Sets

Release 10.2

The Sage Development Team

Dec 06, 2023
## CONTENTS

1 Set Constructions .................................................. 1
2 Sets of Numbers .................................................... 83
3 Indices and Tables .................................................. 115
Python Module Index ................................................ 117
Index ........................................................................ 119
1.1 Cartesian products

AUTHORS:

• Nicolas Thiery (2010-03): initial version

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

```python
sage: G = cartesian_product([GF(5), Permutations(10)])
sage: G.cartesian_factors()
(Finite Field of size 5, Standard permutations of 10)
sage: G.cardinality()
18144000
sage: G.random_element()  # random
(1, [4, 7, 6, 5, 10, 1, 3, 2, 8, 9])
sage: G.category()
Join of Category of finite monoids and Category of Cartesian products of monoids and Category of Cartesian products of finite enumerated sets
```

```python
_cartesian_product_of_elements(elements)
```

Return the Cartesian product of the given elements.


INPUT:

• `elements` – an iterable (e.g. tuple, list) with one element of each Cartesian factor of `self`

```python
Warning:  This is meant as a fast low-level method.  In particular, no coercion is attempted.  When coercion or sanity checks are desirable, please use instead `self(elements)` or `self._element_constructor_(elements)`.
```

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: ElementWrapperCheckWrappedClass
cartesian_factors()
Return the tuple of elements that compose this element.

EXAMPLES:

```sage
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
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 tuple

an_element()

EXAMPLES:

```sage
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:

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

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
• Florent Hivert (2008-04): various fixes, cleanups and improvements.

```python
class sage.sets.family.AbstractFamily
    Bases: 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:

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:

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}
```

```python
items()
    Return an iterator for key-value pairs.

A key can only appear once, but if the function is not injective, values may appear multiple times.

EXAMPLES:

sage: f = Family([-2, -1, 0, 1, 2], abs)
sage: list(f.items())
[(-2, 2), (-1, 1), (0, 0), (1, 1), (2, 2)]
```

```python
keys()
    Return the keys of the family.

EXAMPLES:

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

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

Todo: good name?

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

values()
Return the elements (values) of this family.
EXAMPLES:

sage: f = Family(['c', 'a', 'b'], lambda x: x + x)
sage: sorted(f.values())
['aa', 'bb', 'cc']

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:

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: LazyFamily

EnumeratedFamily turns an enumerated set \(c\) into a family indexed by the set \(\{0, \ldots, |c| - 1\}\) (or \(\mathbb{N}\) if \(|c|\) is countably infinite).

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

cardinality()
Return the number of elements in self.

EXAMPLES:

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 $\ell$ is `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'
sage: len(f)
3
sage: list(f)
['a', 'b', 'd']
sage: [ x for x in f ]
['a', 'b', 'd']
sage: f.keys()
[3, 4, 7]
sage: 'b' in f
True
```

(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
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:

```python
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:

```python
sage: f = Family(Permutations(3), attrcall("to_lehmer_code"))
sage: list(f)
[[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 0], [2, 0, 0], [2, 1, 0]]
sage: f.cardinality()
6
```

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:

```python
sage: f = Family(Permutations(3), lambda p: p.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())_{i in Standard permutations of 3}
sage: f == loads(dumps(f))
True
```

But this one does not:

```python
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:

```python
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]
sage: f[7]
14
sage: f[2]
4
sage: list(f)
```

(continues on next page)
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]
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: len(f)
3
```

(continues on next page)
Family accept finite and infinite EnumeratedSets as input:

```python
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: `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 \(f_7 = d\):

```python
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:

```python
sage: f[7]
'd'
```

And the other usual dictionary operations are also available:

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

cardinality()

Returns the number of elements in self.

```python
definite = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})
sage: definite.cardinality()
3
```

has_key(k)

Returns whether k is a key of self

```python
Family({'a':1, 'b':2, 'c':3}).has_key('a')
True

Family({'a':1, 'b':2, 'c':3}).has_key('d')
False
```

keys()

Returns the index set of this family

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

values()

Returns the elements of this family

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

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

Bases: 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: `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: `AbstractFamily`

`TrivialFamily` turns a list/tuple `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 TrivialFamily
sage: f = TrivialFamily([3,4,7])
sage: f.cardinality()
3
```
keys()
Returns self's keys.

EXAMPLES:

```
sage: from sage.sets.family import TrivialFamily
sage: f = TrivialFamily([3,4,7])
sage: f.keys()
[0, 1, 2]
```

map(f, name=None)
Return the family \( f(\text{self}[i]) \) for \( i \in I \), where \( I \) is the index set of self.
The result is again a \texttt{TrivialFamily}.

EXAMPLES:

```
sage: from sage.sets.family import TrivialFamily
sage: f = TrivialFamily(['a', 'b', 'd'])
sage: g = f.map(lambda x: x + '1'); g
Family ('a1', 'b1', 'd1')
```

### 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 \	exttt{Set_object_binary}, fixed \texttt{__hash__}.

\texttt{sage.sets.set.Set}(X=None, category=None)
Create the underlying set of \( X \).

If \( X \) is a list, tuple, Python set, or \( X \text{.is_finite()} \) is True, this returns a wrapper around Python’s enumerated immutable \texttt{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:

```
sage: # needs sage.rings.finite_rings
sage: X = Set(GF(9, 'a'))
sage: X
{0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2}
sage: type(X)
<class 'sage.sets.set.Set_object_enumerated_with_category'>
sage: Y = X.union(Set(QQ))
sage: Y
Set-theoretic union of
```

(continues on next page)
\{0, 1, 2, a, a + 1, a + 2, 2^a, 2^a + 1, 2^a + 2\} and
Set of elements of Rational Field
\texttt{sage: type(Y)}
\texttt{<class 'sage.sets.set.Set_object_union_with_category'>}

Usually sets can be used as dictionary keys.

\texttt{sage: # needs sage.symbolic}
\texttt{sage: d = \{Set([2*I, 1 + I]): 10\}}
\texttt{sage: d} # key is randomly ordered
\texttt{\{{I + 1, 2*I}: 10\}}
\texttt{sage: d[Set([1+I,2*I])]} 10
\texttt{sage: d[Set((1+I,2*I))]} 10

The original object is often forgotten.

\texttt{sage: v = [1,2,3]}
\texttt{sage: X = Set(v)}
\texttt{sage: X} \{1, 2, 3\}
\texttt{sage: v.append(5)}
\texttt{sage: X} \{1, 2, 3\}
\texttt{sage: 5 in X} False

Set also accepts iterators, but be careful to only give \textit{finite} sets:

\texttt{sage: \texttt{sorted} (Set(range(1,6)))}
\{1, 2, 3, 4, 5\}
\texttt{sage: \texttt{sorted} (Set(list(range(1,6))))}
\{1, 2, 3, 4, 5\}
\texttt{sage: \texttt{sorted} (Set(iter(range(1,6))))}
\{1, 2, 3, 4, 5\}

We can also create sets from different types:

\texttt{sage: \texttt{sorted} (Set([Sequence([3,1], immutable=True), 5, QQ, Partition([3,1,1])]),…)
\texttt{˓→key=str} # needs sage.combinat}
\{5, Rational Field, [3, 1, 1], [3, 1]\}

Sets with unhashable objects work, but with less functionality:

\texttt{sage: A = Set([\texttt{QQ}, (3, 1), 5])} # hashable
\texttt{sage: \texttt{sorted} (A.list(), key=\texttt{repr})}
\{[(3, 1), 5, \text{Rational Field}\}
\texttt{sage: type(A)}
\texttt{<class 'sage.sets.set.Set_object_enumerated_with_category'>}
\texttt{sage: B = Set([\texttt{QQ}, [3, 1], 5])} # unhashable
\texttt{sage: \texttt{sorted} (B.list(), key=\texttt{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_add_sub_operators  
Bases: object  
Mix-in class providing the operators __add__ and __sub__.  
The operators delegate to the methods union and intersection, which need to be implemented by the class.

class sage.sets.set.Set_base  
Bases: object  
Abstract base class for sets, not necessarily parents.

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'))); X  
needs sage.rings.finite_rings
{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'))); X  
needs sage.rings.finite_rings
{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:

```python
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'))); X  
needs sage.rings.finite_rings
```

(continues on next page)
{}  

```python
sage: X = Set(GF(9, 'b')).intersection(Set(GF(27, 'b'))); X
˓→ needs sage.rings.finite_rings
{}
```

**symmetric_difference(X)**

Returns the symmetric difference of **self** and **X**.

**EXAMPLES:**

```python
sage: X = Set([1,2,3]).symmetric_difference(Set([3,4])); X
sage: X
{1, 2, 4}
```

**union(X)**

Return the union of **self** and **X**.

**EXAMPLES:**

```python
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  #needs sage.libs.pari
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]
```

**class** `sage.sets.set.Set_boolean_operators`

Bases: `object`

Mix-in class providing the Boolean operators `__or__`, `__and__`, `__xor__`.

The operators delegate to the methods `union`, `intersection`, and `symmetric_difference`, which need to be implemented by the class.

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

Bases: `Set_generic`, `Set_base`, `Set_boolean_operators`, `Set_add_sub_operators`

A set attached to an almost arbitrary object.

**EXAMPLES:**

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

**cardinality()**

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

EXAMPLES:

sage: Set(ZZ).cardinality()
+Infinity
sage: Primes().cardinality()
+Infinity
sage: Set(GF(5)).cardinality()
5
sage: Set(GF(5^2, 'a')).cardinality() # needs sage.rings.finite_rings
25

**is_empty()**

Return boolean representing emptiness of the set.

OUTPUT:

True if the set is empty, False if otherwise.

EXAMPLES:

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()  # needs sage.groups
False
sage: Set(ZZ).is_empty()
False

**is_finite()**

Return True if self is finite.

EXAMPLES:

sage: Set(QQ).is_finite()
False

(continues on next page)
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: X = Set(QQ)
sage: X.object()  
Rational Field
sage: X = Primes()
sage: X.object()  
Set of all prime numbers: 2, 3, 5, 7, ...

subsets(size=None)

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

EXAMPLES:

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], [1, 3], [2, 3]]

subsets_lattice()

Return the lattice of subsets ordered by containment.

EXAMPLES:

sage: X = Set([1,2,3])
sage: X.subsets_lattice()  
# needs sage.graphs
Finite lattice containing 8 elements
sage: Y = Set()  
sage: Y.subsets_lattice()  
# needs sage.graphs
Finite lattice containing 1 elements

class sage.sets.set.Set_object_binary(X, Y, op, latex_op, category=None)

Bases: 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
• \texttt{op} – a string describing the binary operation
• \texttt{latex\_op} – a string used for rendering this object in \LaTeX

**EXAMPLES:**

```
sage: X = Set(QQ^2)  # needs sage.modules
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

```python
class sage.sets.set.Set_object_difference(X, Y, category=None)
Bases: Set_object_binary

Formal difference of two sets.

\texttt{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):
  ... Not Implemented Error
```

```python
class sage.sets.set.Set_objectEnumerated(X, category=None)
Bases: Set_object

A finite enumerated set.

\texttt{cardinality()}

Return the cardinality of \texttt{self}.

**EXAMPLES:**

```
sage: Set([1,1]).cardinality()
1
```

\texttt{difference(other)}

Return the set difference \texttt{self} - \texttt{other}.

**EXAMPLES:**
Sets, Release 10.2

```python
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)  # needs sage.rings.real_mpfr
{2.50000000000000}
```

### frozenset()

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

**EXAMPLES:**

```python
sage: # needs sage.rings.finite_rings
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()))  # needs sage.rings.finite_rings
True
sage: type(s)  # needs sage.rings.finite_rings
<...
'frozenset'>
```

### intersection(other)

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

**EXAMPLES:**

```python
sage: X = Set(GF(8,'c'))  # needs sage.rings.finite_rings
sage: Y = Set([GF(8,'c').0, 1, 2, 3])  # needs sage.rings.finite_rings
sage: sorted(X.intersection(Y), key=str)  # needs sage.rings.finite_rings
[1, c]
```

### is_finite()

Return `True` as this is a finite set.

**EXAMPLES:**

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

**issubset(other)**
Return whether `self` is a subset 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.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])
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: # needs sage.rings.finite_rings
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:

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

**set()**

Return the Python set object associated to this set.

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

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
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:

```
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 = Z.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.

---

22 Chapter 1. Set Constructions
EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
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, category=None)

Bases: Set_object_binary

Formal intersection of two sets.

is_finite()

Return whether this set is finite.

EXAMPLES:

```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()  # Not implemented
Traceback (most recent call last):
... NotImplementedError
```

class sage.sets.set.Set_object_symmetric_difference(X, Y, category=None)

Bases: 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: Y.symmetric_difference(X).is_finite()
False
sage: Z.symmetric_difference(Set(ZZ)).is_finite()  # Not implemented
Traceback (most recent call last):
... NotImplementedError
```
class sage.sets.set.Set_object_union(X, Y, category=None)

Bases: 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:

```python
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:

```python
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))  # needs sage.rings.finite_rings
True
sage: has_finite_length(ZZ)
False
```
1.4 Disjoint-set data structure

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

This module defines a class for mutable partitioning of a set, which cannot be used as a key of a dictionary, vertex of a graph etc. For immutable partitioning see $\text{SetPartition}$.

AUTHORS:
- Sébastien Labbé (2009-11-24) - Pickling support
- Sébastien Labbé (2010-01) - Inclusion into sage (github issue #6775).

EXAMPLES:
Disjoint set of integers from $\emptyset$ to $n - 1$:

```python
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}}
sage: s.find(3)
1
sage: list(map(s.find, range(6)))
[0, 1, 2, 1, 2, 1]
```

Disjoint set of hashables objects:

```python
d = DisjointSet('abcde')
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 $\text{arg}$ is in its own set. If $\text{arg}$ is an integer, then the disjoint set returned is made of the integers from $\emptyset$ to $\text{arg} - 1$.

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:

- $\text{find()}$ – Determine which set a particular element is in.
- $\text{union()}$ – 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'}}
```

```python
sage: DisjointSet(range(6))
{{0}, {1}, {2}, {3}, {4}, {5}}
```

```python
sage: DisjointSet(['yi', 45, 'cheval'])
{{'cheval'}, {'yi'}, {45}}
```

class sage.sets.disjoint_set.DisjointSet_class

Bases: 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
```
class sage.sets.disjoint_set.DisjointSet_of_hashables

Bases: 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'}}
```

`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()
sage: sorted(e.items())
[(0, 0), (1, 4), (2, 2), (3, 2), (4, 4)]
sage: WordMorphism(e)  # needs sage.combinat
WordMorphism: 0->0, 1->4, 2->2, 3->2, 4->4
```

`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
```
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:

```
sage: d = DisjointSet(range(5))
sage: d.union(2,3)
sage: d.union(4,1)
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:

```
sage: d = DisjointSet(range(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(sort=True)
 Looped digraph on 5 vertices
```

The result depends on the ordering of the union:
union($e, f$)

Combine the set of $e$ and the set of $f$ into one.
All elements in those two sets will share the same representative that can be gotten using find.

**INPUT:**
- $e$ – element in self
- $f$ – element in self

**EXAMPLES:**

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

class sage.sets.disjoint_set.DisjointSet_of_integers

Bases: DisjointSet_class

Disjoint set of integers from 0 to n-1.

**EXAMPLES:**

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

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:**
Sets, Release 10.2

```python
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) # needs sage.combinat
WordMorphism: 0->0, 1->4, 2->2, 3->2, 4->4

find(i)

Return the representative of the set that i currently belongs to.

INPUT:

• i – element in self

EXAMPLES:

```python
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
```

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.

EXAMPLES:

```python
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])]
```

(continues on next page)
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:**

```python
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(sort=True)
# Looped digraph on 5 vertices
```

The result depends on the ordering of the union:

```python
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(sort=True)
# Looped digraph on 5 vertices
```

**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:**

```python
sage: d = DisjointSet(5)
sage: d
{{0}, {1}, {2}, {3}, {4}}
sage: d.union(0,1)
sage: d
{{0, 1}, {2}, {3}, {4}}
```
1.5 Disjoint union of enumerated sets

AUTHORS:

- Florent Hivert (2010-03): classcall related stuff.
- Florent Hivert (2010-12): fixed facade element construction.

class sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets

Bases: UniqueRepresentation, Parent

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: d.union(2,4)
sage: d
{{0, 1}, {2, 4}, {3}}
sage: d.union(1,4)
sage: d
{{0, 1, 2, 4}, {3}}
sage: d.union(1,5)
Traceback (most recent call last):
...  
ValueError: j(=5) must be between 0 and 4
```
Sets, Release 10.2

```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: # needs sage.combinat
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: # needs sage.combinat
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], [1, 2], [2, 1], [1, 2, 3], [1, 3, 2]]
sage: U4.unrank(18)
[2, 3, 1, 4]
```

1.5. Disjoint union of enumerated sets
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:

```python
sage: # needs sage.combinat
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])
<... 'sage.combinat.permutation.StandardPermutations_n_with_category.element_class'

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.value
[2, 1, 3]
```

We now demonstrate the facade option:

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

```python
sage: el[0]  # needs sage.combinat
Traceback (most recent call last):
...
TypeError: 'sage.structure.element_wrapper.ElementWrapper' object is not subscriptable
```

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.

**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_sets.EnumeratedSets`.

- 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()`.

**cardinality()**

Returns the cardinality of this disjoint union.

**EXAMPLES:**

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

sage: U4.an_element()
[1, 2, 3]
```

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

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

sage: U.cardinality()
10
```
For infinite disjoint unions, this makes the assumption that the result is infinite:

```
sage: U = DisjointUnionEnumeratedSets(
    ....:     Family(NonNegativeIntegers(), Permutations))
sage: U.cardinality()
+Infinity
```

**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: # needs sage.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, ...}
```
```python
sage: from sage.sets.set_from_iterator import set_from_method
sage: class A:
    ....: @set_from_method
    ....: def f(self,n):
    ....:     return xsrange(n)
sage: a = A()
sage: a.f(3)
{0, 1, 2}
sage: a.f(100)
{0, 1, 2, 3, 4, ...}
```

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:

```python
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:

```python
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, ...}
```

**start** = 10

**stop** = 100

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

Bases: Parent

1.6. Enumerated set from iterator 37
A class for enumerated set built from an iterator.

**INPUT:**
- \( f \) – a function that returns an iterable from which the set is built from
- \( \text{args} \) – tuple – arguments to be sent to the function \( f \)
- \( \text{kwds} \) – dictionary – keywords to be sent to the function \( f \)
- \( \text{name} \) – an optional name for the set
- \( \text{category} \) – (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 \( \text{FiniteEnumeratedSets} \), conversely if you know that your iterator will run over and over you should set it as \( \text{InfiniteEnumeratedSets} \).
- \( \text{cache} \) – 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,)); E
# Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, Graph on 7 vertices, ...
```

```python
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,), cache=True, name="Graphs with 8 vertices", category=FiniteEnumeratedSets()); E
```

```python
Graphs with 8 vertices
sage: E.unrank(3)
# Graph on 8 vertices
```

```python
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; e1
lazy list [1, 2, 3, ...]
```

(continues on next page)
sage: E.clear_cache()
sage: E._cache
lazy list [1, 2, 3, ...]
sage: el is E._cache
False

is_parent_of(x)

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

If the set is infinite, only the answer \( \text{True} \) 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)  # needs sage.combinat
sage: E = EnumeratedSetFromIterator(P.__iter__)  # needs sage.combinat
sage: P([5,5,2]) in E  # needs sage.combinat
True
```

unrank()

Returns the element at position \( i \).

EXAMPLES:

```
sage: # needs sage.graphs
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

Bases: Decorator

Decorator for \( \text{EnumeratedSetFromIterator} \).

Name could be string or a function \( \text{(args, kwds)} \rightarrow \text{string} \).

**Warning:** If you are going to use this with the decorator \( \text{cached_function()} \), you must place the \( @\text{cached_function} \) 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(); F
{1, 2, 3, 5, 8, ...}
sage: F.cardinality()
+Infinity
```

A simple example with many options:

```
sage: @set_from_function(name="From %(m)d to %(n)d",
category=FiniteEnumeratedSets())
....: def f(m, n):
return xsrange(m, n + 1)
sage: E = f(3,10); E
From 3 to 10
[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()*:

```
sage: @cached_function
....: @set_from_function(name="Graphs on %(n)d vertices",
category=FiniteEnumeratedSets(), cache=True)
....: 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 \texttt{@cached\_function} must go first:

```
sage: \texttt{@set\_from\_function(name="Graphs on \%}(n)\%d vertices",}
......: \texttt{\ldots category=FiniteEnumeratedSets(), cache=True})
......: \texttt{\ldots \@cached\_function}
......: \texttt{\ldots : def \texttt{Graphs}(n): return graphs(n)}
sage: \texttt{Graphs(10)}
\ldots needs \texttt{sage\_graphs}
Graphs on 10 vertices
sage: \texttt{Graphs(10).unrank(0)}
\ldots needs \texttt{sage\_graphs}
Graph on 10 vertices
sage: \texttt{Graphs(10) is Graphs(10)}
\ldots needs \texttt{sage\_graphs}
False
```

```python
class sage.sets.set_from_iterator.EnumeratedSetFromIterator\_method\_caller(
    inst, f, name=None,
    **options)
```

Bases: \texttt{Decorator}

Caller for decorated method in class.

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 -> string}).
- \texttt{options} – any option accepted by \texttt{EnumeratedSetFromIterator}

```python
class sage.sets.set_from_iterator.EnumeratedSetFromIterator\_method\_decorator(f=None,
    **options)
```

Bases: \texttt{object}

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})-> \texttt{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()
(continues on next page)
```
A more complicated example with a parametrized name:

```python
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); G3
Graphs(3)
sage: G3.cardinality()
#needs sage.graphs
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:

```python
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
aaa
sage: E.list()
[0, 1, 2]
sage: F = d.subset(n=10); F
aaaaaaaaaa
sage: F.list()
[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 EnumeratedSetFromIterator_function_decorator
sage.sets.set_from_iterator.set_from_method
    alias of EnumeratedSetFromIterator_method_decorator
```
1.7 Finite Enumerated Sets

class sage.sets.finite_enumerated_set.FiniteEnumeratedSet(elements)

    Bases: UniqueRepresentation, 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:

```python
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()  # random
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()
cardinality()

first()
Return the first element of the enumeration or raise an EmptySetError if the set is empty.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet('abc')
sage: S.first()
'a'
```

index(x)
Returns the index of x in this finite enumerated set.

EXAMPLES:

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

is_parent_of(x)

last()
Returns the last element of the iteration or raise an EmptySetError if the set is empty.

EXAMPLES:

```
sage: S = FiniteEnumeratedSet([0,'a',1.23,'d'])
sage: S.last()
'd'
```

list()

random_element()
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:
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 $\text{successors}$. 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 Wikipedia article Recursively_enumerable_set.

See documentation of $\text{RecursivelyEnumeratedSet()}$ 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:

$$\text{sage: } \text{succ} = \lambda a: [a+2, a+3]$$

$$\text{sage: } C = \text{RecursivelyEnumeratedSet}([0], \text{succ})$$

$$\text{sage: } C$$

A recursively enumerated set (breadth first search)

Breadth first search:

$$\text{sage: } \text{it} = C.\text{breadth_first_search_iterator()}$$

$$\text{sage: } [\text{next(it)} \text{ for } _\text{in range(10)}]$$

[0, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Depth first search:
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: [next(it_depth) for _ in range(10)]
[(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))
[(-1, 0), (0, -1), (1, 0), (1, 1)]
sage: sorted(C.graded_component(2))
[(-2, 0), (-1, -1), (-1, 1), (0, -2), (1, -1), (2, 0), (1, 1), (2, 0)]
```

1.8.3 Graded structure

Identity permutation as seed and permutohedron_succ as successor function:

```python
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)
```

Depth first search iterator:
Breadth first search iterator:

```python
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]]
```

Elements of given depth iterator:

```python
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]]
```

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

```python
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]]
```

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 = ['', 'a', 'aa', 'aaa', 'aaaa', 'aaaaa']
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:

```python
sage: S = RecursivelyEnumeratedSet([''],
    ...:     lambda x: [x+letter for letter in ['a', 'b', 'c']] if len(x) < 2 else [],
    ...:     structure='forest', enumeration='depth',
    ...:     category=FiniteEnumeratedSets())
sage: S.list()
['', 'a', 'aa', 'ab', 'ac', 'b', 'ba', 'bb', 'bc', 'c', 'ca', 'cb', 'cc']
```

or:

```python
sage: S = RecursivelyEnumeratedSet([''], children,
    ...:     structure='forest', enumeration='depth',
    ...:     category=FiniteEnumeratedSets())
sage: S.list()
['', 'a', 'aa', 'ab', 'ac', 'b', 'ba', 'bb', 'bc', 'c', 'ca', 'cb', 'cc']
```
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 positions. 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 `post_process = f` argument. The convention is that the generated elements are the $s := f(n)$, except when $s$ not `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(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), (3, 1, 8, 6), (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 `successors : U \to 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 successors(x) if and only if x in successors(y)
- **enumeration** – 'depth', 'breadth', 'naive' or None (optional, default: None). The default enumeration for the _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:

```python
sage: f = lambda a: [2*a, 2*a+1]
sage: C = RecursivelyEnumeratedSet([1], f, structure='forest')
sage: C
An enumerated set with a forest structure
sage: it = C.depth_first_search_iterator()
sage: [next(it) for _ in range(7)]
[1, 2, 4, 8, 16, 32, 64]
sage: it = C.breadth_first_search_iterator()
sage: [next(it) for _ in range(7)]
[1, 2, 3, 4, 5, 6, 7]
```

A recursive set given by a symmetric relation:
**Sets, Release 10.2**

```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: def f(a):
....:     return [a + 1, a + I]
sage: C = RecursivelyEnumeratedSet([0], f, structure='graded'); 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.recursively_enumerated_set.RecursivelyEnumeratedSet_forest(roots=None, children=None, post_process=None, algorithm='depth', facade=None, category=None):
    Bases: 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 sage.combinat.backtrack.GenericBacktracker, RecursivelyEnumeratedSet_graded, and RecursivelyEnumeratedSet_symmetric.

    INPUT:
    - roots -- a list (or iterable)
    - children -- a function returning a list (or iterable, or iterator)
    - post_process -- a function defined over the nodes of the forest (default: no post processing)
    - algorithm -- 'depth' or 'breadth' (default: 'depth')
    - category -- a category (default: EnumeratedSets)

    The option post_process allows for customizing the nodes that are actually produced. Furthermore, if f(x) returns None, then x won't be output at all.

    EXAMPLES:
```

**1.8. Recursively enumerated set**
We construct the set of all binary sequences of length at most three, and list them:

```
sage: from sage.sets.recursively_enumerated_set 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]]
```

RecursivelyEnumeratedSet_forest needs to be explicitly told that the set is finite for the following to work:

```
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):

```
sage: from sage.sets.recursively_enumerated_set 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 \(j \leq 1\):

```
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)])
```

With a depth first search, only the elements of the form \((i, 0)\) are generated:

```
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: 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__()`:

```python
sage: from sage.sets.recursively Enumerated_set import RecursivelyEnumeratedSet_forest
sage: class A(UniqueRepresentation, RecursivelyEnumeratedSet_forest):
    ...:     def __init__(self):
    ...:         RecursivelyEnumeratedSet_forest.__init__(self, []for [1, 2, 3, 4, 6, 5, 7, 8, 10, 14, 9, 13, 11, 16, 24, 20, 28, 18, 26, 22, 17, 25, 21, 19, 32, 48, 40, 56, 36]

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:
... 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)))

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 pickleable if and only if the input functions are themselves pickleable. This excludes anonymous or interactively defined functions:

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:

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:

sage: from sage.sets.recursivelyEnumeratedSet import RecursivelyEnumeratedSet_→forest
sage: f = RecursivelyEnumeratedSet_forest([],
.....: lambda 1: [l+[0], l+[1]] if len(l) < 3 else [])
sage: list(f.breadth_first_search_iterator())
children(x)

Return the children of the element x

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

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], l[1]+1)] if l[1] == 0 else [l[0], l[1]]], lambda x: x)
sage: [i for i in I.children((0,0))]  # (1, 0), (0, 1)
sage: [i for i in I.children((1,0))]  # (2, 0), (1, 1)
sage: [i for i in I.children((1,1))]  # (1, 2)
sage: [i for i in I.children((4,1))]  # (4, 2)
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:

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

```
sage: from sage.sets.recursively_enumerated_set import RecursivelyEnumeratedSet_forest
sage: S = RecursivelyEnumeratedSet_forest([(0,0), lambda x: x])
```

1.8. Recursively enumerated set
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, i) \text{ for } i \text{ in range}(1,10)]

sage: def succ(t):
..... list, sum, last = t
..... return [(list + [i], sum + i, i) \text{ for } i \text{ in range}(1, last)]

sage: F = RecursivelyEnumeratedSet(seeds, succ,
..... structure='forest', enumeration='depth')

sage: # needs sage.symbolic

sage: y = var('y')

sage: def map_function(t):
..... li, sum, _ = t
..... return y ^ sum

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:
See also:

sage.parallel.map_reduce

roots()

Return an iterable over the roots of self.

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], l[1]+1)])
```

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

```python
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)])
```

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

class sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic

Bases: Parent

A generic recursively enumerated set.

For more information, see `RecursivelyEnumeratedSet()`.

EXAMPLES:

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

Different structure for the sets:

```python
sage: RecursivelyEnumeratedSet([0], f, structure='None')
A recursively enumerated set (breadth first search)
```

```python
sage: RecursivelyEnumeratedSet([0], f, structure='graded')
A recursively enumerated set with a graded structure (breadth first search)
```

```python
sage: RecursivelyEnumeratedSet([0], f, structure='symmetric')
A recursively enumerated set with a symmetric structure (breadth first search)
```

```python
sage: RecursivelyEnumeratedSet([0], f, structure='forest')
An enumerated set with a forest structure
```

Different default enumeration algorithms:

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

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

```python
sage: 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).

**INPUT:**

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

**EXAMPLES:**

```python
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:**

```python
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]
```

**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:**

```python
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:**

```python
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: C.graded_component(0)
Traceback (most recent call last):
  ... Not Implemented Error: 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:**

```python
sage: f = lambda a: [a+3, a+5]
sage: C = RecursivelyEnumeratedSet([0], f)
sage: it = C.graded_component_iterator()
```

**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:**

```python
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

**Warning:** If the set is infinite, this will loop forever unless `max_depth` is finite.

**EXAMPLES:**

```python
sage: child = lambda i: [(i+3) % 10, (i+8) % 10]

sage: R = RecursivelyEnumeratedSet([0], child)
sage: R.to_digraph()  # needs sage.graphs
Looped multi-digraph on 10 vertices
```

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

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

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

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

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

Digraph of an recursively enumerated set with a graded structure:

```python
sage: f = lambda a: [a+1, a+I]

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

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

class sage.sets.recursivelyEnumeratedSet.RecursivelyEnumeratedSet_graded

**Bases:** `RecursivelyEnumeratedSet_generic`

Generic tool for constructing ideals of a graded 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[0]+1,a[1]), (a[0],a[1]+1)]
```

```python
c = RecursivelyEnumeratedSet([(0,0)], f, structure='graded', max_depth=3)
```

```python
sage: list(c)
[(0, 0),
 (1, 0), (0, 1),
 (2, 0), (1, 1), (0, 2),
 (3, 0), (2, 1), (1, 2), (0, 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 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:**

- **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)]
```

```python
c = RecursivelyEnumeratedSet([(0,0)], f, structure='graded')
```

```python
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**(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:**

1.8. Recursively enumerated set
sage: # needs sage.symbolic
sage: def f(a):
    .....:     return [a + 1, a + I]
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 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:

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

```python
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:**

```python
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*. 

1.8. Recursively enumerated set
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:

```python
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:

```python
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: # needs sage.symbolic
sage: def f(a):
....:     return [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, I + 1, 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, I + 4, 5],
 [6*I, 5*I + 1, 4*I + 2, 3*I + 3, 2*I + 4, I + 5, 6]]
```

`sage.sets.recursively_enumerated_set.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.recursively_enumerated_set import search_forest_iterator
sage: list(search_forest_iterator([], lambda l: [l+[0], l+[1]]))
```

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

\[
\begin{align*}
\text{sage: } & \text{list(search_forest_iterator([], lambda l: [l+[0], l+[1]], algorithm='breadth'))} \\
& \text{[[], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1], [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]}
\end{align*}
\]

This allows for iterating through trees of infinite depth:

\[
\begin{align*}
\text{sage: } & \text{it = search_forest_iterator([], lambda l: [l+[0], l+[1]], algorithm='breadth')} \\
\text{sage: } & \text{[ next(it) for i in range(16) ]}
\end{align*}
\]

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:

\[
\begin{align*}
\text{sage: } & \text{list(search_forest_iterator([], lambda l: [1+i for i in range(3) if i not in l], algorithm='breadth'))} \\
& \text{[[], [0], [1], [0, 0], [0, 1], [1, 0], [1, 1], [0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]}
\end{align*}
\]

1.9 Subsets of a Universe Defined by Predicates

```python
class sage.sets.condition_set.ConditionSet(universe, *predicates, names=None, category=None):
    Bases: Set_generic, Set_base, Set_boolean_operators, Set_add_sub_operators, UniqueRepresentation
    
    Set of elements of a universe that satisfy given predicates
    
    INPUT:
    
    - universe -- a set
    - *predicates -- callables
```
• vars or names – (default: inferred from predicates if any predicate is an element of a 
CallableSymbolicExpressionRing_class) variables or names of variables

• category – (default: inferred from universe) a category

EXAMPLES:

```
sage: Evens = ConditionSet(ZZ, is_even); Evens
{x \in \text{Integer Ring} : <function is_even at 0x...>(x) }
sage: 2 in Evens
True
sage: 3 in Evens
False
sage: 2.0 in Evens
True
```

```
sage: Odds = ConditionSet(ZZ, is_odd); Odds
{x \in \text{Integer Ring} : <function is_odd at 0x...>(x) }
sage: EvensAndOdds = Evens | Odds; EvensAndOdds
Set-theoretic union of 
{x \in \text{Integer Ring} : <function is_even at 0x...>(x) } and 
{x \in \text{Integer Ring} : <function is_odd at 0x...>(x) }
sage: 5 in EvensAndOdds
True
sage: 7/2 in EvensAndOdds
False
```

```
sage: var('y')
# needs sage.symbolic
y
sage: SmallOdds = ConditionSet(ZZ, is_odd, abs(y) <= 11, vars=[y]); SmallOdds  
# needs sage.symbolic
{ y \in \text{Integer Ring} : abs(y) <= 11, <function is_odd at 0x...>(y) }
```

```
sage: # needs sage.geometry.polyhedron
sage: P = polytopes.cube(); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: P.rename("P")
sage: P_inter_B = ConditionSet(P, lambda x: x.norm() < 1.2); P_inter_B
{x \in P : <function <lambda> at 0x...>(x) }
sage: vector([1, 0, 0]) in P_inter_B
True
sage: vector([1, 1, 1]) in P_inter_B  
# needs sage.symbolic
False
```

```
sage: # needs sage.symbolic
sage: predicate(x, y, z) = sqrt(x^2 + y^2 + z^2) < 1.2; predicate
(x, y, z) |--> sqrt(x^2 + y^2 + z^2) < 1.20000000000000
sage: P_inter_B_again = ConditionSet(P, predicate); P_inter_B_again
# needs sage.geometry.polyhedron
{ (x, y, z) \in P : sqrt(x^2 + y^2 + z^2) < 1.20000000000000 }
sage: vector([1, 0, 0]) in P_inter_B_again  
# needs sage.geometry.polyhedron
```

(continues on next page)
True
\[
sage: \text{vector([1, 1, 1]) in P\textunderscore inter\textunderscore B\textunderscore again} \quad \#.
\]

Iterating over subsets determined by predicates:

\[
sage: \text{Odds} = \text{ConditionSet}(\text{ZZ}, \text{is\_odd}); \text{Odds}
\{ x \in \text{Integer Ring} : <\text{function is\_odd at 0x...}>\!(x) \}
\]

\[
sage: \text{list(Odds.iterator\_range(stop=6))}
[1, -1, 3, -3, 5, -5]
\]

\[
sage: R = \text{IntegerModRing}(8)
\]

\[
sage: \text{R\_primes} = \text{ConditionSet}(\text{R}, \text{is\_prime}); \text{R\_primes}
\{ x \in \text{Ring of integers modulo 8} : <\text{function is\_prime at 0x...}>\!(x) \}
\]

\[
sage: \text{R\_primes.is\_finite()}
True
\]

\[
sage: \text{list(R\_primes)}
[2, 6]
\]

Using \texttt{ConditionSet} without predicates provides a way of attaching variable names to a set:

\[
sage: \text{Z3} = \text{ConditionSet}(\text{ZZ}^3, \text{vars}=['x', 'y', 'z']); \text{Z3}
\{ (x, y, z) \in \text{Ambient free module of rank 3 over the principal ideal domain Integer Ring} \}
\]

\[
sage: \text{Z3.variable\_names()}
('x', 'y', 'z')
\]

\[
sage: \text{Z3.arguments()}
(x, y, z)
\]

\[
sage: \text{Q4.<a, b, c, d> = ConditionSet(QQ^4); Q4}
\{ (a, b, c, d) \in \text{Vector space of dimension 4 over Rational Field} \}
\]

\[
sage: \text{Q4.variable\_names()}
('a', 'b', 'c', 'd')
\]

\[
sage: \text{Q4.arguments()}
(a, b, c, d)
\]

\textbf{ambient()}

Return the universe of self.

\textbf{EXAMPLES:}

\[
sage: \text{Evens} = \text{ConditionSet}(\text{ZZ}, \text{is\_even}); \text{Evens}
\{ x \in \text{Integer Ring} : <\text{function is\_even at 0x...}>\!(x) \}
\]

\[
sage: \text{Evens.ambient()}
\text{Integer Ring}
\]
arguments()

Return the variables of self as elements of the symbolic ring.

EXAMPLES:

```
sage: Odds = ConditionSet(ZZ, is_odd); Odds
{x ∈ Integer Ring : <function is_odd at 0x...>(x) }
sage: args = Odds.arguments(); args

# needs sage.symbolic
(x, )
sage: args[0].parent()

# needs sage.symbolic
Symbolic Ring
```

intersection(X)

Return the intersection of self and X.

EXAMPLES:

```
sage: # needs sage.modules sage.symbolic
sage: in_small_oblong(x, y) = x^2 + 3 * y^2 <= 42
sage: SmallOblongUniverse = ConditionSet(QQ^2, in_small_oblong)
sage: SmallOblongUniverse

{x ∈ Vector space of dimension 2 over Rational Field : x^2 + 3*y^2 <= 42 }
sage: parity_check(x, y) = abs(sin(pi/2*(x + y))) < 1/1000
sage: EvenUniverse = ConditionSet(ZZ^2, parity_check); EvenUniverse

{x ∈ Ambient free module of rank 2 over the principal ideal domain Integer Ring : abs(sin(1/2*pi*x + 1/2*pi*y)) < (1/1000) }
sage: SmallOblongUniverse & EvenUniverse

{x ∈ Vector space of dimension 2 over Rational Field : x^2 + 3*y^2 <= 42 }
sage: SmallMirrorUniverse = ConditionSet(QQ^2, in_small_oblong, ...

# needs sage.symbolic
....:

vars=(y, x))
sage: SmallMirrorUniverse

{ (y, x) ∈ Vector space of dimension 2 over Rational Field : 3*x^2 + y^2 <= 42 }
sage: SmallOblongUniverse & SmallMirrorUniverse

{ (x, y) ∈ Vector space of dimension 2 over Rational Field : x^2 + 3*y^2 <= 42 }
sage: SmallMirrorUniverse & SmallOblongUniverse

{ (y, x) ∈ Vector space of dimension 2 over Rational Field : 3*x^2 + y^2 <= 42 }
```

Combining two ConditionSet`s with different formal variables works correctly. The formal variables of the intersection are taken from `\`self`: 

```
sage: # needs sage.modules sage.symbolic
sage: SmallMirrorUniverse = ConditionSet(QQ^2, in_small_oblong, ...

# needs sage.symbolic
....:

vars=(y, x))
sage: SmallMirrorUniverse

{ (y, x) ∈ Vector space of dimension 2 over Rational Field : 3*x^2 + y^2 <= 42 }
sage: SmallOblongUniverse & SmallMirrorUniverse

{ (x, y) ∈ Vector space of dimension 2 over Rational Field : x^2 + 3*y^2 <= 42 }
sage: SmallMirrorUniverse & SmallOblongUniverse

{ (y, x) ∈ Vector space of dimension 2 over Rational Field : 3*x^2 + y^2 <= 42 }
```

Chapter 1. Set Constructions
1.10 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
class sage.sets.finite_set_maps.FiniteSetEndoMaps_N(n, action, category=None)
    Bases: 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.

    Element
    alias of `FiniteSetEndoMap_N`

    an_element()
    Returns a map in self
    EXAMPLES:
    ```sage```
    sage: M = FiniteSetMaps(4)
    sage: M.an_element()
    [3, 2, 1, 0]
    ```

    one()
    EXAMPLES:
    ```sage```
    sage: M = FiniteSetMaps(4)
    sage: M.one()
    [0, 1, 2, 3]
    ```
```
```python
class sage.sets.finite_set_maps.FiniteSetEndoMaps_Set(domain, action, category=None)
    Bases: FiniteSetMaps_Set, 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 `FiniteSetEndoMap_Set`
```
class sage.sets.finite_set_maps.FiniteSetMaps

    Bases: UniqueRepresentation, 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 \( \mathcal{M} \) 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
map: a -> 5, b -> 3
map: a -> 5, b -> 4
map: a -> 5, b -> 5
```

Elements can be constructed from functions and dictionaries:

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

(continues on next page)
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:**

```python
sage: FiniteSetMaps(4, 3).cardinality()
81
```

**class** `sage.sets.finite_set_maps.FiniteSetMaps_MN(m, n, category=None)`

**Bases:** `FiniteSetMaps`

The set of all maps from $\{1, 2, \ldots, m\}$ to $\{1, 2, \ldots, n\}$. 

**1.10. Maps between finite sets**
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 FiniteSetMap_MN*

**an_element()**

Returns a map in *self*

**EXAMPLES:**

```python
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.

```python
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:**

```python
sage: FiniteSetMaps(3,2).codomain()
{0, 1}
```

**domain()**

The domain of *self*

**EXAMPLES:**

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

**class** `sage.sets.finite_set_maps.FiniteSetMaps_Set (domain, codomain, category=None)`

Bases: `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 \texttt{FiniteSets()}.  
- **codomain** – an object in the category \texttt{FiniteSets()}.  
- **category** – the category in which the sets of maps is constructed. It must be a sub-category of \texttt{EnumeratedSets()}.\texttt{Finite()} which is the default value.

**Element**

alias of \texttt{FiniteSetMap_Set}

codomain()

The codomain of self

**EXAMPLES:**

\begin{verbatim}sage: FiniteSetMaps(["a", "b"], [3, 4, 5]).codomain()
{3, 4, 5}
\end{verbatim}

domain()

The domain of self

**EXAMPLES:**

\begin{verbatim}sage: FiniteSetMaps(["a", "b"], [3, 4, 5]).domain()
{"a", "b"}
\end{verbatim}

from_dict(d)

Create a map from a dictionary

**EXAMPLES:**

\begin{verbatim}sage: M = FiniteSetMaps(["a", "b"], [3, 4, 5])
sage: M.from_dict({"a": 4, "b": 3})
map: a -> 4, b -> 3
\end{verbatim}

1.11 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 \texttt{FiniteSetMaps}.

**EXAMPLES:**

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

\begin{verbatim}sage: M = FiniteSetMaps(["a", "b"], [3, 4, 5])
\end{verbatim}

The map can then be constructed either from a function:

\begin{verbatim}sage: f1 = M(lambd\ a: ord(c)-94); f1
map: a -> 3, b -> 4
\end{verbatim}

or from a dictionary:

\begin{verbatim}sage: f2 = M.from_dict({"a":3, "b":4}); f2
map: a -> 3, b -> 4
\end{verbatim}
The two created maps are equal:

```python
sage: f1 == f2
True
```

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:

```python
sage: list(f2)
[0, 1]
```

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

```python
sage: f3 = M._from_list_([0, 1]); f3
map: a -> 3, b -> 4
sage: f1 == f3
True
```

AUTHORS:
- Florent Hivert

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

EXAMPLES:

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

domain()
Returns the domain of self

EXAMPLES:

```python
sage: 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:

```python
sage: FiniteSetMaps(4, 3)([1, 0, 2, 1]).fibers()
{0: {1}, 1: {0, 3}, 2: {2}}
sage: F = FiniteSetMaps(["a", "b", "c"])
sage: 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:

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

image_set()
Returns the image set of self

EXAMPLES:

```python
sage: FiniteSetMaps(4, 3)([1, 0, 2, 1]).image_set()
{0, 1, 2}
sage: FiniteSetMaps(4, 3)([1, 0, 0, 1]).image_set()
{0, 1}
```
items()
The items of self
Return the list of the ordered pairs \((x, \text{self}(x))\)

EXAMPLES:

```
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: 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, parent, lst)

Creates a FiniteSetMap from a list

Warning: no check is performed!

def getimage(i)

Returns the image of i by self

INPUT:

• i – an int

EXAMPLES:

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

image_set()

Returns the image set of self

EXAMPLES:

sage: F = FiniteSetMaps(["a", "b", "c"])
sage: 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, 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:

```python
g = FiniteMaps(['a', 'b', 'c', 'd'], ['u', 'v', 'w'])
g = g(lambda x: 'v')
g.setimage('a', 'w')
g
map: a -> w, b -> v, c -> v, d -> v
g = g.clone() as g:
....
g.setimage('a', 'u')
....
g.setimage('c', 'w')
g
map: a -> u, b -> v, c -> w, d -> v
```

`sage.sets.finite_set_map_cy.FiniteSetMap_Set_from_dict(t, parent, d)`

Creates a `FiniteSetMap` from a dictionary

**Warning:** no check is performed!

`sage.sets.finite_set_map_cy.FiniteSetMap_Set_from_list(t, parent, lst)`

Creates a `FiniteSetMap` from a list

**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 $domain$

INPUT:
• $f$ – a function or callable
• $domain$ – a finite iterable

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

EXAMPLES:

```python
from sage.sets.finite_set_map_cy import fibers, fibers_args
fibers(lambda x: 1, [])
{}  # Example 1
fibers(lambda x: x**2, [-1, 2, -3, 1, 3, 4])
{1: {1, -1}, 4: {2}, 9: {3, -3}, 16: {4}}
fibers(lambda x: x, [-1, 2, -3, 1, 3, 4])
{1: {1, 2, 3, 4, -3, -1}}
fibers(lambda x: 1, [1, 1, 1])
{1: {1}}
```
See also:

fibers_args() if one needs to pass extra arguments to \( f \).

```python
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 fibers() except that one can pass extra argument for \( f \) (with a small overhead)

EXAMPLES:

```python
sage: from sage.sets.finite_set_map_cy import fibers_args
sage: fibers_args(operator.pow, [-1, 2, -3, 1, 3, 4], 2)
{1: {1, -1}, 4: {2}, 9: {3, -3}, 16: {4}}
```

### 1.12 Totally Ordered Finite Sets

**AUTHORS:**

- Stepan Starosta (2012): Initial version

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

Bases: `FiniteEnumeratedSet`

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:**

```python
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:

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

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

```python
sage: T1 = TotallyOrderedFiniteSet([3,2,5,1])
sage: T1(3) < T1(1)
False
```

(continues on next page)
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:

```
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:

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

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

```
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:

```
sage: TotallyOrderedFiniteSet([1, 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:**

```
sage: T = TotallyOrderedFiniteSet([1,3,2], facade=False)
sage: T1, T3, T2 = T.list()
sage: T.le(T1,T3)
True
sage: T.le(T3,T2)
True
```

```python
class sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSetElement(
    parent, data)

Bases: Element

Element of a finite totally ordered set.

**EXAMPLES:**

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

---

**1.13 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:**

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

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

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

```python
class sage.sets.pythonclass.Set_PythonType_class

Bases: Set_generic

The set of Python objects of a given class.
```
The elements of this set are not instances of \texttt{Element}; they are instances of the given class.

\textbf{INPUT:}

- \texttt{typ} – a Python (new-style) class

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{cardinality}()

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{object}()

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.sets.pythonclass import Set_PythonType
sage: Set_PythonType(tuple).object()
<... 'tuple'>
\end{verbatim}
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.

```python
class sage.sets.integer_range.IntegerRange
    Bases: UniqueRepresentation, 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:
```
sage: list(IntegerRange(5))
[0, 1, 2, 3, 4]
sage: list(IntegerRange(2,5))
[2, 3, 4]
sage: I = IntegerRange(2,100,5); I
```
(continues on next page)
\{2, 7, \ldots, 97\}
sage: list(I)
[2, 7, 12, 17, 22, 27, 32, 37, 42, 47, 52, 57, 62, 67, 72, 77, 82, 87, 92, 97]
sage: I.category()
Category of facade finite enumerated sets
sage: I[1].parent()
Integer Ring

When begin and end are both finite, \texttt{IntegerRange(begin, end, step)} is the set whose list of elements is equivalent to the python construction \texttt{range(begin, end, step)}:

\begin{verbatim}
sage: list(IntegerRange(4,105,3)) == list(range(4,105,3))
True
sage: list(IntegerRange(-54,13,12)) == list(range(-54,13,12))
True
\end{verbatim}

Except for the type of the numbers:

\begin{verbatim}
sage: type(IntegerRange(-54,13,12)[0]), type(list(range(-54,13,12))[0])
(<<... 'sage.rings.integer.Integer'>, <... 'int'>)
\end{verbatim}

When begin is finite and end is +Infinity, \texttt{self} is the infinite arithmetic progression starting from the begin by step step:

\begin{verbatim}
sage: I = IntegerRange(54,Infinity,3); I
{54, 57, \ldots}
sage: I.category()
Category of facade infinite enumerated sets
sage: p = iter(I)
sage: (next(p), next(p), next(p), next(p), next(p), next(p))
(54, 57, 60, 63, 66, 69)
\end{verbatim}

When begin is finite and end is -Infinity, \texttt{self} is the infinite arithmetic progression starting from the begin by step step:

\begin{verbatim}
sage: I = IntegerRange(54,-Infinity,-3); I
{54, 51, \ldots}
sage: I.category()
Category of facade infinite enumerated sets
sage: p = iter(I)
sage: (next(p), next(p), next(p), next(p), next(p), next(p))
(54, 51, 48, 45, 42, 39)
\end{verbatim}

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

\begin{verbatim}
sage: I = IntegerRange(-Infinity,Infinity,37,-12); I
Integer progression containing -12 with increment 37 and bounded with -Infinity and+
\rightarrow+Infinity
sage: I.category()
Category of facade infinite enumerated sets
sage: -12 in I
True
sage: -15 in I
False
\end{verbatim}
\begin{verbatim}
sage: p = iter(I)
sage: (next(p), next(p), next(p), next(p), next(p), next(p), next(p), next(p))
(-12, 25, -49, 62, -86, 99, -123, 136)
\end{verbatim}

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:

\begin{verbatim}
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]
\end{verbatim}

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:

\begin{verbatim}
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]
\end{verbatim}

\textbf{Note:} The input is normalized so that:

\begin{verbatim}
sage: IntegerRange(1, 6, 2) is IntegerRange(1, 7, 2)
True
sage: IntegerRange(1, 8, 3) is IntegerRange(1, 10, 3)
True
\end{verbatim}

\begin{verbatim}
element_class
    alias of Integer
class sage.sets.integer_range.IntegerRangeEmpty(elements)
    Bases: IntegerRange, FiniteEnumeratedSet
    A singleton class for empty integer ranges
    See IntegerRange for more details.
class sage.sets.integer_range.IntegerRangeFinite(begin, end, step=1)
    Bases: IntegerRange
\end{verbatim}

\section{2.1. Integer Range}
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.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
```

(continues on next page)
True
sage: l.reverse()
sage: [I[i] for i in range(-1,-I.cardinality()-1,-1)] == l
True

class sage.sets.integer_range.IntegerRangeFromMiddle

Bases: 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

Bases: 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

class sage.sets.positive_integers.PositiveIntegers
    Bases: 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: 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: PositiveIntegers().an_element()
42

2.3 Non Negative Integers

class sage.sets.non_negative_integers.NonNegativeIntegers(category=None)
    Bases: UniqueRepresentation, Parent
    The enumerated set of non negative integers.
    This class implements the set of non negative integers, as an enumerated set (see InfiniteEnumeratedSets).
    EXAMPLES:
sage: NN = NonNegativeIntegers()
sage: NN
Non negative integers
sage: NN.cardinality()
+Infinity
sage: TestSuite(NN).run()
sage: NN.list()
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 Integer

**an_element()**

EXAMPLES:

sage: NonNegativeIntegers().an_element()
42

**from_integer**

alias of Integer
2.4 The set of prime numbers

AUTHORS:
- William Stein (2005): original version

class sage.sets.primes.Primes(proof)
Bases: Set_generic, UniqueRepresentation
The set of prime numbers.

EXAMPLES:

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

We show various operations on the set of prime numbers:

```python
sage: P.cardinality()
+Infinity
sage: R = Primes()
sage: P == R
True
sage: 5 in P
True
sage: 100 in P
False
```

```python
sage: len(P)
Traceback (most recent call last):
... Not Implemented: 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: P = Primes()
sage: P.next(5) # needs sage.libs.pari
7
```

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

EXAMPLES:

```
sage: P = Primes()
sage: P.unrank(0) # needs sage.libs.pari
2
sage: P.unrank(5) # needs sage.libs.pari
13
sage: P.unrank(42) # needs sage.libs.pari
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: RealSet(0,1)
(0, 1)
sage: RealSet((0,1), [2,3])
(0, 1) \cup [2, 3]
sage: RealSet((1,3), (0,2))
(0, 3)
sage: RealSet(-oo, oo)
(-oo, +oo)
```

Brackets must be balanced in Python, so the naive notation for half-open intervals does not work:
Instead, you can use the following construction functions:

```python
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)
```

The lower and upper endpoints will be sorted if necessary:

```python
sage: RealSet.interval(1, 0, lower_closed=True, upper_closed=False)
[0, 1)
```

Relations containing symbols and numeric values or constants:

```python
sage: # needs sage.symbolic
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:

```python
sage: RealSet(x >= 0, x < 1) # needs sage.symbolic
(-oo, +oo)
sage: RealSet(x >= 0, x > 1) # needs sage.symbolic
[0, +oo)
sage: RealSet(x >= 0, x > -1) # needs sage.symbolic
(-1, +oo)
```

AUTHORS:
Sets, Release 10.2

- 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
- Yueqi Li, Yuan Zhou (2022-07-31): Rewrite RealSet. Adapt faster operations by scan-line (merging) techniques from the code by Matthias Köppe et al., at https://github.com/mkoeppe/cutgeneratingfunctionology/blob/master/cutgeneratingfunctionology/igp/intervals.py

```python
class sage.sets.real_set.InternalRealInterval(lower, lower_closed, upper, upper_closed, check=True)

Bases: UniqueRepresentation, Parent

A real interval.

You are not supposed to create InternalRealInterval 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

boundary_points()

Generate the boundary points of self

EXAMPLES:

```
sage: list(RealSet.open_closed(-oo, 1)[0].boundary_points())
[1]
sage: list(RealSet.open(1, 2)[0].boundary_points())
[1, 2]
```

closure()

Return the closure

OUTPUT:

The closure as a new InternalRealInterval

EXAMPLES:

```
sage: RealSet.open(0, 1)[0].closure()
[0, 1]
sage: RealSet.open(-oo, 1)[0].closure()
(-oo, 1]
sage: RealSet.open(0, oo)[0].closure()
[0, +oo]
```

contains(x)

Return whether x is contained in the interval

INPUT:
• $x$ – a real number.

OUTPUT:

Boolean.

EXAMPLES:

```python
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 `InternalRealInterval`.

EXAMPLES:

```python
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 `LazyFieldElement`

interior()

Return the interior

OUTPUT:

The interior as a new `InternalRealInterval`

EXAMPLES:
### intersection(other)

Return the intersection of the two intervals

**INPUT:**

- `other` – a `InternalRealInterval`

**OUTPUT:**

The intersection as a new `InternalRealInterval`

**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())
```

(continues on next page)
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:

```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
```

lower_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
```

upper()
Return the upper bound

OUTPUT:
The upper bound as it was originally specified.

EXAMPLES:

```python
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:

```
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:

```
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, normalized=True)

Bases: UniqueRepresentation, Parent, Set_base, Set_boolean_operators, Set_add_sub_operators

A subset of the real line, a finite union of intervals

INPUT:

- *args – arguments defining a real set. Possibilities are either:
  - two extended real numbers \(a, b\), to construct the open interval \((a, b)\), or
  - a list/tuple/iterable of (not necessarily disjoint) intervals or real sets, whose union is taken. The individual intervals can be specified by either
    - a tuple \((a, b)\) of two extended real numbers (constructing an open interval),
    - a list \([a, b]\) of two real numbers (constructing a closed interval),
    - an InternalRealInterval,
∗ an OpenInterval.

- **structure**  – (default: None) if None, construct the real set as an instance of RealSet; if "differentiable", construct it as a subset of an instance of RealLine, representing the differentiable manifold \( \mathbb{R} \).

- **ambient**  – (default: None) an instance of RealLine; construct a subset of it. Using this keyword implies structure='differentiable'.

- **names** or **coordinate** – coordinate symbol for the canonical chart; see RealLine. Using these keywords implies structure='differentiable'.

- **name**, **latex_name**, **start_index** – see RealLine.

- **normalized** – (default: None) if True, the input is already normalized, i.e., *args are the connected components (type InternalRealInterval) of the real set in ascending order; no other keyword is provided.

There are also specialized constructors for various types of intervals:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealSet.open()</td>
<td>((a, b))</td>
</tr>
<tr>
<td>RealSet.closed()</td>
<td>([a, b])</td>
</tr>
<tr>
<td>RealSet.point()</td>
<td>{a}</td>
</tr>
<tr>
<td>RealSet.open_closed()</td>
<td>((a, b])</td>
</tr>
<tr>
<td>RealSet.closed_open()</td>
<td>([a, b)]</td>
</tr>
<tr>
<td>RealSet.unbounded_below_closed()</td>
<td>((−\infty, b])</td>
</tr>
<tr>
<td>RealSet.unbounded_below_open()</td>
<td>((−\infty, b))</td>
</tr>
<tr>
<td>RealSet.unbounded_above_closed()</td>
<td>([a, +\infty))</td>
</tr>
<tr>
<td>RealSet.unbounded_above_open()</td>
<td>((a, +\infty))</td>
</tr>
<tr>
<td>RealSet.real_line()</td>
<td>((−\infty, +\infty))</td>
</tr>
<tr>
<td>RealSet.interval()</td>
<td>any</td>
</tr>
</tbody>
</table>

**EXAMPLES:**

```python
sage: RealSet(0, 1)  # open set from two numbers
(0, 1)
sage: RealSet(1, 0)  # the two numbers will be sorted
(0, 1)
sage: s1 = RealSet((1,2)); s1  # tuple of two numbers = open set
(1, 2)
sage: s2 = RealSet([3,4]); s2  # list of two numbers = closed set
[3, 4]
sage: i1, i2 = s1[0], s2[0]
sage: RealSet(i2, i1)  # union of intervals
(1, 2) \cup [3, 4]
sage: RealSet((-oo, 0), x > 6, i1, RealSet.point(5)),  #...
˓needs sage.symbolic
    ....:    RealSet.closed_open(4, 3))
(-\infty, 0) \cup (1, 2) \cup [3, 4] \cup \{5\} \cup (6, +\infty)
```

Initialization from manifold objects:

```python
sage: # needs sage.symbolic
sage: R = manifolds.RealLine(); R
Real number line \( \mathbb{R} \)
sage: RealSet(R)
```

(continues on next page)
Real sets belong to a subcategory of topological spaces:

```
sage: RealSet().category()
Join of
  Category of finite sets and
  Category of subobjects of sets and
  Category of connected topological spaces

sage: RealSet.point(1).category()
Join of
  Category of finite sets and
  Category of subobjects of sets and
  Category of connected topological spaces

sage: RealSet([1, 2]).category()
Join of
  Category of infinite sets and
  Category of compact topological spaces and
  Category of subobjects of sets and
  Category of connected topological spaces

sage: RealSet((1, 2), (3, 4)).category()
Join of
  Category of infinite sets and
  Category of subobjects of sets and
  Category of topological spaces
```

Constructing real sets as manifolds or manifold subsets by passing `structure='differentiable'`:

```
sage: # needs sage.symbolic
sage: RealSet(-oo, oo, structure='differentiable')
Real number line $\mathbb{R}$

sage: RealSet([0, 1], structure='differentiable')
Subset $[0, 1]$ of the Real number line $\mathbb{R}$

sage: _.category()
Category of subobjects of sets
```

This is implied when a coordinate name is given using the keywords `coordinate` or `names`:

```sage
sage: RealSet(0, 1, coordinate='\lambda')
Open subset (0, 1) of the Real number line \mathbb{R}
```

It is also implied by assigning a coordinate name using generator notation:

```sage
sage: R_xi.<\xi> = RealSet.real_line(); R_xi
Real number line \mathbb{R}
```

With the keyword `ambient`, we can construct a subset of a previously constructed manifold:

```sage
sage: P_xi = RealSet(0, oo, ambient=R_xi); P_xi
Open subset (0, +oo) of the Real number line \mathbb{R}
```

2.5. Subsets of the Real Line
static are_pairwise_disjoint(*real_set_collection)

Test whether the real sets are pairwise disjoint

INPUT:

• *real_set_collection – a list/tuple/iterable of RealSet or data that defines one.

OUTPUT:

Boolean.

See also:

is_disjoint()

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

boundary()

Return the topological boundary of self as a new RealSet.

EXAMPLES:

```
sage: RealSet(-oo, oo).boundary()
{}
sage: RealSet().boundary()
{}
sage: RealSet.point(2).boundary()
{2}
sage: RealSet([1, 2], (3, 4)).boundary()
{1} ∪ {2} ∪ {3} ∪ {4}
sage: RealSet((1, 2), (2, 3)).boundary()
{1} ∪ {2} ∪ {3}
```

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()
3
sage: RealSet(0,3).cardinality()
+Infinity
```
static closed(lower, upper, **kwds)

Construct a closed interval

INPUT:

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

OUTPUT:

A new RealSet.

EXAMPLES:

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

static closed_open(lower, upper, **kwds)

Construct an half-open interval

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

closure()

Return the topological closure of self as a new RealSet.

EXAMPLES:

sage: RealSet(-oo, oo).closure()
(-oo, +oo)
sage: RealSet((1, 2), (2, 3)).closure()
[1, 3]
sage: RealSet().closure()
{

complement()

Return the complement

OUTPUT:

The set-theoretic complement as a new RealSet.

EXAMPLES:

sage: RealSet(0,1).complement() (-oo, 0] \cup [1, +oo)
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1

(continues on next page)
contains\(x\)

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

INPUT:

\* \(x\) – a real number.

OUTPUT:

Boolean.

EXAMPLES:

```python
sage: s = RealSet(0,2) + RealSet.unbounded_above_closed(10); s
(0, 2) \cup [10, +\infty)

sage: s.contains(1)
True

sage: s.contains(0)
False

sage: s.contains(10.0)
True

sage: 10 in s          # syntactic sugar
True

sage: s.contains(+\infty)
False

sage: RealSet().contains(1)
False
```

**static convex_hull\(*\text{real_set_collection}\)**

Return the convex hull of real sets.

INPUT:

\* \(*\text{real_set_collection}\) – a list/tuple/iterable of \(\text{RealSet}\) or data that defines one.

OUTPUT:

The convex hull as a new \(\text{RealSet}\).

EXAMPLES:

```python
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1 # unbounded
(0, 2) \cup [10, +\infty)

sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s2
(-\infty, -10] \cup (1, 3)

sage: s3 = RealSet((0,2), RealSet.point(8)); s3
(0, 2) \cup \{8\}
```
\texttt{sage: } s4 = \texttt{RealSet(); } s4 \quad \# \textit{empty set} \\
{} \\
\texttt{sage: } \texttt{RealSet.convex_hull(s1)} \\
(0, \infty) \\
\texttt{sage: } \texttt{RealSet.convex_hull(s2)} \\
(-\infty, 3) \\
\texttt{sage: } \texttt{RealSet.convex_hull(s3)} \\
(0, 8] \\
\texttt{sage: } \texttt{RealSet.convex_hull(s4)} \\
{} \\
\texttt{sage: } \texttt{RealSet.convex_hull(s1, s2)} \\
(-\infty, +\infty) \\
\texttt{sage: } \texttt{RealSet.convex_hull(s2, s3)} \\
(-\infty, 8] \\
\texttt{sage: } \texttt{RealSet.convex_hull(s2, s3, s4)} \\
(-\infty, 8] \\
\texttt{difference(*other)} \\
\text{Return self with other subtracted} \\
\text{INPUT:} \\
\quad \bullet \text{ other – a } \texttt{RealSet} \text{ or data that defines one.} \\
\text{OUTPUT:} \\
\text{The set-theoretic difference of self with other removed as a new } \texttt{RealSet}. \\
\text{EXAMPLES:} \\
\texttt{sage: } s1 = \texttt{RealSet(0,2) + RealSet.unbounded_above_closed(10); } s1 \\
(0, 2) \cup [10, +\infty) \\
\texttt{sage: } s2 = \texttt{RealSet(1,3) + RealSet.unbounded_below_closed(-10); } s2 \\
(-\infty, -10] \cup (1, 3) \\
\texttt{sage: } s1.\texttt{difference(s2)} \\
(0, 1] \cup [10, +\infty) \\
\texttt{sage: } s1 - s2 \quad \# \textit{syntactic sugar} \\
(0, 1] \cup [10, +\infty) \\
\texttt{sage: } s2.\texttt{difference(s1)} \\
(-\infty, -10] \cup [2, 3) \\
\texttt{sage: } s2 - s1 \quad \# \textit{syntactic sugar} \\
(-\infty, -10] \cup [2, 3) \\
\texttt{sage: } s1.\texttt{difference(1,11)} \\
(0, 1] \cup [11, +\infty) \\
\texttt{get_interval(i)} \\
\text{Return the i-th connected component.} \\
\text{Note that the intervals representing the real set are always normalized, i.e., they are sorted, disjoint and not connected.} \\
\text{INPUT:} \\
\quad \bullet \text{ i – integer.} \\
\text{OUTPUT:}
The $i$-th connected component as a \texttt{InternalRealInterval}.

**EXAMPLES:**

```python
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
```

\texttt{inf()}

Return the infimum

**OUTPUT:**

A real number or infinity.

**EXAMPLES:**

```python
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1
(0, 2) ∪ [10, +∞)
sage: s1.inf()
0
sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s2
(-∞, -10] ∪ (1, 3)
sage: s2.inf()
-∞
```

\texttt{interior()}

Return the topological interior of \texttt{self} as a new \texttt{RealSet}.

**EXAMPLES:**

```python
sage: RealSet(-oo, oo).interior()
(-oo, +oo)
sage: RealSet().interior()
{}
sage: RealSet.point(2).interior()
{}
sage: RealSet([1, 2], (3, 4)).interior()
(1, 2) ∪ (3, 4)
```

\texttt{intersection(*real\_set\_collection)}

Return the intersection of real sets

**INPUT:**

- *real\_set\_collection* – a list/tuple/iterable of \texttt{RealSet} or data that defines one.

**OUTPUT:**

The set-theoretic intersection as a new \texttt{RealSet}.

**EXAMPLES:**
sage: s1 = RealSet(0,2) + RealSet.unbounded_above_closed(10); s1
(0, 2) \cup [10, +\infty)
sage: s2 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s2
(-\infty, -10] \cup (1, 3)
sage: s1.intersection(s2)
(1, 2)
sage: s1 & s2  # syntactic sugar
(1, 2)
sage: s3 = RealSet((0, 1), (2, 3)); s3
(0, 1) \cup (2, 3)
sage: s4 = RealSet([0, 1], [2, 3]); s4
[0, 1] \cup [2, 3]
sage: s3.intersection(s4)
(0, 1) \cup (2, 3)
sage: s3.intersection([1, 2])
{}
sage: s4.intersection([1, 2])
{1} \cup {2}
sage: s4.intersection(1, 2)
{}
sage: s5 = RealSet.closed_open(1, 10); s5
[1, 10)
sage: s5.intersection(-\infty, +\infty)
[1, 10)
sage: s5.intersection(x != 2, (-\infty, 3), RealSet.real_line()[0])  # needs sage.symbolic
[1, 2) \cup (2, 3)

\textbf{static} \textbf{interval}(lower, upper, lower_closed, upper_closed, **kwds)

Construct an interval

INPUT:

- lower, upper – two real numbers or infinity. They will be sorted if necessary.
- lower\_closed, upper\_closed – boolean; whether the interval is closed at the lower and upper bound of the interval, respectively.
- **kwds – see RealSet.

OUTPUT:

A new \textit{RealSet}.

EXAMPLES:

sage: RealSet.interval(1, 0, lower_closed=True, upper_closed=False)
[0, 1)

\textbf{is\_closed}()

Return whether self is a closed set.

EXAMPLES:

sage: RealSet().is\_closed()
True

(continues on next page)
### is_connected()

Return whether self is a connected set.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```python
sage: s1 = RealSet((1, 2), (2, 4)); s1
(1, 2) ∪ (2, 4)
sage: s1.is_connected()
False
sage: s2 = RealSet((1, 2), (2, 4), RealSet.point(2)); s2
(1, 4)
sage: s2.is_connected()
True
sage: s3 = RealSet(1,3) + RealSet.unbounded_below_closed(-10); s3
(-oo, -10] ∪ (1, 3)
sage: s3.is_connected()
False
sage: RealSet(x != 0).is_connected()  # needs sage.symbolic
False
sage: RealSet(-oo, oo).is_connected()  # needs sage.symbolic
True
sage: RealSet().is_connected()
False
```

### is_disjoint(*other)

Test whether the two sets are disjoint

**INPUT:**

- other – a RealSet or data defining one.

**OUTPUT:**

Boolean.

**See also:**

- are_pairwise_disjoint()

**EXAMPLES:**

```python
sage: s = RealSet((0, 1), (2, 3)); s
(0, 1) ∪ (2, 3)
```
sage: s.is_disjoint(RealSet([1, 2]))
True
sage: s.is_disjoint([3/2, 5/2])
False
sage: s.is_disjoint(RealSet())
True
sage: s.is_disjoint(RealSet().real_line())
False

is_disjoint_from(*args, **kwds)

Deprecated: Use is_disjoint() instead. See github issue #31927 for details.

is_empty()

Return whether the set is empty

EXAMPLES:

sage: RealSet(0, 1).is_empty()
False
sage: RealSet(0, 0).is_empty()
True
sage: RealSet.interval(1, 1, lower_closed=False, upper_closed=True).is_empty()
True
sage: RealSet.interval(1, -1, lower_closed=False, upper_closed=True).is_empty()
False

is_included_in(*args, **kwds)

Deprecated: Use is_subset() instead. See github issue #31927 for details.

is_open()

Return whether self is an open set.

EXAMPLES:

sage: RealSet().is_open()
True
sage: RealSet.point(1).is_open()
False
sage: RealSet((1, 2)).is_open()
True
sage: RealSet([1, 2], (3, 4)).is_open()
False
sage: RealSet(-oo, +oo).is_open()
True

is_subset(*other)

Return whether self is a subset of other.

INPUT:

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

OUTPUT:

Boolean.

EXAMPLES:
Sets, Release 10.2

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

```
is_universe()
Return whether the set is the ambient space (the real line).

EXAMPLES:
```
sage: RealSet().ambient().is_universe()
True
```

```
lift(x)
Lift x to the ambient space for self.
This version of the method just returns x.

EXAMPLES:
```
sage: s = RealSet(0, 2); s
(0, 2)
sage: s.lift(1)
1
```

```
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
```

```
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:
```
```python
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),)
```

**static open**(lower, upper, **kwds)

Construct an open interval

**INPUT:**

- `lower`, `upper` – two real numbers or infinity. They will be sorted if necessary.
- `**kwds` – see `RealSet`.

**OUTPUT:**

A new `RealSet`.

**EXAMPLES:**

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

**static open_closed**(lower, upper, **kwds)

Construct a half-open interval

**INPUT:**

- `lower`, `upper` – two real numbers or infinity. They will be sorted if necessary.
- `**kwds` – see `RealSet`.

**OUTPUT:**

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

**EXAMPLES:**

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

**static point**(p, **kwds)

Construct an interval containing a single point

**INPUT:**

- `p` – a real number.
- `**kwds` – see `RealSet`.

**OUTPUT:**

A new `RealSet`.

**EXAMPLES:**

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

---

2.5. Subsets of the Real Line 111
**static real_line(**kwds)**
Construct the real line

INPUT:

- **kwds** – see *RealSet*.

EXAMPLES:

```
sage: RealSet.real_line()
(-oo, +oo)
```

**retract**(x)
Retract x to self.
It raises an error if x does not lie in the set self.

EXAMPLES:

```
sage: s = RealSet(0, 2); s
(0, 2)
sage: s.retract(1)
1
sage: s.retract(2)
Traceback (most recent call last):
...
ValueError: 2 is not an element of (0, 2)
```

**sup**()
Return the supremum

OUTPUT:

A real number or infinity.

EXAMPLES:

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

**symmetric_difference**(other)
Returns the symmetric difference of self and other.

INPUT:

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

OUTPUT:

The set-theoretic symmetric difference of self and other as a new *RealSet*.

EXAMPLES:
```python
sage: s1 = RealSet(0,2); s1
(0, 2)
sage: s2 = RealSet.unbounded_above_open(1); s2
(1, +oo)
sage: s1.symmetric_difference(s2)
(0, 1] ∪ [2, +oo)
```

**static unbounded_above_closed**(bound, **kwds)

Construct a semi-infinite interval

**INPUT:**

- bound – a real number.
- **kwds – see `RealSet`.

**OUTPUT:**

A new `RealSet` from the bound (including) to plus infinity.

**EXAMPLES:**

```python
sage: RealSet.unbounded_above_closed(1)
[1, +oo)
```

**static unbounded_above_open**(bound, **kwds)

Construct a semi-infinite interval

**INPUT:**

- bound – a real number.
- **kwds – see `RealSet`.

**OUTPUT:**

A new `RealSet` from the bound (excluding) to plus infinity.

**EXAMPLES:**

```python
sage: RealSet.unbounded_above_open(1)
(1, +oo)
```

**static unbounded_below_closed**(bound, **kwds)

Construct a semi-infinite interval

**INPUT:**

- bound – a real number.
- **kwds – see `RealSet`.

**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, **kwds)

Construct a semi-infinite interval

INPUT:
  - bound – a real number.

OUTPUT:
A new RealSet from minus infinity to the bound (excluding).
  - **kwds – see RealSet.

EXAMPLES:

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

union(*real_set_collection)

Return the union of real sets

INPUT:
  - *real_set_collection – a list/tuple/iterable of RealSet or data that defines one.

OUTPUT:
The set-theoretic union as a new RealSet.

EXAMPLES:

```
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)
sage: RealSet().union(RealSet.real_line())
(-oo, +oo)
sage: s = RealSet().union([1, 2], (2, 3)); s
[1, 3)
sage: RealSet().union((-oo, 0), x > 6, s[0],
˓→ needs sage.symbolic
.....: RealSet.point(5.0), RealSet.closed_open(2, 4))
(-oo, 0) ∪ [1, 4) ∪ {5} ∪ (6, +oo)
```
CHAPTER THREE

INDICES AND TABLES

- Index
- Module Index
- Search Page
sage.sets.cartesian_product, 1
sage.sets.condition_set, 65
sage.sets.disjoint_set, 25
sage.sets.disjoint_union Enumerated Sets, 32
sage.sets.family, 3
sage.sets.finite enumerated set, 43
sage.sets.finite_set_map cy, 73
sage.sets.finite_set_maps, 69
sage.sets.integer_range, 83
sage.sets.non_negative_integers, 88
sage.sets.positive integers, 88
sage.sets.primes, 90
sage.sets.pythonclass, 81
sage.sets.real_set, 91
sage.sets.recursively enumerated set, 45
sage.sets.set, 13
sage.sets.set_from_iterator, 36
sage.sets.totally_ordered Finite Set, 79
INDEX

Symbols

_(cartesian_product_of_elements) (sage.sets.cartesian_product.CartesianProduct method), 1

A

AbstractFamily (class in sage.sets.family), 4

ambient() (sage.sets.condition_set.ConditionSet method), 67

ambient() (sage.sets.real_set.RealSet method), 101

an_element() (sage.sets.cartesian_product.CartesianProduct method), 2

an_element() (sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets method), 35

an_element() (sage.sets.finite_enumenrated_set.FiniteEnumeratedSet method), 43

an_element() (sage.sets.finite_set_maps.FiniteSetMaps method), 69

an_element() (sage.sets.finite_set_maps.FiniteSetMaps_MN method), 72

an_element() (sage.sets.non_negative_integers.NonNegativeIntegers method), 89

an_element() (sage.sets.positive_integers.PositiveIntegers method), 88

are_pairwise_disjoint() (sage.sets.real_set.RealSet method), 101

arguments() (sage.sets.condition_set.ConditionSet method), 67

B

boundary() (sage.sets.real_set.RealSet method), 102

boundary_points() (sage.sets.real_set.InternalRealInterval method), 93

breadth_first_search_iterator() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet method), 119
Sets, Release 10.2

- CartesianProduct (class in sage.sets.cartesian_product), 1
- CartesianProduct.Element (class in sage.sets.cartesian_product), 2
- check() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 74
- children() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic method), 55
- clear_cache() (sage.sets.set_from_iterator.EnumeratedSetFromIterator method), 38
- closed() (sage.sets.real_set.RealSet static method), 102
- closed_open() (sage.sets.real_set.RealSet static method), 103
- closure() (sage.sets.real_set.InternalRealInterval method), 93
- codomain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 74
- codomain() (sage.sets.finite_set_maps.FiniteSetMaps_MN method), 72
- codomain() (sage.sets.finite_set_maps.FiniteSetMaps_Set method), 73
- complement() (sage.sets.real_set.RealSet method), 103
- ConditionSet (class in sage.sets.condition_set), 65
- construction() (sage.sets.cartesian_product.CartesianProduct method), 3
- contains() (sage.sets.real_set.InternalRealInterval method), 93
- contains() (sage.sets.real_set.RealSet method), 94
- convex_hull() (sage.sets.real_set.InternalRealInterval method), 94
- convex_hull() (sage.sets.real_set.RealSet static method), 104

D

- Decorator (class in sage.sets.set_from_iterator), 37
- depth_first_search_iterator() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic method), 55
- depth_first_search_iterator() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic method), 58
- difference() (sage.sets.real_set.RealSet method), 105
- difference() (sage.sets.set.Set_base method), 15
- difference() (sage.sets.set.Set_objectEnumerated method), 19
- DisjointSet() (in module sage.sets.disjoint_set), 25
- DisjointSet_class (class in sage.sets.disjoint_set), 26
- DisjointSet_of_hasables (class in sage.sets.disjoint_set), 27
- DisjointSet_of_integers (class in sage.sets.disjoint_set), 29
- DisjointUnionEnumeratedSets (class in sage.sets.disjoint_union_enumerated_sets), 32
- domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 75
- domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 74
- domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 72
- domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 73
- domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 72
- domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 73
- domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 72
- domain() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 73
- Element (sage.sets.finite_set_map_cy.FiniteSetMap_MN attribute), 69
- Element (sage.sets.finite_set_map_cy.FiniteSetMap_MN attribute), 69
- Element (sage.sets.finite_set_map_cy.FiniteSetMap_MN attribute), 72
- Element (sage.sets.finite_set_map_cy.FiniteSetMap_MN attribute), 73
- Element (sage.sets.non_negative_integers.NonNegativeIntegers attribute), 89
- Element (sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet attribute), 80
- element_class (sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets element_class), 35
- element_class (sage.sets.integer_range.IntegerRange attribute), 85
- element_class (sage.sets.real_set.InternalRealInterval attribute), 94
- element_to_root_dict() (sage.sets.disjoint_set.DisjointSet_of_hasables method), 27
- element_to_root_dict() (sage.sets.disjoint_set.DisjointSet_of_integers method), 29
- elements_of_depth_iterator() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSetGeneric elements_of_depth_iterator), 58
- EnumeratedFamily (class in sage.sets.family), 5
- EnumeratedSetFromIterator (class in sage.sets.set_from_iterator), 37
- EnumeratedSetFromIterator_function_decorator (class in sage.sets.set_from_iterator), 39
- EnumeratedSetFromIterator_method_caller (class in sage.sets.set_from_iterator), 41
- EnumeratedSetFromIterator_method_decorator (class in sage.sets.set_from_iterator), 41
intersection() (sage.sets.real_set.InternalRealInterval method), 95
intersection() (sage.sets.real_set.RealSet method), 106
intersection() (sage.sets.Set_base method), 15
intersection() (sage.sets.Set_object_enumerated method), 20
interval() (sage.sets.real_set.RealSet static method), 107
inverse_family() (sage.sets.family.AbstractFamily method), 4
is_closed() (sage.sets.real_set.RealSet method), 107
is_connected() (sage.sets.real_set.InternalRealInterval method), 95
is_connected() (sage.sets.real_set.RealSet method), 108
is_disjoint() (sage.sets.real_set.RealSet method), 108
is_disjoint_from() (sage.sets.real_set.RealSet method), 109
is_empty() (sage.sets.real_set.InternalRealInterval method), 96
is_empty() (sage.sets.real_set.RealSet method), 109
is_empty() (sage.sets.Set_object method), 17
is_finite() (sage.sets.Set_object method), 17
is_finite() (sage.sets.Set_object_difference method), 19
is_finite() (sage.sets.Set_object_enumerated method), 20
is_finite() (sage.sets.Set_object_intersection method), 23
is_finite() (sage.sets.set.Set_object_symmetric_difference method), 23
is_finite() (sage.sets.Set_object_union method), 24
is_included_in() (sage.sets.real_set.RealSet method), 109
is_open() (sage.sets.real_set.RealSet method), 109
is_parent_of() (sage.sets.finite_enumerated_set.FiniteEnumeratedSet method), 44
is_parent_of() (sage.sets.set_from_iterator.EnumeratedSetFromIterator method), 39
is_point() (sage.sets.real_set.InternalRealInterval method), 96
is_subset() (sage.sets.real_set.RealSet method), 109
is_universe() (sage.sets.real_set.RealSet method), 110
issubset() (sage.sets.Set_object_enumerated method), 21
issuperset() (sage.sets.Set_object_enumerated method), 21
items() (sage.sets.family.AbstractFamily method), 4
items() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 75
items() (sage.sets.finite_set_map_cy.FiniteSetMap_Set method), 77
K
keys() (sage.sets.family.AbstractFamily method), 4
keys() (sage.sets.family.FiniteFamily method), 11
keys() (sage.sets.family.LazyFamily method), 12
keys() (sage.sets.family.TrivialFamily method), 12
L
last() (sage.sets.finite_enumerated_set.FiniteEnumeratedSet method), 44
LazyFamily (class in sage.sets.family), 12
le() (sage.sets.totally_ordered_finite_set.TotallyOrderedFiniteSet method), 81
lift() (sage.sets.real_set.RealSet method), 110
list() (sage.sets.finite_enumerated_set.FiniteEnumeratedSet method), 44
list() (sage.sets.Set_object_enumerated method), 21
lower() (sage.sets.real_set.InternalRealInterval method), 96
lower_closed() (sage.sets.real_set.InternalRealInterval method), 97
lower_open() (sage.sets.real_set.InternalRealInterval method), 97
M
map() (sage.sets.family.AbstractFamily method), 4
map() (sage.sets.family.TrivialFamily method), 13
map_reduce() (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet method), 56
module
sage.sets.cartesian_product, 1
sage.sets.condition_set, 65
sage.sets.disjoint_set, 25
sage.sets.disjoint_union_enumerated_sets, 32
sage.sets.family, 3
sage.sets.finite_enumerated_set, 43
sage.sets.finite_set_map_cy, 73
sage.sets.finite_set_maps, 69
sage.sets.integer_range, 83
sage.sets.non_negative_integers, 88
sage.sets.positive_integers, 88
sage.sets.primes, 90
sage.sets.pythonclass, 81
sage.sets.real_set, 91
sage.sets.recursively_enumerated_set, 45
sage.sets.set, 13
sage.sets.set_from_iterator, 36
sage.sets.totally_ordered_finite_set, 79
N
n_components() (sage.sets.real_set.RealSet method), 110
set() (sage.sets.Set object enumerated method), 22
Set_add_sub_operators (class in sage.sets), 15
Set_base (class in sage.sets), 15
Set_boolean_operators (class in sage.sets), 16
set_from_function (in module sage.sets.set_from_iterator), 42
set_from_method (in module sage.sets.set_from_iterator), 42
Set_object (class in sage.sets), 16
Set_object_binary (class in sage.sets), 18
Set_object_difference (class in sage.sets), 19
Set_object.enumerated (class in sage.sets), 19
Set_object_intersection (class in sage.sets), 23
Set_object_symmetric_difference (class in sage.sets), 23
Set_object_union (class in sage.sets), 23
Set_PythonType() (in module sage.sets.pythonclass), 81
Set_PythonType_class (class in sage.sets.pythonclass), 81
setimage() (sage.sets.finite_set_map_cy.FiniteSetMap_MN method), 76
setimage() (sage.sets.finite_set_map_cy.FiniteSetMap_Set method), 77
some_elements() (sage.sets.disjoint_set.DisjointSet_set, 90
start (sage.sets.set_from_iterator.DummyExampleForPicklingTest attribute), 37
stop (sage.sets.set_from_iterator.DummyExampleForPicklingTest attribute), 37
subsets() (sage.sets.set.Set object method), 18
subsets_lattice() (sage.sets.set.Set object method), 18
successors (sage.sets.recursively_enumerated_set.RecursivelyEnumeratedSet_generic attribute), 59
sup() (sage.sets.real_set.RealSet method), 112
symmetric_difference() (sage.sets.real_set.RealSet method), 112
symmetric_difference() (sage.sets.set.Set base method), 16
symmetric_difference() (sage.sets.set.Set object enumerated method), 22

TotallyOrderedFiniteSetElement (class in sage.sets.totally_ordered_finite_set), 81
TrivialFamily (class in sage.sets.family), 12

Un
unbounded_above_closed() (sage.sets.real_set.RealSet static method), 113
unbounded_above_open() (sage.sets.real_set.RealSet static method), 113
unbounded_below_closed() (sage.sets.real_set.RealSet static method), 113
unbounded_below_open() (sage.sets.real_set.RealSet static method), 113
union() (sage.sets.disjoint_set.DisjointSet_set, 29
union() (sage.sets.disjoint_set.DisjointSet_set, 31
union() (sage.sets.real_set.RealSet method), 114
union() (sage.sets.set.Set object enumerated method), 22
unrank() (sage.sets.finite_set_set.FiniteEnumeratedSet method), 44
unrank() (sage.sets.integer_range.IntegerRangeFinite method), 86
unrank() (sage.sets.integer_range.IntegerRangeInfinite method), 87
unrank() (sage.sets.non_negative_integers.NonNegativeInteger method), 90
unrank() (sage.sets.primes.Primes method), 91
unrank() (sage.sets.set_from_iterator.EnumeratedSetFromIterator method), 39
upper() (sage.sets.real_set.InternalRealInterval method), 97
upper_closed() (sage.sets.real_set.InternalRealInterval method), 97
upper_open() (sage.sets.real_set.InternalRealInterval method), 98

V
values() (sage.sets.family.AbstractFamily method), 5
values() (sage.sets.family.FiniteFamily method), 11

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

Z
zip() (sage.sets.family.AbstractFamily method), 5