call last):
...
NotImplementedError: graded_component_iterator method currently implemented only for graded or symmetric structure
```

**graded_component_iterator()**
Iterate over the graded components of self.

A graded component is a set of elements of the same depth.

It is currently implemented only for graded or symmetric structure.

OUTPUT:
An iterator of sets.

EXAMPLES:

```
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: it = C.graded_component_iterator()  # todo: not implemented
```

**naive_search_iterator()**
Iterate on the elements of self (in no particular order).

This code remembers every elements generated.

**seeds()**
Return an iterable over the seeds of self.

EXAMPLES:

```
sage: R = RecursivelyEnumeratedSet([1], lambda x: [x+1, x-1])
sage: R.seeds()
[1]
```

**successors**

**to_digraph** *(max_depth=None, loops=True, multiedges=True)*
Return the directed graph of the recursively enumerated set.

INPUT:

• max_depth – (default: self._max_depth) specifies the maximal depth for which outgoing edges of elements are computed
• loops – (default: True) option for the digraph
• multiedges – (default: True) option of the digraph

OUTPUT:
A directed graph

1.8. Recursively enumerated set
Warning: If the set is infinite, this will loop forever unless max_depth is finite.

EXAMPLES:

```
sage: child = \texttt{lambda} i: [(i+3) \% 10, (i+8) \% 10]

sage: R = RecursivelyEnumeratedSet([0], child)

sage: R.to_digraph()
Looped multi-digraph on 10 vertices
```

Digraph of an recursively enumerated set with a symmetric structure of infinite cardinality using max_depth argument:

```
sage: succ = \texttt{lambda} a: [(a[0]-1,a[1]), (a[0],a[1]-1), (a[0]+1,a[1]), (a[0], a[1]+1)]

sage: seeds = [(0,0)]

sage: C = RecursivelyEnumeratedSet(seeds, succ, structure='symmetric')

sage: C.to_digraph(max_depth=3)
Looped multi-digraph on 41 vertices
```

The max_depth argument can be given at the creation of the set:

```
sage: C = RecursivelyEnumeratedSet(seeds, succ, structure='symmetric', max_depth=2)

sage: C.to_digraph()
Looped multi-digraph on 25 vertices
```

Digraph of an recursively enumerated set with a graded structure:

```
sage: f = \texttt{lambda} a: [a[0]+1, a[1]]

sage: C = RecursivelyEnumeratedSet([0], f, structure='graded')

sage: C.to_digraph(max_depth=4)
Looped multi-digraph on 21 vertices
```

```
\textbf{class} \texttt{sage.sets.recursivelyEnumeratedSet.RecursivelyEnumeratedSet_graded}

\textbf{Bases}: \texttt{sage.sets.recursivelyEnumeratedSet.RecursivelyEnumeratedSet_generic}

\textbf{Generic tool for constructing ideals of a graded relation.}

\textbf{INPUT}:

- \texttt{seeds} – list (or iterable) of hashable objects
- \texttt{successors} – function (or callable) returning a list (or iterable)
- \texttt{enumeration} – 'depth', 'breadth' or None (default: None)
- \texttt{max_depth} – integer (default: float("inf"))

\textbf{EXAMPLES}:

```
sage: f = \texttt{lambda} a: [(a[0]+1,a[1]), (a[0],a[1]+1)]

sage: C = RecursivelyEnumeratedSet([0], f, structure='graded', max_depth=3)

sage: C
A recursively enumerated set with a graded structure (breadth first search) with max_depth=3

sage: list(C)
[(0, 0),
 (1, 0), (0, 1),
```

(continues on next page)
breadth_first_search_iterator \((\text{max\_depth}=\text{None})\)

Iterate on the elements of \(\text{self}\) (breadth first).

This iterator makes use of the graded structure by remembering only the elements of the current depth.

The elements are guaranteed to be enumerated in the order in which they are first visited (left-to-right traversal).

INPUT:

• \(\text{max\_depth} \) – (default: \(\text{self\_max\_depth}\)) specifies the maximal depth to which elements are computed

EXAMPLES:

```python
sage: f = lambda a: [(a[0]+1,a[1]), (a[0],a[1]+1)]
sage: C = RecursivelyEnumeratedSet([0], f, structure='graded')
sage: list(C.breadth_first_search_iterator(max_depth=3))
[(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), (3, 0), (2, 1), (1, 2), (0, 3)]
```

graded_component \((\text{depth})\)

Return the graded component of given depth.

This method caches each lower graded component. See \texttt{graded\_component\_iterator()} to generate each graded component without caching the previous ones.

A graded component is a set of elements of the same depth where the depth of an element is its minimal distance to a root.

INPUT:

• \(\text{depth} \) – integer

OUTPUT:

A set.

EXAMPLES:

```python
sage: f = lambda a: [a+1, a+1j]
sage: C = RecursivelyEnumeratedSet([0], f, structure='graded')
sage: for i in range(5): sorted(C.graded_component(i))
[0]
[I, 1]
[2*I, I + 1, 2]
[3*I, 2*I + 1, I + 2, 3]
[4*I, 3*I + 1, 2*I + 2, I + 3, 4]
```

graded_component_iterator()

Iterate over the graded components of \(\text{self}\).

A graded component is a set of elements of the same depth.

The algorithm remembers only the current graded component generated since the structure is graded.

OUTPUT:
An iterator of sets.

EXAMPLES:

```python
sage: f = lambda a: [(a[0]+1,a[1]), (a[0],a[1]+1)]
sage: C = RecursivelyEnumeratedSet([(0,0)], f, structure='graded', max_depth=3)
sage: it = C.graded_component_iterator()
sage: for _ in range(4): sorted(next(it))
[(0, 0)]
[(0, 1), (1, 0)]
[(0, 2), (1, 1), (2, 0)]
[(0, 3), (1, 2), (2, 1), (3, 0)]
```

class `sage.sets.recursivelyEnumeratedSet.RecursivelyEnumeratedSet_symmetric`

Bases: `sage.sets.recursivelyEnumeratedSet.RecursivelyEnumeratedSet_generic`

Generic tool for constructing ideals of a symmetric relation.

INPUT:

- `seeds` – list (or iterable) of hashable objects
- `successors` – function (or callable) returning a list (or iterable)
- `enumeration` – 'depth', 'breadth' or None (default: None)
- `max_depth` – integer (default: float("inf"))

EXAMPLES:

```python
sage: f = lambda a: [a-1,a+1]
sage: C = RecursivelyEnumeratedSet([0], f, structure='symmetric')
sage: for _ in range(7): next(it)
[0, -1, 1, -2, 2, -3, 3]
```

`breadth_first_search_iterator(max_depth=None)`

Iterate on the elements of `self` (breadth first).

This iterator makes use of the graded structure by remembering only the last two graded components since the structure is symmetric.

The elements are guaranteed to be enumerated in the order in which they are first visited (left-to-right traversal).

INPUT:

- `max_depth` – (default: `self._max_depth`) specifies the maximal depth to which elements are computed

EXAMPLES:

```python
sage: f = lambda a: [(a[0]-1,a[1]), (a[0],a[1]-1), (a[0]+1,a[1]), (a[0],a[1]+1)]
sage: C = RecursivelyEnumeratedSet([(0,0)], f, structure='symmetric')
sage: s = list(C.breadth_first_search_iterator(max_depth=2)); s
[(0, 0), (-1, 0), (0, -1), (1, 0), (0, 1), (-2, 0), (-1, -1), (-1, 1), (0, 2), (1, -1), (2, 0), (1, 1), (0, 2)]
```
This iterator is used by default for symmetric structure:

```
sage: it = iter(C)
sage: s == [next(it) for _ in range(13)]
True
```

**graded_component**(depth)

Return the graded component of given depth.

This method caches each lower graded component. See **graded_component_iterator()** to generate each graded component without caching the previous ones.

A graded component is a set of elements of the same depth where the depth of an element is its minimal distance to a root.

**INPUT:**

- depth – integer

**OUTPUT:**

A set.

**EXAMPLES:**

```
sage: f = lambda a: [a-1, a+1]
sage: C = RecursivelyEnumeratedSet([10, 15], f, structure='symmetric')
sage: for i in range(5): sorted(C.graded_component(i))
[[10, 15], [9, 11, 14, 16], [8, 12, 13, 17], [7, 18], [6, 19]]
```

**graded_component_iterator()**

Iterate over the graded components of self.

A graded component is a set of elements of the same depth.

The enumeration remembers only the last two graded components generated since the structure is symmetric.

**OUTPUT:**

An iterator of sets.

**EXAMPLES:**

```
sage: f = lambda a: [a-1, a+1]
sage: S = RecursivelyEnumeratedSet([10], f, structure='symmetric')
sage: it = S.graded_component_iterator()
sage: [sorted(next(it)) for _ in range(5)]
[[10], [9, 11], [8, 12], [7, 13], [6, 14]]
```

Starting with two generators:

```
sage: f = lambda a: [a-1, a+1]
sage: S = RecursivelyEnumeratedSet([5, 10], f, structure='symmetric')
sage: it = S.graded_component_iterator()
sage: [sorted(next(it)) for _ in range(5)]
[[5, 10], [4, 6, 9, 11], [3, 7, 8, 12], [2, 13], [1, 14]]
```

Gaussian integers:
```python
sage: f = lambda a: [a+1, a*I]
sage: S = RecursivelyEnumeratedSet([0], f, structure='symmetric')
sage: it = S.graded_component_iterator()
sage: [sorted(next(it)) for _ in range(7)]
[[0],
 [I, 1],
 [2*I, 1 + I, 2],
 [3*I, 2*I + 1, I + 2, 3],
 [4*I, 3*I + 1, 2*I + 2, I + 3, 4],
 [5*I, 4*I + 1, 3*I + 2, 2*I + 3, 1 + 4, 5],
 [6*I, 5*I + 1, 4*I + 2, 3*I + 3, 2*I + 4, I + 5, 6]]
```

`sage.sets.recursivelyEnumeratedSet.search_forest_iterator` *(roots, children, algorithm='depth')*

Return an iterator on the nodes of the forest having the given roots, and where `children(x)` returns the children of the node `x` of the forest. Note that every node of the tree is returned, not simply the leaves.

**INPUT:**

- `roots` – a list (or iterable)
- `children` – a function returning a list (or iterable)
- `algorithm` – 'depth' or 'breadth' (default: 'depth')

**EXAMPLES:**

We construct the prefix tree of binary sequences of length at most three, and enumerate its nodes:

```python
sage: from sage.sets.recursivelyEnumeratedSet import search_forest_iterator
sage: list(search_forest_iterator([[]], lambda l: [l+[0], l+[1]].
.....: if len(l) < 3 else []))
[[], [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0],
 [0, 1, 1], [1], [1, 0], [1, 0, 0], [1, 0, 1], [1, 1], [1, 1, 0], [1, 1, 1]]
```

By default, the nodes are iterated through by depth first search. We can instead use a breadth first search (increasing depth):

```python
sage: list(search_forest_iterator([[]], lambda l: [l+[0], l+[1]].
.....: if len(l) < 3 else [],
.....: algorithm='breadth'))
[[],
 [0], [1],
 [0, 0], [0, 1], [1, 0], [1, 1],
 [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
 [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
```

This allows for iterating through trees of infinite depth:

```python
sage: it = search_forest_iterator([[]], lambda l: [l+[0], l+[1]], algorithm='breadth')
sage: [next(it) for i in range(16)]
[[],
 [0], [1],
 [0, 0], [0, 1], [1, 0], [1, 1],
 [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
 [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1],
 [0, 0, 0], [0, 0, 1]]
```

Here is an iterator through the prefix tree of sequences of letters in 0, 1, 2 without repetitions, sorted by length; the leaves are therefore permutations:

```python
sage: it = search_forest_iterator([[]], lambda l: [l+[0], l+[1]], algorithm='breadth')
sage: [next(it) for i in range(16)]
[[],
 [0], [1],
 [0, 0], [0, 1], [1, 0], [1, 1],
 [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
 [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1],
 [0, 0, 0], [0, 0, 1]]
```
1.9 Maps between finite sets

This module implements parents modeling the set of all maps between two finite sets. At the user level, any such parent should be constructed using the factory class `FiniteSetMaps` which properly selects which of its subclasses to use.

AUTHORS:
• Florent Hivert

```python
sage: list(search_forest_iterator([], lambda l: [l + [i] for i in range(3) if i not in l], algorithm='breadth'))
[[], [0], [1], [2], [0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1], [0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]
```

```python
class sage.sets.finite_set_maps.FiniteSetEndoMaps_N(n, action, category=None)
```

Bases: `sage.sets.finite_set_maps.FiniteSetMaps_MN`

The sets of all maps from \{1, 2, \ldots, n\} to itself

Users should use the factory class `FiniteSetMaps` to create instances of this class.

INPUT:

• `n` – an integer.
• `category` – the category in which the sets of maps is constructed. It must be a sub-category of `Monoids().Finite()` and `EnumeratedSets().Finite()` which is the default value.

```python
Element
alias of sage.sets.finite_set_map_cy.FiniteSetEndoMap_N

an_element()
Returns a map in self

EXAMPLES:
```
```python
sage: M = FiniteSetMaps(4)
sage: M.an_element()
[3, 2, 1, 0]
```

```python
one()

EXAMPLES:
```
```python
sage: M = FiniteSetMaps(4)
sage: M.one()
[0, 1, 2, 3]
```

class sage.sets.finite_set_maps.FiniteSetEndoMaps_Set(domain, action, category=None)
```

Bases: `sage.sets.finite_set_maps.FiniteSetMaps_Set`, `sage.sets.finite_set_maps.FiniteSetEndoMaps_N`

The sets of all maps from a set to itself

Users should use the factory class `FiniteSetMaps` to create instances of this class.
INPUT:

- domain – an object in the category `FiniteSets()`.
- category – the category in which the sets of maps is constructed. It must be a sub-category of `Monoids().Finite()` and `EnumeratedSets().Finite()` which is the default value.

Element

alias of `sage.sets.finite_set_map_cy.FiniteSetEndoMap_Set`

class `sage.sets.finite_set_maps.FiniteSetMaps`

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent.Parent`

Maps between finite sets

Constructs the set of all maps between two sets. The sets can be given using any of the three following ways:

1. an object in the category `Sets()`.
2. a finite iterable. In this case, an object of the class `FiniteEnumeratedSet` is constructed from the iterable.
3. an integer \( n \) designing the set \( \{0, 1, \ldots, n-1\} \). In this case an object of the class `IntegerRange` is constructed.

INPUT:

- domain – a set, finite iterable, or integer.
- codomain – a set, finite iterable, integer, or `None` (default). In this last case, the maps are endo-maps of the domain.
- action – "left" (default) or "right". The side where the maps act on the domain. This is used in particular to define the meaning of the product (composition) of two maps.
- category – the category in which the sets of maps is constructed. By default, this is `FiniteMonoids()` if the domain and codomain coincide, and `FiniteEnumeratedSets()` otherwise.

OUTPUT:

an instance of a subclass of `FiniteSetMaps` modeling the set of all maps between `domain` and `codomain`.

EXAMPLES:

We construct the set \( M \) of all maps from \{a, b\} to \{3, 4, 5\}:

```python
sage: M = FiniteSetMaps(['a', 'b'], [3, 4, 5]); M
Maps from {'a', 'b'} to {3, 4, 5}
sage: M.cardinality()
9
sage: M.domain()
{'a', 'b'}
sage: M.codomain()
{3, 4, 5}
sage: for f in M: print(f)
map: a -> 3, b -> 3
map: a -> 3, b -> 4
map: a -> 3, b -> 5
map: a -> 4, b -> 3
map: a -> 4, b -> 4
map: a -> 4, b -> 5
```
Elements can be constructed from functions and dictionaries:

```python
sage: M(lambda c: ord(c)-94)
map: a -> 3, b -> 4
sage: M.from_dict({'a':3, 'b':5})
map: a -> 3, b -> 5
```

If the domain is equal to the codomain, then maps can be composed:

```python
sage: M = FiniteSetMaps([1, 2, 3])
sage: f = M.from_dict({1:2, 2:1, 3:3}); f
map: 1 -> 2, 2 -> 1, 3 -> 3
sage: g = M.from_dict({1:2, 2:3, 3:1}); g
map: 1 -> 2, 2 -> 3, 3 -> 1
sage: f * g
map: 1 -> 1, 2 -> 3, 3 -> 2
```

This makes $M$ into a monoid:

```python
sage: M.category()
Category of finite enumerated monoids
sage: M.one()
map: 1 -> 1, 2 -> 2, 3 -> 3
```

By default, composition is from right to left, which corresponds to an action on the left. If one specifies `action` to right, then the composition is from left to right:

```python
sage: M = FiniteSetMaps([1, 2, 3], action = 'right')
sage: f = M.from_dict({1:2, 2:1, 3:3})
sage: g = M.from_dict({1:2, 2:3, 3:1})
sage: f * g
map: 1 -> 3, 2 -> 2, 3 -> 1
```

If the domains and codomains are both of the form \{0, \ldots\}, then one can use the shortcut:

```python
sage: M = FiniteSetMaps(2,3); M
Maps from \{0, 1\} to \{0, 1, 2\}
sage: M.cardinality()
9
```

For a compact notation, the elements are then printed as lists $[f(i), i = 0, \ldots]$:

```python
sage: list(M)
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
```

**cardinality**

- The cardinality of `self`

**EXAMPLES:**
class sage.sets.finite_set_maps.FiniteSetMaps_MN(m, n, category=None)
    Bases: sage.sets.finite_set_maps.FiniteSetMaps
    The set of all maps from \{1, 2, \ldots, m\} to \{1, 2, \ldots, n\}.
    Users should use the factory class FiniteSetMaps to create instances of this class.
    INPUT:
    • m, n – integers
    • category – the category in which the sets of maps is constructed. It must be a sub-category of
      EnumeratedSets().Finite() which is the default value.

Element
    alias of sage.sets.finite_set_map_cy.FiniteSetMap_MN

an_element()
    Returns a map in self
    EXAMPLES:

    sage: M = FiniteSetMaps(4, 2)
    sage: M.an_element()
    [0, 0, 0, 0]
    sage: M = FiniteSetMaps(0, 0)
    sage: M.an_element()
    []

An exception EmptySetError is raised if this set is empty, that is if the codomain is empty and the
domain is not.

    sage: M = FiniteSetMaps(4, 0) sage: M.cardinality() 0 sage: M.an_element() Traceback (most recent call last): ... EmptySetError
codomain()
    The codomain of self
    EXAMPLES:

    sage: FiniteSetMaps(3,2).codomain()
    {0, 1}
domain()
    The domain of self
    EXAMPLES:

    sage: FiniteSetMaps(3,2).domain()
    {0, 1, 2}

class sage.sets.finite_set_maps.FiniteSetMaps_Set(domain, codomain, category=None)
    Bases: sage.sets.finite_set_maps.FiniteSetMaps_MN
    The sets of all maps between two sets
    Users should use the factory class FiniteSetMaps to create instances of this class.
INPUT:

- domain – an object in the category `FiniteSets()`.
- codomain – an object in the category `FiniteSets()`.
- category – the category in which the sets of maps is constructed. It must be a sub-category of `EnumeratedSets().Finite()` which is the default value.

**Element**

alias of `sage.sets.finite_set_map_cy.FiniteSetMap_Set`  

codomain()

The codomain of `self`

EXAMPLES:

```python
sage: FiniteSetMaps(['a', 'b'], [3, 4, 5]).codomain()
{3, 4, 5}
```

domain()

The domain of `self`

EXAMPLES:

```python
sage: FiniteSetMaps(['a', 'b'], [3, 4, 5]).domain()
{'a', 'b'}
```

from_dict(d)

Create a map from a dictionary

EXAMPLES:

```python
sage: M = FiniteSetMaps(['a', 'b'], [3, 4, 5])
sage: M.from_dict({'a':3, 'b':4})
map: a -> 3, b -> 4
```

### 1.10 Data structures for maps between finite sets

This module implements several fast Cython data structures for maps between two finite set. Those classes are not intended to be used directly. Instead, such a map should be constructed via its parent, using the class `FiniteSetMaps`.

EXAMPLES:

To create a map between two sets, one first creates the set of such maps:

```python
sage: M = FiniteSetMaps(['a', 'b'], [3, 4, 5])
```

The map can then be constructed either from a function:

```python
sage: f1 = M(lambda c: ord(c)-94); f1
map: a -> 3, b -> 4
```

or from a dictionary:

```python
sage: f2 = M.from_dict({'a':3, 'b':4}); f2
map: a -> 3, b -> 4
```

The two created maps are equal:
Internally, maps are represented as the list of the ranks of the images $f(x)$ in the co-domain, in the order of the domain:

```
sage: f2 = dict([(0, 3), (1, 4)])
```

A third fast way to create a map it to use such a list. It should be kept for internal use:

```
sage: f3 = M._from_list_([0, 1]); f3
map: a -> 3, b -> 4
```

AUTHORS:

- Florent Hivert

```python
class sage.sets.finite_set_map_cy.FiniteSetEndoMap_N
    Bases: sage.sets.finite_set_map_cy.FiniteSetMap_MN

    Maps from $\text{range}(n)$ to itself.

    See also:
    FiniteSetMap_MN for assumptions on the parent

class sage.sets.finite_set_map_cy.FiniteSetEndoMap_Set
    Bases: sage.sets.finite_set_map_cy.FiniteSetMap_Set

    Maps from a set to itself

    See also:
    FiniteSetMap_Set for assumptions on the parent

class sage.sets.finite_set_map_cy.FiniteSetMap_MN
    Bases: sage.structure.list_clone.ClonableIntArray

    Data structure for maps from $\text{range}(m)$ to $\text{range}(n)$.

    We assume that the parent given as argument is such that:
    - $m$ is stored in self.parent()._m
    - $n$ is stored in self.parent()._n
    - the domain is in self.parent().domain()
    - the codomain is in self.parent().codomain()

    check()
        Performs checks on self
        Check that self is a proper function and then calls parent.check_element(self) where parent is the parent of self.

codomain()
    Returns the codomain of self
```
finite_set_maps = FiniteSetMaps(4, 3)([1, 0, 2, 1]).codomain()
{0, 1, 2}

**domain()**
Returns the domain of self

**EXAMPLES:**
finite_set_maps = FiniteSetMaps(4, 3)([1, 0, 2, 1]).domain()
{0, 1, 2, 3}

**fibers()**
Returns the fibers of self

**OUTPUT:**

a dictionary d such that d[y] is the set of all x in domain such that f(x) = y

**EXAMPLES:**
finite_set_maps = FiniteSetMaps(4, 3)([1, 0, 2, 1]).fibers()
{0: {1}, 1: {0, 3}, 2: {2}}
f = FiniteSetMaps(['a', 'b', 'c'])
f.from_dict({'a': 'b', 'b': 'a', 'c': 'b'}).fibers() == {'a': {'b'}, 'b' →: {'a', 'c'}}
True

**getimage(i)**
Returns the image of i by self

**INPUT:**

* i – any object.

**Note:** if you need speed, please use instead _getimage()

**EXAMPLES:**
finite_set_maps = FiniteSetMaps(4, 3)([1, 0, 2, 1])
finite_set_maps.getimage(0), finite_set_maps.getimage(1), finite_set_maps.getimage(2), finite_set_maps.getimage(3)
(1, 0, 2, 1)

**image_set()**
Returns the image set of self

**EXAMPLES:**
finite_set_maps = FiniteSetMaps(4, 3)([1, 0, 2, 1])
f = finite_set_maps.getimage(0), f.getimage(1), f.getimage(2), f.getimage(3)
{1, 0, 2, 1}

**items()**
The items of self
Return the list of the ordered pairs (x, self(x))

**EXAMPLES:**

```python
sage: FiniteSetMaps(4, 3)([1, 0, 2, 1]).items()
[(0, 1), (1, 0), (2, 2), (3, 1)]
```

### setimage(i, j)
Set the image of `i` as `j` in `self`

**Warning:** `self` must be mutable; otherwise an exception is raised.

**INPUT:**
- `i, j` – two object's

**OUTPUT:** None

**Note:** if you need speed, please use instead `_setimage()`

**EXAMPLES:**
```
sage: fs = FiniteSetMaps(4, 3)([1, 0, 2, 1])
sage: fs2 = copy(fs)
sage: fs2.setimage(2, 1)
sage: fs2
[1, 0, 1, 1]
sage: with fs.clone() as fs3: .....: fs3.setimage(0, 2) .....: fs3.setimage(1, 2) sage: fs3 [2, 2, 2, 1]
```

class **sage.sets.finite_set_map_cy.FiniteSetMap_Set**
Bases: **sage.sets.finite_set_map_cy.FiniteSetMap_MN**

Data structure for maps

We assume that the parent given as argument is such that:
- the domain is in `parent.domain()`
- the codomain is in `parent.codomain()`
- `parent._m` contains the cardinality of the domain
- `parent._n` contains the cardinality of the codomain
- `parent._unrank_domain` and `parent._rank_domain` is a pair of reciprocal rank and unrank functions between the domain and `range(parent._m)`.
- `parent._unrank_codomain` and `parent._rank_codomain` is a pair of reciprocal rank and unrank functions between the codomain and `range(parent._n)`.

**classmethod from_dict**(t, parent, d)

Creates a `FiniteSetMap` from a dictionary

**Warning:** no check is performed!
classmethod from_list \((t, \text{parent}, \text{lst})\)

Creates a FiniteSetMap from a list

**Warning:** no check is performed!

getimage \((i)\)

Returns the image of \(i\) by self

**INPUT:**
- \(i\) – an int

**EXAMPLES:**

```sage
F = FiniteSetMaps(["a", "b", "c", "d"], ["u", "v", "w"])
s = F._from_list_([1, 0, 2, 1])
list(map(s.getimage, ["a", "b", "c", "d"]))
['v', 'u', 'w', 'v']
```

image_set()

Returns the image set of self

**EXAMPLES:**

```sage
F = FiniteSetMaps(["a", "b", "c"])
sorted(F.from_dict({"a": "b", "b": "a", "c": "b"}).image_set())
['a', 'b']
sage: F = FiniteSetMaps(["a", "b", "c"])
sage: F(lambda x: "c").image_set()
{'c'}
```

items()

The items of self

Return the list of the couple \((x, \text{self}(x))\)

**EXAMPLES:**

```sage
F = FiniteSetMaps(["a", "b", "c"])
sage: F.from_dict({"a": "b", "b": "a", "c": "b"}).items()
[('a', 'b'), ('b', 'a'), ('c', 'b')]`

setimage \((i, j)\)

Set the image of \(i\) as \(j\) in self

**Warning:** self must be mutable otherwise an exception is raised.

**INPUT:**
- \(i, j\) – two object’s

**OUTPUT:** None

**EXAMPLES:**

```sage
F = FiniteSetMaps(["a", "b", "c"])
sage: F.from_dict({"a": "b", "b": "a", "c": "b"}).items()
[('a', 'b'), ('b', 'a'), ('c', 'b')]```

1.10. Data structures for maps between finite sets
sage: F = FiniteSetMaps(["a", "b", "c", "d"], ["u", "v", "w"])

sage: fs = F(lambda x: "v")

sage: fs2 = copy(fs)

sage: fs2.setimage("a", "w")

sage: fs2
map: a \rightarrow w, b \rightarrow v, c \rightarrow v, d \rightarrow v

sage: with fs.clone() as fs3:
....:     fs3.setimage("a", "u")
....:     fs3.setimage("c", "w")

sage: fs3
map: a \rightarrow u, b \rightarrow v, c \rightarrow w, d \rightarrow v

\begin{verbatim}

sage.sets.finite_set_map_cy.FiniteSetMap_Set_from_dict (t, parent, d)
Creates a FiniteSetMap from a dictionary

\textbf{Warning:} no check is performed!

sage.sets.finite_set_map_cy.FiniteSetMap_Set_from_list (t, parent, lst)
Creates a FiniteSetMap from a list

\textbf{Warning:} no check is performed!

sage.sets.finite_set_map_cy.fibers (f, domain)
Returns the fibers of the function \( f \) on the finite set \( \text{domain} \)

\textbf{INPUT:}

- \( f \) – a function or callable
- \( \text{domain} \) – a finite iterable

\textbf{OUTPUT:}

- a dictionary \( d \) such that \( d[y] \) is the set of all \( x \) in \( \text{domain} \) such that \( f(x) = y \)

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.sets.finite_set_map_cy import fibers, fibers_args
sage: fibers (lambda x: 1, [])
[{}]
sage: fibers (lambda x: x^2, [-1, 2, -3, 1, 3, 4])
{1: [1, -1], 3: [3, -3], 4: [4, -4]}
sage: fibers (lambda x: 1, [-1, 2, -3, 1, 3, 4])
{1: [1, -1, 2, 3, 4, -3, -1]}
sage: fibers (lambda x: 1, [1, 1, 1])
{1: [1]}
\end{verbatim}

See also:

\texttt{fibers_args()} if one needs to pass extra arguments to \textit{f}.

sage.sets.finite_set_map_cy.fibers_args (f, domain, *args, **opts)
Returns the fibers of the function \( f \) on the finite set \( \text{domain} \)

It is the same as \texttt{fibers()} except that one can pass extra argument for \( f \) (with a small overhead)

\textbf{EXAMPLES:}
1.11 Totally Ordered Finite Sets

AUTHORS:

• Stepan Starosta (2012): Initial version

class sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet(elements, facade=True)

    Bases: sage.sets.finite_enumerated_set.FiniteEnumeratedSet

Totally ordered finite set.

This is a finite enumerated set assuming that the elements are ordered based upon their rank (i.e. their position in the set).

INPUT:

• elements – A list of elements in the set

• facade – (default: True) if True, a facade is used; it should be set to False if the elements do not inherit from Element or if you want a funny order. See examples for more details.

See also:

FiniteEnumeratedSet

EXAMPLES:

    sage: S = TotallyOrderedFiniteSet([1,2,3])
sage: S
{1, 2, 3}
sage: S.cardinality()
3

By default, totally ordered finite set behaves as a facade:

    sage: S(1).parent()
Integer Ring

It makes comparison fails when it is not the standard order:

    sage: T1 = TotallyOrderedFiniteSet([3,2,5,1])
sage: T1(3) < T1(1)
False
    sage: T2 = TotallyOrderedFiniteSet([3,var('x')])
sage: T2(3) < T2(var('x'))
3 < x

To make the above example work, you should set the argument facade to False in the constructor. In that case, the elements of the set have a dedicated class:
```python
sage: A = TotallyOrderedFiniteSet([3, 2, 0, 'a', 7, (0, 0), 1], facade=False)
sage: A
{3, 2, 0, 'a', 7, (0, 0), 1}
sage: x = A.an_element()
sage: x
3
sage: x.parent()
{3, 2, 0, 'a', 7, (0, 0), 1}
sage: A(3) < A(2)
True
sage: A('a') < A(7)
True
sage: A(3) > A(2)
False
sage: A(1) < A(3)
False
sage: A(3) == A(3)
True

But then, the equality comparison is always False with elements outside of the set:

```python
sage: A(1) == 1
False
sage: 1 == A(1)
False
sage: 'a' == A('a')
False
sage: A('a') == 'a'
False
```

Since trac ticket #16280, totally ordered sets support elements that do not inherit from `sage.structure.element.Element`, whether they are facade or not:

```python
sage: S = TotallyOrderedFiniteSet(['a', 'b'])
sage: S('a')
'a'
sage: S = TotallyOrderedFiniteSet(['a', 'b'], facade=False)
sage: S('a')
'a'
```

Multiple elements are automatically deleted:

```python
sage: TotallyOrderedFiniteSet([1, 1, 2, 1, 2, 2, 5, 4])
{1, 2, 5, 4}
```

### Element

alias of `TotallyOrderedFiniteSetElement`

#### le `(x, y)`

Return True if $x \leq y$ for the order of self.

**EXAMPLES:**

```python
sage: T = TotallyOrderedFiniteSet([1, 3, 2], facade=False)
sage: T1, T3, T2 = T.list()
sage: T.le(T1, T3)
True
```
class sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSetElement (parent, data)

Element of a finite totally ordered set.

EXAMPLES:

```python
sage: S = TotallyOrderedFiniteSet([2,7], facade=False)
sage: x = S(2)
sage: print(x)
2
sage: x.parent()
{2, 7}
```

### 1.12 Set of all objects of a given Python class

`sage.sets.pythonclass.Set_PythonType(typ)`

Return the (unique) Parent that represents the set of Python objects of a specified type.

EXAMPLES:

```python
sage: from sage.sets.pythonclass import Set_PythonType
sage: S = Set_PythonType(list)
Set of Python objects of class 'list'
sage: Set_PythonType(list) is Set_PythonType(list)
True
sage: S = Set_PythonType(tuple)
S((1, 2, 3))
```

S is a parent which models the set of all lists:

```python
sage: S.category()
Category of sets
```

class sage.sets.pythonclass.Set_PythonType_class

The set of Python objects of a given class.

The elements of this set are not instances of Element; they are instances of the given class.

INPUT:

- typ – a Python (new-style) class

EXAMPLES:

```python
sage: from sage.sets.pythonclass import Set_PythonType
sage: S = Set_PythonType(int); S
Set of Python objects of class 'int'
sage: int('1') in S
True
```
sage: Integer('1') in S
False

sage: Set_PythonType(2)
Traceback (most recent call last):
...
TypeError: must be initialized with a class, not 2

cardinality()
EXAMPLES:

sage: from sage.sets.pythonclass import Set_PythonType
sage: S = Set_PythonType(bool)
sage: S.cardinality()
2
sage: S = Set_PythonType(int)
sage: S.cardinality()
+Infinity

object()
EXAMPLES:

sage: from sage.sets.pythonclass import Set_PythonType
sage: S = Set_PythonType(tuple)
<... 'tuple'>
SETS OF NUMBERS

2.1 Integer Range

AUTHORS:

- Florent Hivert (2010-03): Added a class factory + cardinality method.
- Vincent Delecroix (2012-02): add methods rank/unrank, make it compliant with Python int.

class sage.sets.integer_range.IntegerRange

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The class of Integer ranges

Returns an enumerated set containing an arithmetic progression of integers.

INPUT:

- begin – an integer, Infinity or -Infinity
- end – an integer, Infinity or -Infinity
- step – a non zero integer (default to 1)
- middle_point – an integer inside the set (default to None)

OUTPUT:

A parent in the category FiniteEnumeratedSets() or InfiniteEnumeratedSets() depending on the arguments defining self.

IntegerRange(i, j) returns the set of \{i, i+1, i+2, ..., j-1\}. start () defaults to 0. When step is given, it specifies the increment. The default increment is 1. IntegerRange allows begin and end to be infinite.

IntegerRange is designed to have similar interface Python range. However, whereas range accept and returns Python int, IntegerRange deals with Integer.

If middle_point is given, then the elements are generated starting from it, in a alternating way: \{m, m+1, m-2, m+2, m-2...\}.

EXAMPLES:

```python
sage: list(IntegerRange(5))
[0, 1, 2, 3, 4]
sage: list(IntegerRange(2,5))
```

(continues on next page)
When `begin` and `end` are both finite, `IntegerRange(begin, end, step)` is the set whose list of elements is equivalent to the python construction `range(begin, end, step):

```python
sage: list(IntegerRange(4,105,3)) == list(range(4,105,3))
True
sage: list(IntegerRange(-54,13,12)) == list(range(-54,13,12))
True
```

Except for the type of the numbers:

```python
sage: type(IntegerRange(-54,13,12)[0]), type(list(range(-54,13,12))[0])
(<... 'sage.rings.integer.Integer'>, <... 'int'>)
```

When `begin` is finite and `end` is `+Infinity`, `self` is the infinite arithmetic progression starting from the `begin` by step `step`:

```python
sage: I = IntegerRange(54,Infinity,3); I
{54, 57, ...}
```

```python
sage: -12 in I
True
sage: -15 in I
False
```

When `begin` and `end` are both infinite, you will have to specify the extra argument `middle_point`. `self` is then defined by a point and a progression/regression setting by `step`. The enumeration is done this way: (let us call $m$ the `middle_point`)\{m, m + step, m − step, m + 2step, m − 2step, m + 3step, ...\}:

```python
sage: I = IntegerRange(-Infinity,Infinity,37,-12); I
Integer progression containing -12 with increment 37 and bounded with -Infinity and +Infinity
```

```python
sage: -12 in I
True
sage: -15 in I
False
```
It is also possible to use the argument `middle_point` for other cases, finite or infinite. The set will be the same as if you didn’t give this extra argument but the enumeration will begin with this `middle_point`:

```python
sage: I = IntegerRange(123, -12, -14); I
(123, 109, ..., -3)
sage: list(I)
[123, 109, 95, 81, 67, 53, 39, 25, 11, -3]
sage: J = IntegerRange(123, -12, -14, 25); J
Integer progression containing 25 with increment -14 and bounded with 123 and -12
sage: list(J)
[25, 11, 39, -3, 53, 67, 81, 95, 109, 123]
```

Remember that, like for range, if you define a non empty set, `begin` is supposed to be included and `end` is supposed to be excluded. In the same way, when you define a set with a `middle_point`, the `begin` bound will be supposed to be included and the `end` bound supposed to be excluded:

```python
sage: I = IntegerRange(-100, 100, 10, 0)
sage: J = list(range(-100, 100, 10))
sage: 100 in I
False
sage: 100 in J
False
sage: -100 in I
True
sage: -100 in J
True
sage: list(I)
[0, 10, -10, 20, -20, 30, -30, 40, -40, 50, -50, 60, -60, 70, -70, 80, -80, 90, -90, -100]
```

**Note:** The input is normalized so that:

```python
sage: IntegerRange(1, 6, 2) is IntegerRange(1, 7, 2)
True
sage: IntegerRange(1, 8, 3) is IntegerRange(1, 10, 3)
True
```

**element_class**

alias of `sage.rings.integer.Integer`

class `sage.sets.integer_range.IntegerRangeEmpty (elements)`

   Bases: `sage.sets.integer_range.IntegerRange, sage.sets.finite_enumerated_set.FiniteEnumeratedSet`

A singleton class for empty integer ranges

See `IntegerRange` for more details.

class `sage.sets.integer_range.IntegerRangeFinite (begin, end, step=1)`

   Bases: `sage.sets.integer_range.IntegerRange`

The class of finite enumerated sets of integers defined by finite arithmetic progressions

See `IntegerRange` for more details.
cardinality()
Return the cardinality of self

EXAMPLES:

```
sage: IntegerRange(123, 12, -4).cardinality()
28
sage: IntegerRange(-57, 12, 8).cardinality()
9
sage: IntegerRange(123, 12, 4).cardinality()
0
```

rank(x)
EXAMPLES:

```
sage: I = IntegerRange(-57, 36, 8)
sage: I.rank(23)
10
sage: I.unrank(10)
23
sage: I.rank(22)
Traceback (most recent call last):
  ... IndexError: 22 not in self
sage: I.rank(87)
Traceback (most recent call last):
  ... IndexError: 87 not in self
```

unrank(i)
Return the i-th element of this integer range.

EXAMPLES:

```
sage: I = IntegerRange(1, 13, 5)
sage: I[0], I[1], I[2]
(1, 6, 11)
sage: I[3]
Traceback (most recent call last):
  ... IndexError: out of range
sage: I[-1]
11
sage: I[-4]
Traceback (most recent call last):
  ... IndexError: out of range
sage: I = IntegerRange(13, 1, -1)
sage: l = I.list()
sage: [I[i] for i in range(I.cardinality())] == l
True
sage: l.reverse()
sage: [I[i] for i in range(-1, -I.cardinality()-1, -1)] == l
True
```

class sage.sets.integer_range.IntegerRangeFromMiddle(begin, end, step=1, middle_point=1)

Bases: sage.sets.integer_range.IntegerRange
The class of finite or infinite enumerated sets defined with an inside point, a progression and two limits.

See `IntegerRange` for more details.

```
next (elt)
```
Return the next element of `elt` in `self`.

**EXAMPLES:**

```
sage: from sage.sets.integer_range import IntegerRangeFromMiddle
sage: I = IntegerRangeFromMiddle(-100,100,10,0)
sage: (I.next(0), I.next(10), I.next(-10), I.next(20), I.next(-100))
(10, -10, 20, -20, None)
sage: I = IntegerRangeFromMiddle(-Infinity,Infinity,10,0)
sage: (I.next(0), I.next(10), I.next(-10), I.next(20), I.next(-100))
(10, -10, 20, -20, 110)
sage: I.next(1)
Traceback (most recent call last):
  ...  
LookupError: 1 not in Integer progression containing 0 with increment 10 and → bounded with -Infinity and +Infinity
```

```
class sage.sets.integer_range.IntegerRangeInfinite (begin, step=1)
Bases: sage.sets.integer_range.IntegerRange
```

The class of infinite enumerated sets of integers defined by infinite arithmetic progressions.

See `IntegerRange` for more details.

```
rank (x)
```
EXAMPLES:

```
sage: I = IntegerRange(-57,Infinity,8)
sage: I.rank(23)
10
sage: I.unrank(10)
23
sage: I.rank(22)
Traceback (most recent call last):
  ...  
IndexError: 22 not in self
```

```
unrank (i)
```
Returns the i-th element of `self`.

**EXAMPLES:**

```
sage: I = IntegerRange(-8,Infinity,3)
sage: I.unrank(1)
-5
```
2.2 Positive Integers

```python
class sage.sets.positive_integers.PositiveIntegers
    Bases: sage.sets.integer_range.IntegerRangeInfinite

The enumerated set of positive integers. To fix the ideas, we mean \{1, 2, 3, 4, 5, \ldots \}.
This class implements the set of positive integers, as an enumerated set (see InfiniteEnumeratedSets).
This set is an integer range set. The construction is therefore done by IntegerRange (see IntegerRange).

EXAMPLES:

```sage
definitions:
sage: PP = PositiveIntegers()
sage: PP
Positive integers
sage: PP.cardinality()
+Infinity
sage: TestSuite(PP).run()
sage: PP.list()
Traceback (most recent call last):
... NotImplementedError: cannot list an infinite set
sage: it = iter(PP)
sage: (next(it), next(it), next(it), next(it), next(it))
(1, 2, 3, 4, 5)
sage: PP.first()
1

an_element()

Returns an element of self.

EXAMPLES:

```sage
definitions:

```
Traceback (most recent call last):
...
NotImplementedError: cannot list an infinite set
sage: NN.element_class
<... 'sage.rings.integer.Integer'>
sage: it = iter(NN)
sage: [next(it), next(it), next(it), next(it), next(it)]
[0, 1, 2, 3, 4]
sage: NN.first()
0

Currently, this is just a “facade” parent; namely its elements are plain Sage Integers with Integer Ring as parent:

sage: x = NN(15); type(x)
<... 'sage.rings.integer.Integer'>
sage: x.parent()
Integer Ring
sage: x+3
18

In a later version, there will be an option to specify whether the elements should have Integer Ring or Non negative integers as parent:

sage: NN = NonNegativeIntegers(facade = False) # todo: not implemented
sage: x = NN(5) # todo: not implemented
sage: x.parent() # todo: not implemented

Non negative integers

This runs generic sanity checks on NN:

sage: TestSuite(NN).run()

TODO: do not use NN any more in the doctests for NonNegativeIntegers.

Element

alias of sage.rings.integer.Integer

an_element()

EXAMPLES:

sage: NonNegativeIntegers().an_element()
42

from_integer

alias of sage.rings.integer.Integer

next (o)

EXAMPLES:

sage: NN = NonNegativeIntegers()
sage: NN.next(3)
4

some_elements()

EXAMPLES:
sage: NonNegativeIntegers().some_elements()
[0, 1, 3, 42]

unrank(rnk)
EXAMPLES:

sage: NN = NonNegativeIntegers()
sage: NN.unrank(100)
100

2.4 The set of prime numbers

AUTHORS:
• William Stein (2005): original version
• Florent Hivert (2009-11): adapted to the category framework.

class sage.sets.primes.Primes(proof)
Bases: sage.structure.parent.Set_generic, sage.structure.unique_representation.UniqueRepresentation

The set of prime numbers.

EXAMPLES:

sage: P = Primes(); P
Set of all prime numbers: 2, 3, 5, 7, ...

We show various operations on the set of prime numbers:

sage: P.cardinality()
+Infinity
sage: R = Primes()
sage: P == R
True
sage: 5 in P
True
sage: 100 in P
False
sage: len(P)
Traceback (most recent call last):
... NotImplementedError: infinite set

first()
Return the first prime number.

EXAMPLES:

sage: P = Primes()
sage: P.first()
2

next(pr)
Return the next prime number.
EXAMPLES:

```sage
sage: P = Primes()
sage: P.next(5)
7
```

unrank \((n)\)

Return the \(n\)-th prime number.

EXAMPLES:

```sage
sage: P = Primes()
sage: P.unrank(0)
2
sage: P.unrank(5)
13
sage: P.unrank(42)
191
```

### 2.5 Subsets of the Real Line

This module contains subsets of the real line that can be constructed as the union of a finite set of open and closed intervals.

EXAMPLES:

```sage
sage: RealSet(0,1)
(0, 1)
sage: RealSet((0,1), [2,3])
(0, 1) + [2, 3]
sage: RealSet(-oo, oo)
(-oo, +oo)
```

Brackets must be balanced in Python, so the naive notation for half-open intervals does not work:

```sage
sage: RealSet([0,1))
Traceback (most recent call last):
...  
SyntaxError: ...
```

Instead, you can use the following construction functions:

```sage
sage: RealSet.open_closed(0,1)
(0, 1]
sage: RealSet.closed_open(0,1)
[0, 1)
sage: RealSet.point(1/2)
[1/2]
sage: RealSet.unbounded_below_open(0)
(-oo, 0)
sage: RealSet.unbounded_below_closed(0)
(-oo, 0]
sage: RealSet.unbounded_above_open(1)
(1, +oo)
sage: RealSet.unbounded_above_closed(1)
[1, +oo)
```
Relations containing symbols and numeric values or constants:

```
sage: RealSet(x != 0)
(-oo, 0) + (0, +oo)
sage: RealSet(x == pi)
{pi}
sage: RealSet(x < 1/2)
(-oo, 1/2)
sage: RealSet(1/2 < x)
(1/2, +oo)
sage: RealSet(1.5 <= x)
[1.50000000000000, +oo]
```

Note that multiple arguments are combined as union:

```
sage: RealSet(x >= 0, x < 1)
(-oo, +oo)
sage: RealSet(x >= 0, x > 1)
[0, +oo)
sage: RealSet(x >= 0, x > -1)
(-1, +oo)
```

AUTHORS:

- Laurent Claessens (2010-12-10): Interval and ContinuousSet, posted to sage-devel at http://www.mail-archive.com/sage-support@googlegroups.com/msg21326.html.
- Ares Ribo (2011-10-24): Extended the previous work defining the class RealSet.
- Jordi Saludes (2011-12-10): Documentation and file reorganization.
- Volker Braun (2013-06-22): Rewrite

```python
class sage.sets.real_set.InternalRealInterval(lower, lower_closed, upper, upper_closed, check=True)
    Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A real interval.

You are not supposed to create RealInterval objects yourself. Always use RealSet instead.

INPUT:

- lower – real or minus infinity; the lower bound of the interval.
- lower_closed – boolean; whether the interval is closed at the lower bound
- upper – real or (plus) infinity; the upper bound of the interval
- upper_closed – boolean; whether the interval is closed at the upper bound
- check – boolean; whether to check the other arguments for validity

```
contains(x)

Return whether \( x \) is contained in the interval

**INPUT:**

- \( x \) – a real number.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```sage
sage: i = RealSet.open_closed(0, 2)[0]; i
(0, 2]
sage: i.contains(0)
False
sage: i.contains(1)
True
sage: i.contains(2)
True
```

convex_hull(other)

Return the convex hull of the two intervals

**OUTPUT:**

The convex hull as a new RealInterval.

**EXAMPLES:**

```sage
sage: I1 = RealSet.open(0, 1)[0]; I1
(0, 1)
sage: I2 = RealSet.closed(1, 2)[0]; I2
[1, 2]
sage: I1.convex_hull(I2)
(0, 2]
sage: I2.convex_hull(I1)
(0, 2]
sage: I1.convex_hull(I2.interior())
(0, 2)
sage: I1.closure().convex_hull(I2.interior())
[0, 2]
sage: I1.closure().convex_hull(I2)
[0, 2]
sage: I3 = RealSet.closed(1/2, 3/2)[0]; I3
[1/2, 3/2]
sage: I1.convex_hull(I3)
(0, 3/2]
```

element_class

alias of `sage.rings.real_lazy.LazyFieldElement`
interior()  
Return the interior

OUTPUT:

The interior as a new RealInterval

EXAMPLES:

```python
sage: RealSet.closed(0, 1)[0].interior()
(0, 1)
sage: RealSet.open_closed(-oo, 1)[0].interior()
(-oo, 1)
sage: RealSet.closed_open(0, oo)[0].interior()
(0, +oo)
```

intersection(other)  
Return the intersection of the two intervals

INPUT:

• other — a RealInterval

OUTPUT:

The intersection as a new RealInterval

EXAMPLES:

```python
sage: I1 = RealSet.open(0, 2)[0]; I1
(0, 2)
sage: I2 = RealSet.closed(1, 3)[0]; I2
[1, 3]
sage: I1.intersection(I2)
[1, 2)
sage: I2.intersection(I1)
[1, 2)
sage: I1.closure().intersection(I2.interior())
(1, 2]
sage: I2.interior().intersection(I1.closure())
(1, 2]
sage: I3 = RealSet.closed(10, 11)[0]; I3
[10, 11]
sage: I1.intersection(I3)
(0, 0)
sage: I3.intersection(I1)
(0, 0)
```

is_connected(other)  
Test whether two intervals are connected

OUTPUT:

Boolean. Whether the set-theoretic union of the two intervals has a single connected component.

EXAMPLES:

```python
sage: I1 = RealSet.open(0, 1)[0]; I1
(0, 1)
sage: I2 = RealSet.closed(1, 2)[0]; I2
[1, 2]
```
sage: I1.is_connected(I2)
True
sage: I1.is_connected(I2.interior())
False
sage: I1.closure().is_connected(I2.interior())
True
sage: I2.is_connected(I1)
True
sage: I2.interior().is_connected(I1)
False
sage: I2.closure().is_connected(I1.interior())
True
sage: I3 = RealSet.closed(1/2, 3/2)[0]; I3
[1/2, 3/2]
sage: I1.is_connected(I3)
True
sage: I3.is_connected(I1)
True

is_empty()

Return whether the interval is empty

The normalized form of RealSet has all intervals non-empty, so this method usually returns False.

OUTPUT:

Boolean.

EXAMPLES:

sage: I = RealSet(0, 1)[0]
sage: I.is_empty()
False

is_point()

Return whether the interval consists of a single point

OUTPUT:

Boolean.

EXAMPLES:

sage: I = RealSet(0, 1)[0]
sage: I.is_point()
False

lower()

Return the lower bound

OUTPUT:

The lower bound as it was originally specified.

EXAMPLES:

sage: I = RealSet(0, 1)[0]
sage: I.lower()
0
lower_closed()
Return whether the interval is open at the lower bound

OUTPUT:
Boolean.

EXAMPLES:
```
sage: I = RealSet.open_closed(0, 1)[0]; I
(0, 1]
sage: I.lower_closed()
False
sage: I.lower_open()
True
sage: I.upper_closed()
True
sage: I.upper_open()
False
```

lower_open()
Return whether the interval is closed at the upper bound

OUTPUT:
Boolean.

EXAMPLES:
```
sage: I = RealSet.open_closed(0, 1)[0]; I
(0, 1]
sage: I.lower_closed()
False
sage: I.lower_open()
True
sage: I.upper_closed()
True
sage: I.upper_open()
False
```

upper()
Return the upper bound

OUTPUT:
The upper bound as it was originally specified.

EXAMPLES:
```
sage: I = RealSet(0, 1)[0]
sage: I.lower()
0
sage: I.upper()
1
```

upper_closed()
Return whether the interval is closed at the lower bound
OUTPUT:
Boolean.

EXAMPLES:

```python
sage: I = RealSet.open_closed(0, 1)[0]; I
(0, 1]
sage: I.lower_closed()
False
sage: I.lower_open()
True
sage: I.upper_closed()
True
sage: I.upper_open()
False
```

**upper_open()**
Return whether the interval is closed at the upper bound

OUTPUT:
Boolean.

EXAMPLES:

```python
sage: I = RealSet.open_closed(0, 1)[0]; I
(0, 1]
sage: I.lower_closed()
False
sage: I.lower_open()
True
sage: I.upper_closed()
True
sage: I.upper_open()
False
```

class sage.sets.real_set.RealSet(*intervals)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

A subset of the real line

INPUT:
Arguments defining a real set. Possibilities are either two real numbers to construct an open set or a list/tuple/iterable of intervals. The individual intervals can be specified by either a `RealInterval`, a tuple of two real numbers (constructing an open interval), or a list of two number (constructing a closed interval).

EXAMPLES:

```python
sage: RealSet(0,1)  # open set from two numbers
(0, 1)
sage: i = RealSet(0,1)[0]
sage: RealSet(i)  # interval
(0, 1)
sage: RealSet(i, (3,4))  # tuple of two numbers = open set
(0, 1) + (3, 4)
sage: RealSet(i, [3,4])  # list of two numbers = closed set
(0, 1) + [3, 4]
```
an_element()
Return a point of the set

OUTPUT:
A real number. ValueError if the set is empty.

EXAMPLES:

```
sage: RealSet.open_closed(0, 1).an_element()
sage: 1
sage: RealSet(0, 1).an_element()
sage: 1/2
sage: RealSet(-oo, +oo).an_element()
sage: 0
sage: RealSet(-oo, 7).an_element()
sage: 6
sage: RealSet(7, +oo).an_element()
sage: 8
```

static are_pairwise_disjoint(*real_set_collection)
Test whether sets are pairwise disjoint

INPUT:
- *real_set_collection – a list/tuple/iterable of RealSet.

OUTPUT:
Boolean.

EXAMPLES:

```
sage: s1 = RealSet((0, 1), (2, 3))
sage: s2 = RealSet((1, 2))
sage: s3 = RealSet.point(3)
sage: RealSet.are_pairwise_disjoint(s1, s2, s3)
sage: True
sage: RealSet.are_pairwise_disjoint(s1, s2, s3, [10, 10])
sage: True
sage: RealSet.are_pairwise_disjoint(s1, s2, s3, [-1, 1/2])
sage: False
```

cardinality()
Return the cardinality of the subset of the real line.

OUTPUT:
Integer or infinity. The size of a discrete set is the number of points; the size of a real interval is Infinity.

EXAMPLES:

```
sage: RealSet([0, 0], [1, 1], [3, 3]).cardinality()
sage: 3
sage: RealSet(0, 3).cardinality()
sage: +Infinity
```

static closed(lower, upper)
Construct a closed interval

INPUT:
- lower, upper – two real numbers or infinity. They will be sorted if necessary.
OUTPUT:

A new RealSet.

EXAMPLES:

```sage
sage: RealSet.closed(1, 0)
[0, 1]
```

**static closed_open**(lower, upper)

Construct an half-open interval

INPUT:

• lower, upper – two real numbers or infinity. They will be sorted if necessary.

OUTPUT:

A new RealSet that is closed at the lower bound and open an the upper bound.

EXAMPLES:

```sage
sage: RealSet.closed_open(1, 0)
[0, 1)
```

**complement()**

Return the complement

OUTPUT:

The set-theoretic complement as a new RealSet.

EXAMPLES:

```sage
sage: RealSet(0,1).complement()
(-oo, 0] + [1, +oo)
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1
(0, 2) + [10, +oo)
sage: s1.complement()
(-oo, 0] + [2, 10)
sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s2
(-oo, -10] + (1, 3)
sage: s2.complement()
(-10, 1] + [3, +oo)
```

**contains**(x)

Return whether x is contained in the set

INPUT:

• x – a real number.

OUTPUT:

Boolean.

EXAMPLES:

```sage
sage: s = RealSet(0,2) + RealSet.unbounded_above_closed(10); s
(0, 2) + [10, +oo)
sage: s.contains(1)
```

(continues on next page)

```
True
sage: s.contains(0)
False
sage: 10 in s  # syntactic sugar
True

difference(*other)

Return `self` with `other` subtracted

**INPUT:**

- other – a `RealSet` or data that defines one.

**OUTPUT:**

The set-theoretic difference of `self` with `other` removed as a new `RealSet`.

**EXAMPLES:**

```
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1
(0, 2) + [10, +oo)
sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s2
(-oo, -10] + (1, 3)
sage: s1.difference(s2)
(0, 1] + [10, +oo)
sage: s1 - s2  # syntactic sugar
(0, 1] + [10, +oo)
sage: s2.difference(s1)
(-oo, -10] + [2, 3)
sage: s2 - s1  # syntactic sugar
(-oo, -10] + [2, 3)
sage: s1.difference(1,11)
(0, 1] + [11, +oo)
```

**get_interval(i)**

Return the `i`-th connected component.

Note that the intervals representing the real set are always normalized, see `normalize()`.

**INPUT:**

- `i` – integer.

**OUTPUT:**

The `i`-th connected component as a `RealInterval`.

**EXAMPLES:**

```
sage: s = RealSet(RealSet.open_closed(0,1), RealSet.closed_open(2,3))
sage: s.get_interval(0)
(0, 1]
sage: s[0]  # shorthand
(0, 1]
sage: s.get_interval(1)
[2, 3)
sage: s[0] == s.get_interval(0)
True
```

**inf()**

Return the infimum
OUTPUT:

A real number or infinity.

EXAMPLES:

```
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1
      (0, 2) + [10, +oo)
sage: s1.inf()
      0

sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s2
      (-oo, -10] + (1, 3)
sage: s2.inf()
      -Infinity
```

**intersection** (*other*)

Return the intersection of the two sets

INPUT:

- *other* – a `RealSet` or data defining one.

OUTPUT:

The set-theoretic intersection as a new `RealSet`.

EXAMPLES:

```
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1
      (0, 2) + [10, +oo)
sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s2
      (-oo, -10] + (1, 3)
sage: s1.intersection(s2)
      (1, 2)
sage: s1 & s2       # syntactic sugar
      (1, 2)

sage: s1 = RealSet((0, 1), (2, 3)); s1
      (0, 1) + (2, 3)
sage: s2 = RealSet([0, 1], [2, 3]); s2
      [0, 1] + [2, 3]
sage: s3 = RealSet([1, 2]); s3
      [1, 2]
sage: s1.intersection(s2)
      (0, 1) + (2, 3)
sage: s1.intersection(s3)
      {}
sage: s2.intersection(s3)
      {}
```

**is_disjoint_from** (*other*)

Test whether the two sets are disjoint

INPUT:

- *other* – a `RealSet` or data defining one.

OUTPUT:

Boolean.

EXAMPLES:
sage: s1 = RealSet((0, 1), (2, 3)); s1
(0, 1) + (2, 3)
sage: s2 = RealSet([1, 2]); s2
[1, 2]
sage: s1.is_disjoint_from(s2)
True
sage: s1.is_disjoint_from([1, 2])
True

is_empty()
Return whether the set is empty

EXAMPLES:

sage: RealSet(0, 1).is_empty()
False
sage: RealSet(0, 0).is_empty()
True

is_included_in(*other)
Tests interval inclusion

INPUT:

- *args – a RealSet or something that defines one.

OUTPUT:

Boolean.

EXAMPLES:

sage: I = RealSet((1,2))
sage: J = RealSet((1,3))
sage: K = RealSet((2,3))
sage: I.is_included_in(J)
True
sage: J.is_included_in(K)
False

n_components()
Return the number of connected components

See also get_interval()

EXAMPLES:

sage: s = RealSet(RealSet.open_closed(0,1), RealSet.closed_open(2,3))
sage: s.n_components()
2

static normalize(intervals)
Bring a collection of intervals into canonical form

INPUT:

- intervals – a list/tuple/iterable of intervals.

OUTPUT:

A tuple of intervals such that
they are sorted in ascending order (by lower bound)
• there is a gap between each interval
• all intervals are non-empty

EXAMPLES:

```
sage: i1 = RealSet((0, 1))[0]
sage: i2 = RealSet([1, 2])[0]
sage: i3 = RealSet((2, 3))[0]
sage: RealSet.normalize([i1, i2, i3])
((0, 3),)
sage: RealSet((0, 1), [1, 2], (2, 3))
(0, 3)
sage: RealSet((0, 1), [1, 2], (2, 3))
(0, 1) + (1, 2) + (2, 3)
sage: RealSet([0, 1], [2, 3])
[0, 1] + [2, 3]
sage: RealSet((0, 2), (1, 3))
(0, 3)
sage: RealSet(0,0)
{}``

**static open** (*lower*, *upper*)
Construct an open interval

INPUT:
• *lower*, *upper* – two real numbers or infinity. They will be sorted if necessary.

OUTPUT:
A new `RealSet`.

EXAMPLES:

```
sage: RealSet.open(1, 0)
(0, 1)
```

**static open_closed** (*lower*, *upper*)
Construct a half-open interval

INPUT:
• *lower*, *upper* – two real numbers or infinity. They will be sorted if necessary.

OUTPUT:
A new `RealSet` that is open at the lower bound and closed at the upper bound.

EXAMPLES:

```
sage: RealSet.open_closed(1, 0)
[0, 1]
```

**static point** (*p*)
Construct an interval containing a single point

INPUT:
• *p* – a real number.
OUTPUT:

A new \texttt{RealSet}.

EXAMPLES:

\begin{verbatim}
sage: RealSet.open(1, 0)
(0, 1)
\end{verbatim}

\texttt{sup}()

Return the supremum

OUTPUT:

A real number or infinity.

EXAMPLES:

\begin{verbatim}
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1
(0, 2) + [10, +\infty)
sage: s1.sup()
+\infty
sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s2
(-\infty, -10] + (1, 3)
sage: s2.sup()
3
\end{verbatim}

\texttt{static unbounded_above_closed}(\texttt{bound})

Construct a semi-infinite interval

INPUT:

• \texttt{bound} – a real number.

OUTPUT:

A new \texttt{RealSet} from the bound (including) to plus infinity.

EXAMPLES:

\begin{verbatim}
sage: RealSet.unbounded_above_closed(1)
[1, +\infty)
\end{verbatim}

\texttt{static unbounded_above_open}(\texttt{bound})

Construct a semi-infinite interval

INPUT:

• \texttt{bound} – a real number.

OUTPUT:

A new \texttt{RealSet} from the bound (excluding) to plus infinity.

EXAMPLES:

\begin{verbatim}
sage: RealSet.unbounded_above_open(1)
(1, +\infty)
\end{verbatim}

\texttt{static unbounded_below_closed}(\texttt{bound})

Construct a semi-infinite interval

INPUT:
• **bound** – a real number.

**OUTPUT:**

A new *RealSet* from minus infinity to the bound (including).

**EXAMPLES:**

```python
sage: RealSet.unbounded_below_closed(1)
(-oo, 1]
```

**static unbounded_below_open**(bound)

Construct a semi-infinite interval

**INPUT:**

• **bound** – a real number.

**OUTPUT:**

A new *RealSet* from minus infinity to the bound (excluding).

**EXAMPLES:**

```python
sage: RealSet.unbounded_below_open(1)
(-oo, 1)
```

**union(**other**)

Return the union of the two sets

**INPUT:**

• **other** – a *RealSet* or data that defines one.

**OUTPUT:**

The set-theoretic union as a new *RealSet*.

**EXAMPLES:**

```python
sage: s1 = RealSet(0,2)
sage: s2 = RealSet(1,3)
sage: s1.union(s2)
(0, 3)
sage: s1.union(1,3)
(0, 3)
sage: s1 | s2  # syntactic sugar
(0, 3)
sage: s1 + s2  # syntactic sugar
(0, 3)
```
INDICES AND TABLES

• Index
• Module Index
• Search Page
s
sage.sets.cartesian_product, 1
sage.sets.disjoint_set, 22
sage.sets.disjoint_union Enumerated Sets, 29
sage.sets.family, 3
sage.sets.finiteEnumerated Set, 39
sage.sets.finite_set_map_cy, 65
sage.sets.finite_set_maps, 61
sage.sets.integer_range, 75
sage.sets.non_negative_integers, 80
sage.sets.positive_integers, 80
sage.sets.primes, 82
sage.sets.pythonclass, 73
sage.sets.real_set, 83
sage.sets.recursively enumerated_set, 41
sage.sets.set, 12
sage.sets.set_from_iterator, 33
sage.sets.totally ordered finite_set, 71
INDEX

Symbols

_cartesian_product_of_elements() (sage.sets.cartesian_product.CartesianProduct method), 1

A

AbstractFamily (class in sage.sets.family), 3
an_element() (sage.sets.cartesian_product.CartesianProduct method), 2
an_element() (sage.sets.disjoint_union EnumeratedSets method), 32
an_element() (sage.sets.finite_enumerated_set.FiniteEnumeratedSet method), 40
an_element() (sage.sets.finite_set_maps.FiniteSetEndoMaps_N method), 61
an_element() (sage.sets.finite_set_maps.FiniteSetMaps_MN method), 64
an_element() (sage.sets.non_negative_integers.NonNegativeIntegers method), 81
an_element() (sage.sets.positive_integers.PositiveIntegers method), 80
an_element() (sage.sets.real_set.RealSet method), 89
an_element() (sage.sets.set.Set_object method), 13
are_pairwise_disjoint() (sage.sets.real_set.RealSet static method), 90

B

breadth_first_search_iterator() (sage.sets.recursively enumerated set.RecursivelyEnumeratedSet forest method), 50
breadth_first_search_iterator() (sage.sets.recursively enumerated set.RecursivelyEnumeratedSet_generic method), 53
breadth_first_search_iterator() (sage.sets.recursively enumerated set.RecursivelyEnumeratedSet_graded method), 57
breadth_first_search_iterator() (sage.sets.recursively enumerated set.RecursivelyEnumeratedSet_symmetric method), 58

C

cardinality() (sage.sets.disjoint_set.DisjointSet_class method), 23
cardinality() (sage.sets.disjoint_union EnumeratedSets method), 32
cardinality() (sage.sets.family.EnumeratedFamily method), 4
cardinality() (sage.sets.family.FiniteFamily method), 10
cardinality() (sage.sets.family.LazyFamily method), 11
cardinality() (sage.sets.family.TrivialFamily method), 11
cardinality() (sage.sets.finite enumerated set.FiniteEnumeratedSet method), 40
cardinality() (sage.sets.finite_set_maps.FiniteSetMaps method), 63
cardinality() (sage.sets.integer_range.IntegerRangeFinite method), 77
cardinality() (sage.sets.pythonclass.Set_PythonType_class method), 74
cardinality() (sage.sets.real_set.RealSet method), 90
cardinality() (sage.sets.set.Set_object method), 14
cardinality() (sage.sets.set.Set_objectEnumerated method), 17
cardinality() (sage.sets.set.Set_objectUnion method), 21
cartesian_factors() (sage.sets.cartesian_product.CartesianProduct method), 2
cartesian_factors() (sage.sets.cartesian_product.CartesianProductElement method), 2
cartesian_projection() (sage.sets.cartesian_product.CartesianProduct method), 3
CartesianProduct (class in sage.sets.cartesian_product), 1
CartesianProduct.Element (class in sage.sets.cartesian_product), 2
cartesian_product() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 66
cartesian_product() (sage.sets.recursivelyEnumerated_set.RecursivelyEnumeratedSetForest method), 51
clear_cache() (sage.sets.setFromIterator.EnumeratedSetFromIterator method), 35
closed() (sage.sets.real_set.RealSet static method), 90
closed_open() (sage.sets.real_set.RealSet static method), 91
closure() (sage.sets.real_set.InternalRealInterval method), 84
codomain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 66
codomain() (sage.sets.finite_set_maps.FiniteSetMaps_MN method), 64
codomain() (sage.sets.finite_set_maps.FiniteSetMaps_Set method), 65
complement() (sage.sets.real_set.RealSet method), 91
closure() (sage.sets.cartesian_product.CartesianProduct method), 3
contains() (sage.sets.real_set.InternalRealInterval method), 85
contains() (sage.sets.real_set.RealSet method), 91
convex_hull() (sage.sets.real_set.InternalRealInterval method), 85

D

Decorator (class in sage.sets.set_from_iterator), 34
depth_first_search_iterator() (sage.sets.recursivelyEnumerated_set.RecursivelyEnumeratedSetForest method), 51
depth_first_search_iterator() (sage.sets.recursivelyEnumerated_set.RecursivelyEnumeratedSetGeneric method), 54
difference() (sage.sets.real_set.RealSet method), 92
difference() (sage.sets.set.Set_object method), 14
difference() (sage.sets.set.Set_objectEnumerated method), 17
DisjointSet () (in module sage.sets.disjoint_set), 23
DisjointSet_class (class in sage.sets.disjoint_set), 23
DisjointSet_of_hashables (class in sage.sets.disjoint_set), 24
DisjointSet_of_integers (class in sage.sets.disjoint_set), 26
DisjointUnionEnumeratedSets (class in sage.sets.disjointUnionEnumeratedSets), 29
domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 67
domain() (sage.sets.finite_set_maps.FiniteSetMaps_MN method), 64
domain() (sage.sets.finite_set_maps.FiniteSetMaps_Set method), 65
DummyExampleForPicklingTest (class in sage.sets.set_from_iterator), 34

E

Element (sage.sets.finite_set_maps.FiniteSetEndoMaps_N attribute), 61
Element (sage.sets.finite_set_maps.FiniteSetEndoMaps_Set attribute), 62
Element (sage.sets.finite_set_maps.FiniteSetMaps_N attribute), 64
Element (sage.sets.finite_set_maps.FiniteSetMaps_Set attribute), 65
Element (sage.sets.non_negative_integers.NonNegativeIntegers attribute), 81
Element (sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet attribute), 72
Element () (sage.sets.disjoint_unionEnumeratedSets.DisjointUnionEnumeratedSets method), 32
element_class (sage.sets.integer_range.IntegerRange attribute), 77
element_class (sage.sets.real_set.InternalRealInterval attribute), 85
element_to_root_dict () (sage.sets.disjoint_set.DisjointSetOfHashtables method), 24
element_to_root_dict () (sage.sets.disjoint_set.DisjointSetOfIntegers method), 27
elements_of_depth_iterator () (sage.sets.recursivelyEnumeratedSet.ReCURSivelyEnumeratedSetForest
method), 51
elements_of_depth_iterator () (sage.sets.recursivelyEnumeratedSet.ReCURSivelyEnumeratedSetGeneric
method), 54
EnumeratedFamily (class in sage.sets.family), 4
EnumeratedSetFromIterator (class in sage.sets.set_from_iterator), 34
EnumeratedSetFromIterator_function_decorator (class in sage.sets.set_from_iterator), 36
EnumeratedSetFromIterator_method_caller (class in sage.sets.set_from_iterator), 37
EnumeratedSetFromIterator_method_decorator (class in sage.sets.set_from_iterator), 38

F

f () (sage.sets.set_from_iterator.DummyExampleForPicklingTest method), 34
Family() (in module sage.sets.family), 5
fibers() (in module sage.sets.finite_set_map_cy), 70
fibers() (sage.sets.finite_set_map_cy.FiniteSetMapMN method), 67
fibers_args() (in module sage.sets.finite_set_map_cy), 70
find() (sage.sets.disjoint_set.DisjointSetOfHashtables method), 25
find() (sage.sets.disjoint_set.DisjointSetOfIntegers method), 27
FiniteEnumeratedSet (class in sage.sets.finiteEnumeratedSet), 39
FiniteFamily (class in sage.sets.family), 9
FiniteFamilyWithHiddenKeys (class in sage.sets.family), 10
FiniteSetEndoMapN (class in sage.sets.finite_set_map_cy), 66
FiniteSetEndoMapSet (class in sage.sets.finite_set_map_cy), 66
FiniteSetEndoMapsN (class in sage.sets.finite_set_maps), 61
FiniteSetEndoMapsSet (class in sage.sets.finite_set_maps), 61
FiniteSetMapMN (class in sage.sets.finite_set_map_cy), 66
FiniteSetMapSet (class in sage.sets.finite_set_map_cy), 68
FiniteSetMapSet_from_dict () (in module sage.sets.finite_set_map_cy), 70
FiniteSetMapSet_from_list () (in module sage.sets.finite_set_map_cy), 70
FiniteSetMaps (class in sage.sets.finite_set_maps), 62
FiniteSetMapsMN (class in sage.sets.finite_set_maps), 64
FiniteSetMapsSet (class in sage.sets.finite_set_maps), 64
first() (sage.sets.finiteEnumeratedSet.FiniteEnumeratedSet method), 40
first() (sage.sets.primes.Primes method), 82
from_dict() (sage.sets.finite_set_map_cy.FiniteSetMapSet class method), 68
from_dict() (sage.sets.finite_set_maps.FiniteSetMapsSet method), 65
from_integer (sage.sets.non_negative_integers.NonNegativeIntegers attribute), 81
from_list() (sage.sets.finite_set_map_cy.FiniteSetMapSet class method), 68
frozenset() (sage.sets.set_objectEnumerated method), 18

G

g_get_interval() (sage.sets.real_set.RealSets method), 92
getimage() (sage.sets.finite_set_map_cy.FiniteSetMapMN method), 67
getimage() (sage.sets.finite_set_map_cy.FiniteSetMapSet method), 69
graded_component() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic method), 54
graded_component() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_graded method), 57
graded_component() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_symmetric method), 59
graded_component_iterator() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic method), 55
graded_component_iterator() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_graded method), 57
graded_component_iterator() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_symmetric method), 59

H

has_finite_length() (in module sage.sets), 22
has_key() (sage.sets.family.FiniteFamily method), 10
hidden_keys() (sage.sets.family.AbstractFamily method), 3
hidden_keys() (sage.sets.family.FiniteFamilyWithHiddenKeys method), 10

I

image_set() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 67
image_set() (sage.sets.finite_set_map_cy.FiniteSetMap_Set method), 69
index() (sage.sets.finite_enumerated_set.FiniteEnumeratedSet method), 40
inf() (sage.sets.real_set.RealSet method), 92
IntegerRange (class in sage.sets.integer_range), 75
IntegerRangeEmpty (class in sage.sets.integer_range), 77
IntegerRangeFinite (class in sage.sets.integer_range), 77
IntegerRangeFromMiddle (class in sage.sets.integer_range), 78
IntegerRangeInfinite (class in sage.sets.integer_range), 79
interior() (sage.sets.real_set.InternalRealInterval method), 85
InternalRealInterval (class in sage.sets.real_set), 84
intersection() (sage.sets.real_set.InternalRealInterval method), 86
intersection() (sage.sets.real_set.RealSet method), 93
intersection() (sage.sets.set.Set_object method), 14
intersection() (sage.sets.set.Set_object_enumerated method), 18
inverse_family() (sage.sets.family.AbstractFamily method), 4
is_connected() (sage.sets.real_set.InternalRealInterval method), 86
is_disjoint_from() (sage.sets.real_set.RealSet method), 93
is_empty() (sage.sets.real_set.InternalRealInterval method), 87
is_empty() (sage.sets.real_set.RealSet method), 94
is_empty() (sage.sets.set.Set_object method), 15
is_finite() (sage.sets.set.Set_object method), 15
is_finite() (sage.sets.set.Set_object_difference method), 17
is_finite() (sage.sets.set.Set_object_enumerated method), 18
is_finite() (sage.sets.set.Set_object_intersection method), 20
is_finite() (sage.sets.set.Set_object_symmetric_difference method), 21
is_finite() (sage.sets.set.Set_object_union method), 21
is_included_in() (sage.sets.real_set.RealSet method), 94
is_parent_of() (sage.sets.finite_enumerated_set.FiniteEnumeratedSet method), 40
is_parent_of() (sage.sets.set_from_iterator.EnumeratedSetFromIterator method), 35
is_point() (sage.sets.real_set.InternalRealInterval method), 87
issubset() (sage.sets.set.Set_object_enumerated method), 18
issuperset() (sage.sets.set.Set_object_enumerated method), 19
items() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 67
items() (sage.sets.finite_set_map_cy.FiniteSetMap_Set method), 69

K
keys() (sage.sets.family.FiniteFamily method), 10
keys() (sage.sets.family.LazyFamily method), 11
keys() (sage.sets.family.TrivialFamily method), 11

L
last() (sage.sets.finiteEnumeratedSet.FiniteEnumeratedSet method), 40
LazyFamily (class in sage.sets.family), 11
le() (sage.sets.totallyOrderedFiniteSet.TotallyOrderedFiniteSet method), 72
list() (sage.sets.finiteEnumeratedSet.FiniteEnumeratedSet method), 40
list() (sage.sets.set.Set_object_enumerated method), 19
lower() (sage.sets.real_set.InternalRealInterval method), 87
lower_closed() (sage.sets.real_set.InternalRealInterval method), 88
lower_open() (sage.sets.real_set.InternalRealInterval method), 88

M
map() (sage.sets.family.AbstractFamily method), 4
map_reduce() (sage.sets.recursivelyEnumeratedSet.RecursivelyEnumeratedSet_forest method), 52
module
devs
sage.sets.cartesian_product, 1
sage.sets.disjoint_set, 22
sage.sets.disjointUnionEnumeratedSets, 29
sage.sets.family, 3
sage.sets.finiteEnumeratedSet, 39
sage.sets.finiteSetMap, 65
sage.sets.finiteSetMaps, 61
sage.sets.integer_range, 75
sage.sets.non_negative_integers, 80
sage.sets.positive_integers, 80
sage.sets.primes, 82
sage.sets.pythonclass, 73
sage.sets.real_set, 83
sage.sets.recursivelyEnumeratedSet, 41
sage.sets.set, 12
sage.sets.setFromIterator, 33
sage.sets.totallyOrderedFiniteSet, 71

N
n_components() (sage.sets.real_set.RealSet method), 94
naive_search_iterator() (sage.sets.recursivelyEnumeratedSet.RecursivelyEnumeratedSet_forest method), 55
next() (sage.sets.integer_range.IntegerRangeFromMiddle method), 79
next() (sage.sets.non_negative_integers.NonNegativeIntegers method), 81
next() (sage.sets.primes.Primes method), 82
NonNegativeIntegers (class in sage.sets.non_negative_integers), 80
normalize() (sage.sets.real_set.RealSet static method), 94
number_of_subsets() (sage.sets.disjoint_set.DisjointSet_class method), 24
O

object() (sage.sets.pythonclass.Set_PythonType_class method), 74
object() (sage.sets.set.Set_object method), 15
one() (sage.sets.finite_set_maps.FiniteSetEndoMaps_N method), 61
open() (sage.sets.real_set.RealSet static method), 95
open_closed() (sage.sets.real_set.RealSet static method), 95

P

point() (sage.sets.real_set.RealSet static method), 95
PositiveIntegers (class in sage.sets.positive_integers), 80
Primes (class in sage.sets.primes), 82

R

random_element() (sage.sets.finite_enumerated_set.FiniteEnumeratedSet method), 40
random_element() (sage.sets.set.Set_object enumerated method), 19
rank() (sage.sets.finite_enumerated_set.FiniteEnumeratedSet method), 41
rank() (sage.sets.integer_range.IntegerRangeFinite method), 78
rank() (sage.sets.integer_range.IntegerRangeInfinite method), 79
RealSet (class in sage.sets.real_set), 89
RecursivelyEnumeratedSet() (in module sage.sets.recursively_enumerated_set), 45
RecursivelyEnumeratedSet_forest (class in sage.sets.recursively_enumerated_set), 47
RecursivelyEnumeratedSet_generic (class in sage.sets.recursively_enumerated_set), 53
RecursivelyEnumeratedSet_graded (class in sage.sets.recursively_enumerated_set), 56
RecursivelyEnumeratedSet_symmetric (class in sage.sets.recursively_enumerated_set), 58
root_to_elements_dict() (sage.sets.disjoint_set.DisjointSet_of_hashables method), 25
root_to_elements_dict() (sage.sets.disjoint_set.DisjointSet_of_integers method), 27
roots() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_forest method), 53

S

sage.sets.cartesian_product
 module, 1
sage.sets.disjoint_set
 module, 22
sage.sets.disjoint_union Enumerated sets
 module, 29
sage.sets.family
 module, 3
sage.sets.finite_enumerated_set
 module, 39
sage.sets.finite_set_map cy
 module, 65
sage.sets.finite_set_maps
 module, 61
sage.sets.integer_range
 module, 75
sage.sets.non_negative_integers
 module, 80
sage.sets.positive_integers
 module, 80
sage.sets.primes
module, 82
sage.sets.pythonclass
module, 73
sage.sets.real_set
module, 83
sage.sets.recursivelyEnumerated_set
module, 41
sage.sets.set
module, 12
sage.sets.set_from_iterator
module, 12
sage.sets.totallyOrdered_finite_set
module, 71
search_forest_iterator() (in module sage.sets.recursivelyEnumerated_set), 60
seeds() (sage.sets.recursivelyEnumerated_set.RecursivelyEnumeratedSet_generic method), 55
Set() (in module sage.sets.set), 12
set() (sage.sets.Set_object_enumerated method), 19
set_from_function (in module sage.sets.set_from_iterator), 39
set_from_method (in module sage.sets.set_from_iterator), 39
Set_object (class in sage.sets.set), 13
Set_object_binary (class in sage.sets.set), 16
Set_object_difference (class in sage.sets.set), 17
Set_object_enumerated (class in sage.sets.set), 17
Set_object_intersection (class in sage.sets.set), 20
Set_object_symmetric_difference (class in sage.sets.set), 21
Set_object_union (class in sage.sets.set), 21
Set_PythonType() (in module sage.sets.pythonclass), 73
Set_PythonType_class (class in sage.sets.pythonclass), 73
setImage() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 68
setImage() (sage.sets.finite_set_map_cy.FiniteSetMap_Set method), 69
some_elements() (sage.sets.nongaussian_integers.NongaussianIntegers method), 81
subsets() (sage.sets.set.Set_object method), 16
subsets_lattice() (sage.sets.Set_object method), 16
successors (sage.sets.recursivelyEnumerated_set.RecursivelyEnumeratedSet_generic attribute), 55
sup() (sage.sets.real_set.RealSet method), 96
symmetric_difference() (sage.sets.Set_object method), 16
symmetric_difference() (sage.sets.Set_object_enumerated method), 20

to_digraph() (sage.sets.disjoint_set.DisjointSet_of_hashables method), 25
to_digraph() (sage.sets.disjoint_set.DisjointSet_of_integers method), 28
to_digraph() (sage.sets.recursivelyEnumerated_set.RecursivelyEnumeratedSet_generic method), 55
TotallyOrderedFiniteSet (class in sage.sets.totallyOrdered_finite_set), 71
TotallyOrderedFiniteSetElement (class in sage.sets.totallyOrdered_finite_set), 73
TrivialFamily (class in sage.sets.family), 11

unbounded_above_closed() (sage.sets.real_set.RealSet static method), 96
unbounded_above_open() (sage.sets.real_set.RealSet static method), 96
unbounded_below_closed() (sage.sets.real_set.RealSet static method), 96
unbounded_below_open() (sage.sets.real_set.RealSet static method), 97
union() (sage.sets.disjoint_set.DisjointSet_of_hashables method), 26
union() (sage.sets.disjoint_set.DisjointSet_of_integers method), 28
union() (sage.sets.real_set.RealSet method), 97
union() (sage.sets.set.Set_object method), 16
union() (sage.sets.set.Set_object_enumerated method), 20
unrank() (sage.sets.finiteEnumeratedSet.FiniteEnumeratedSet method), 41
unrank() (sage.sets.integer_range.IntegerRangeFinite method), 78
unrank() (sage.sets.integer_range.IntegerRangeInfinite method), 79
unrank() (sage.sets.non_negative_integers.NonNegativeIntegers method), 82
unrank() (sage.sets.primes.Primes method), 83
unrank() (sage.sets.set_from_iterator.EnumeratedSetFromIterator method), 36
upper() (sage.sets.real_set.InternalRealInterval method), 88
upper_closed() (sage.sets.real_set.InternalRealInterval method), 88
upper_open() (sage.sets.real_set.InternalRealInterval method), 89

V
values() (sage.sets.family.FiniteFamily method), 10

W
wrapped_class (sage.sets.cartesian_product.CartesianProduct.Element attribute), 2

Z
zip() (sage.sets.family.AbstractFamily method), 4