Monoids

Release 10.6

The Sage Development Team

CONTENTS

Sage supports free monoids and free abelian monoids in any finite number of indeterminates, as well as free partially commutative monoids (trace monoids).

MONOIDS

class sage.monoids.monoid.**Monoid_class**(*names*)

> Bases: `Parent`
>
> EXAMPLES:

```
sage: from sage.monoids.monoid import Monoid_class
sage: Monoid_class(('a','b'))
<sage.monoids.monoid.Monoid_class_with_category object at ...>
```

```
>>> from sage.all import *
>>> from sage.monoids.monoid import Monoid_class
>>> Monoid_class(('a','b'))
<sage.monoids.monoid.Monoid_class_with_category object at ...>
```

> **gens**()
>
> > Return the generators for `self`.
> >
> > EXAMPLES:

```
sage: F.<a,b,c,d,e> = FreeMonoid(5)
sage: F.gens()
(a, b, c, d, e)
```

```
>>> from sage.all import *
>>> F = FreeMonoid(Integer(5), names=('a', 'b', 'c', 'd', 'e',)); (a, b, c, d,
↪ e,) = F._first_ngens(5)
>>> F.gens()
(a, b, c, d, e)
```

> **monoid_generators**()
>
> > Return the generators for `self`.
> >
> > EXAMPLES:

```
sage: F.<a,b,c,d,e> = FreeMonoid(5)
sage: F.monoid_generators()
Family (a, b, c, d, e)
```

```
>>> from sage.all import *
>>> F = FreeMonoid(Integer(5), names=('a', 'b', 'c', 'd', 'e',)); (a, b, c, d,
↪ e,) = F._first_ngens(5)
```

```
>>> F.monoid_generators()
Family (a, b, c, d, e)
```

sage.monoids.monoid.**is_Monoid**(*x*)

Return `True` if `x` is of type `Monoid_class`.

EXAMPLES:

```
sage: from sage.monoids.monoid import is_Monoid
sage: is_Monoid(0)
doctest:warning...
DeprecationWarning: the function is_Monoid is deprecated;
use 'isinstance(..., Monoid_class)' instead
See https://github.com/sagemath/sage/issues/37897 for details.
False
sage: is_Monoid(ZZ)     # The technical math meaning of monoid has
....:                    # no bearing whatsoever on the result: it's
....:                    # a typecheck which is not satisfied by ZZ
....:                    # since it does not inherit from Monoid_class.
False
sage: is_Monoid(sage.monoids.monoid.Monoid_class(('a','b')))
True
sage: F.<a,b,c,d,e> = FreeMonoid(5)
sage: is_Monoid(F)
True
```

```
>>> from sage.all import *
>>> from sage.monoids.monoid import is_Monoid
>>> is_Monoid(Integer(0))
doctest:warning...
DeprecationWarning: the function is_Monoid is deprecated;
use 'isinstance(..., Monoid_class)' instead
See https://github.com/sagemath/sage/issues/37897 for details.
False
>>> is_Monoid(ZZ)     # The technical math meaning of monoid has
...                   # no bearing whatsoever on the result: it's
...                   # a typecheck which is not satisfied by ZZ
...                   # since it does not inherit from Monoid_class.
False
>>> is_Monoid(sage.monoids.monoid.Monoid_class(('a','b')))
True
>>> F = FreeMonoid(Integer(5), names=('a', 'b', 'c', 'd', 'e',)); (a, b, c, d, e,
↪) = F._first_ngens(5)
>>> is_Monoid(F)
True
```

FREE MONOIDS

AUTHORS:

- David Kohel (2005-09)

- Simon King (2011-04): Put free monoids into the category framework

Sage supports free monoids on any prescribed finite number $n \geq 0$ of generators. Use the `FreeMonoid` function to create a free monoid, and the `gen` and `gens` functions to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `FreeMonoid` function.

class sage.monoids.free_monoid.**FreeMonoid**(*n*, *names=None*)

Bases: *Monoid_class*, `UniqueRepresentation`

Return a free monoid on n generators or with the generators indexed by a set I.

We construct free monoids by specifying either:

- the number of generators and/or the names of the generators

- the indexing set for the generators

INPUT:

- `index_set` – an indexing set for the generators; if an integer n, than this becomes $\{0, 1, \ldots, n-1\}$

- `names` – names of generators

- `commutative` – boolean (default: `False`); whether the free monoid is commutative or not

OUTPUT: a free monoid

EXAMPLES:

```
sage: F = FreeMonoid(3,'x'); F
Free monoid on 3 generators (x0, x1, x2)
sage: x = F.gens()
sage: x[0]*x[1]**5 * (x[0]*x[2])
x0*x1^5*x0*x2
sage: F = FreeMonoid(3, 'a')
sage: F
Free monoid on 3 generators (a0, a1, a2)

sage: F.<a,b,c,d,e> = FreeMonoid(); F
Free monoid on 5 generators (a, b, c, d, e)
sage: FreeMonoid(index_set=ZZ)
Free monoid indexed by Integer Ring
```

```
sage: F.<x,y,z> = FreeMonoid(abelian=True); F
Free abelian monoid on 3 generators (x, y, z)
sage: FreeMonoid(index_set=ZZ, commutative=True)
Free abelian monoid indexed by Integer Ring
```

```
>>> from sage.all import *
>>> F = FreeMonoid(Integer(3),'x'); F
Free monoid on 3 generators (x0, x1, x2)
>>> x = F.gens()
>>> x[Integer(0)]*x[Integer(1)]**Integer(5) * (x[Integer(0)]*x[Integer(2)])
x0*x1^5*x0*x2
>>> F = FreeMonoid(Integer(3), 'a')
>>> F
Free monoid on 3 generators (a0, a1, a2)

>>> F = FreeMonoid(names=('a', 'b', 'c', 'd', 'e',)); (a, b, c, d, e,) = F._first_
↪ngens(5); F
Free monoid on 5 generators (a, b, c, d, e)
>>> FreeMonoid(index_set=ZZ)
Free monoid indexed by Integer Ring

>>> F = FreeMonoid(abelian=True, names=('x', 'y', 'z',)); (x, y, z,) = F._first_
↪ngens(3); F
Free abelian monoid on 3 generators (x, y, z)
>>> FreeMonoid(index_set=ZZ, commutative=True)
Free abelian monoid indexed by Integer Ring
```

Element

alias of *FreeMonoidElement*

cardinality()

Return the cardinality of `self`.

This is ∞ if there is at least one generator.

EXAMPLES:

```
sage: F = FreeMonoid(2005, 'a')
sage: F.cardinality()
+Infinity

sage: F = FreeMonoid(0, [])
sage: F.cardinality()
1
```

```
>>> from sage.all import *
>>> F = FreeMonoid(Integer(2005), 'a')
>>> F.cardinality()
+Infinity

>>> F = FreeMonoid(Integer(0), [])
>>> F.cardinality()
1
```

gen(*i=0*)

> The i-th generator of the monoid.
>
> INPUT:
>
> - i – integer (default: 0)
>
> EXAMPLES:

```
sage: F = FreeMonoid(3, 'a')
sage: F.gen(1)
a1
sage: F.gen(2)
a2
sage: F.gen(5)
Traceback (most recent call last):
...
IndexError: argument i (= 5) must be between 0 and 2
```

```
>>> from sage.all import *
>>> F = FreeMonoid(Integer(3), 'a')
>>> F.gen(Integer(1))
a1
>>> F.gen(Integer(2))
a2
>>> F.gen(Integer(5))
Traceback (most recent call last):
...
IndexError: argument i (= 5) must be between 0 and 2
```

ngens()

> The number of free generators of the monoid.
>
> EXAMPLES:

```
sage: F = FreeMonoid(2005, 'a')
sage: F.ngens()
2005
```

```
>>> from sage.all import *
>>> F = FreeMonoid(Integer(2005), 'a')
>>> F.ngens()
2005
```

sage.monoids.free_monoid.**is_FreeMonoid**(*x*)

> Return `True` if x is a free monoid.
>
> EXAMPLES:

```
sage: from sage.monoids.free_monoid import is_FreeMonoid
sage: is_FreeMonoid(5)
doctest:warning...
DeprecationWarning: the function is_FreeMonoid is deprecated;
use 'isinstance(..., (FreeMonoid, IndexedFreeMonoid))' instead
See https://github.com/sagemath/sage/issues/37897 for details.
```

```
False
sage: is_FreeMonoid(FreeMonoid(7,'a'))
True
sage: is_FreeMonoid(FreeAbelianMonoid(7,'a'))
False
sage: is_FreeMonoid(FreeAbelianMonoid(0,''))
False
sage: is_FreeMonoid(FreeMonoid(index_set=ZZ))
True
sage: is_FreeMonoid(FreeAbelianMonoid(index_set=ZZ))
False
```

```
>>> from sage.all import *
>>> from sage.monoids.free_monoid import is_FreeMonoid
>>> is_FreeMonoid(Integer(5))
doctest:warning...
DeprecationWarning: the function is_FreeMonoid is deprecated;
use 'isinstance(..., (FreeMonoid, IndexedFreeMonoid))' instead
See https://github.com/sagemath/sage/issues/37897 for details.
False
>>> is_FreeMonoid(FreeMonoid(Integer(7),'a'))
True
>>> is_FreeMonoid(FreeAbelianMonoid(Integer(7),'a'))
False
>>> is_FreeMonoid(FreeAbelianMonoid(Integer(0),''))
False
>>> is_FreeMonoid(FreeMonoid(index_set=ZZ))
True
>>> is_FreeMonoid(FreeAbelianMonoid(index_set=ZZ))
False
```

ELEMENTS OF FREE MONOIDS

AUTHORS:

- David Kohel (2005-09-29)

Elements of free monoids are represented internally as lists of pairs of integers.

class sage.monoids.free_monoid_element.**FreeMonoidElement**(*F*, *x*, *check=True*)

Bases: MonoidElement

Element of a free monoid.

EXAMPLES:

```
sage: a = FreeMonoid(5, 'a').gens()
sage: x = a[0]*a[1]*a[4]**3
sage: x**3
a0*a1*a4^3*a0*a1*a4^3*a0*a1*a4^3
sage: x**0
1
sage: x**(-1)
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> from sage.all import *
>>> a = FreeMonoid(Integer(5), 'a').gens()
>>> x = a[Integer(0)]*a[Integer(1)]*a[Integer(4)]**Integer(3)
>>> x**Integer(3)
a0*a1*a4^3*a0*a1*a4^3*a0*a1*a4^3
>>> x**Integer(0)
1
>>> x**(-Integer(1))
Traceback (most recent call last):
...
NotImplementedError
```

to_list(*indices=False*)

Return self as a list of generators.

If self equals $x_{i_1} x_{i_2} \cdots x_{i_n}$, with $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ being some of the generators of the free monoid, then this method returns the list $[x_{i_1}, x_{i_2}, \ldots, x_{i_n}]$.

If the optional argument indices is set to True, then the list $[i_1, i_2, \ldots, i_n]$ is returned instead.

EXAMPLES:

```
sage: M.<x,y,z> = FreeMonoid(3)
sage: a = x * x * y * x
sage: w = a.to_list(); w
[x, x, y, x]
sage: M.prod(w) == a
True
sage: w = a.to_list(indices=True); w
[0, 0, 1, 0]
sage: a = M.one()
sage: a.to_list()
[]
```

```
>>> from sage.all import *
>>> M = FreeMonoid(Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = M._first_
↪ngens(3)
>>> a = x * x * y * x
>>> w = a.to_list(); w
[x, x, y, x]
>>> M.prod(w) == a
True
>>> w = a.to_list(indices=True); w
[0, 0, 1, 0]
>>> a = M.one()
>>> a.to_list()
[]
```

> ↪ **See also**
>
> *to_word()*

to_word(*alph=None*)

> Return self as a word.
>
> INPUT:
>
> • alph – (optional) the alphabet which the result should be specified in
>
> EXAMPLES:

```
sage: M.<x,y,z> = FreeMonoid(3)
sage: a = x * x * y * x
sage: w = a.to_word(); w
word: xxyx
sage: w.to_monoid_element() == a
True
```

```
>>> from sage.all import *
>>> M = FreeMonoid(Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = M._first_
↪ngens(3)
>>> a = x * x * y * x
>>> w = a.to_word(); w
word: xxyx
```

(continues on next page)

```
>>> w.to_monoid_element() == a
True
```

> ↪ **See also**
>
> `to_list()`

sage.monoids.free_monoid_element.**is_FreeMonoidElement**(*x*)

FREE ABELIAN MONOIDS

AUTHORS:

- David Kohel (2005-09)

Sage supports free abelian monoids on any prescribed finite number $n \geq 0$ of generators. Use the `FreeAbelianMonoid` function to create a free abelian monoid, and the `gen` and `gens` functions to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `FreeAbelianMonoid` function.

EXAMPLE 1: It is possible to create an abelian monoid in zero or more variables; the syntax T(1) creates the monoid identity element even in the rank zero case.

```
sage: T = FreeAbelianMonoid(0, '')
sage: T
Free abelian monoid on 0 generators ()
sage: T.gens()
()
sage: T(1)
1
```

```
>>> from sage.all import *
>>> T = FreeAbelianMonoid(Integer(0), '')
>>> T
Free abelian monoid on 0 generators ()
>>> T.gens()
()
>>> T(Integer(1))
1
```

EXAMPLE 2: A free abelian monoid uses a multiplicative representation of elements, but the underlying representation is lists of integer exponents.

```
sage: F = FreeAbelianMonoid(5,names='a,b,c,d,e')
sage: (a,b,c,d,e) = F.gens()
sage: a*b^2*e*d
a*b^2*d*e
sage: x = b^2*e*d*a^7
sage: x
a^7*b^2*d*e
sage: x.list()
[7, 2, 0, 1, 1]
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(Integer(5),names='a,b,c,d,e')
>>> (a,b,c,d,e) = F.gens()
>>> a*b**Integer(2)*e*d
a*b^2*d*e
>>> x = b**Integer(2)*e*d*a**Integer(7)
>>> x
a^7*b^2*d*e
>>> x.list()
[7, 2, 0, 1, 1]
```

sage.monoids.free_abelian_monoid.**FreeAbelianMonoid**(*index_set=None*, *names=None*, ***kwds*)

Return a free abelian monoid on n generators or with the generators indexed by a set I.

We construct free abelian monoids by specifying either:

- the number of generators and/or the names of the generators

- the indexing set for the generators (this ignores the other two inputs)

INPUT:

- index_set – an indexing set for the generators; if an integer, then this becomes $\{0, 1, \ldots, n-1\}$

- names – names of generators

OUTPUT: a free abelian monoid

EXAMPLES:

```
sage: F.<a,b,c,d,e> = FreeAbelianMonoid(); F
Free abelian monoid on 5 generators (a, b, c, d, e)
sage: FreeAbelianMonoid(index_set=ZZ)
Free abelian monoid indexed by Integer Ring
sage: FreeAbelianMonoid(names='x,y')
Free abelian monoid on 2 generators (x, y)
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(names=('a', 'b', 'c', 'd', 'e',)); (a, b, c, d, e,) = F.
↪_first_ngens(5); F
Free abelian monoid on 5 generators (a, b, c, d, e)
>>> FreeAbelianMonoid(index_set=ZZ)
Free abelian monoid indexed by Integer Ring
>>> FreeAbelianMonoid(names='x,y')
Free abelian monoid on 2 generators (x, y)
```

class sage.monoids.free_abelian_monoid.**FreeAbelianMonoidFactory**

Bases: `UniqueFactory`

Create the free abelian monoid in n generators.

INPUT:

- n – integer

- names – names of generators

OUTPUT: free abelian monoid

EXAMPLES:

```
sage: FreeAbelianMonoid(0, '')
Free abelian monoid on 0 generators ()
sage: F = FreeAbelianMonoid(5,names = list("abcde"))
sage: F
Free abelian monoid on 5 generators (a, b, c, d, e)
sage: F(1)
1
sage: (a, b, c, d, e) = F.gens()
sage: mul([ a, b, a, c, b, d, c, d ], F(1))
a^2*b^2*c^2*d^2
sage: a**2 * b**3 * a**2 * b**4
a^4*b^7
```

```
>>> from sage.all import *
>>> FreeAbelianMonoid(Integer(0), '')
Free abelian monoid on 0 generators ()
>>> F = FreeAbelianMonoid(Integer(5),names = list("abcde"))
>>> F
Free abelian monoid on 5 generators (a, b, c, d, e)
>>> F(Integer(1))
1
>>> (a, b, c, d, e) = F.gens()
>>> mul([ a, b, a, c, b, d, c, d ], F(Integer(1)))
a^2*b^2*c^2*d^2
>>> a**Integer(2) * b**Integer(3) * a**Integer(2) * b**Integer(4)
a^4*b^7
```

```
sage: loads(dumps(F)) is F
True
```

```
>>> from sage.all import *
>>> loads(dumps(F)) is F
True
```

> **create_key**(*n*, *names*)
>
> **create_object**(*version*, *key*)

class sage.monoids.free_abelian_monoid.**FreeAbelianMonoid_class**(*n*, *names*)

> Bases: `Parent`
>
> Free abelian monoid on n generators.
>
> **Element**
>
>> alias of *FreeAbelianMonoidElement*
>
> **cardinality**()
>
>> Return the cardinality of `self`, which is ∞.
>>
>> EXAMPLES:
>>
>> ```
>> sage: F = FreeAbelianMonoid(3000, 'a')
>> sage: F.cardinality()
>> +Infinity
>> ```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(Integer(3000), 'a')
>>> F.cardinality()
+Infinity
```

gen(*i=0*)

The i-th generator of the abelian monoid.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(5,'a')
sage: F.gen(0)
a0
sage: F.gen(2)
a2
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(Integer(5),'a')
>>> F.gen(Integer(0))
a0
>>> F.gen(Integer(2))
a2
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(5,'a')
sage: F.gens()
(a0, a1, a2, a3, a4)
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(Integer(5),'a')
>>> F.gens()
(a0, a1, a2, a3, a4)
```

ngens()

The number of free generators of the abelian monoid.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(3000, 'a')
sage: F.ngens()
3000
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(Integer(3000), 'a')
>>> F.ngens()
3000
```

sage.monoids.free_abelian_monoid.**is_FreeAbelianMonoid**(*x*)

Return `True` if x is a free abelian monoid.

EXAMPLES:

```
sage: from sage.monoids.free_abelian_monoid import is_FreeAbelianMonoid
sage: is_FreeAbelianMonoid(5)
doctest:warning...
DeprecationWarning: the function is_FreeAbelianMonoid is deprecated;
use 'isinstance(..., FreeAbelianMonoid_class)' instead
See https://github.com/sagemath/sage/issues/37897 for details.
False
sage: is_FreeAbelianMonoid(FreeAbelianMonoid(7,'a'))
True
sage: is_FreeAbelianMonoid(FreeMonoid(7,'a'))
False
sage: is_FreeAbelianMonoid(FreeMonoid(0,''))
False
```

```
>>> from sage.all import *
>>> from sage.monoids.free_abelian_monoid import is_FreeAbelianMonoid
>>> is_FreeAbelianMonoid(Integer(5))
doctest:warning...
DeprecationWarning: the function is_FreeAbelianMonoid is deprecated;
use 'isinstance(..., FreeAbelianMonoid_class)' instead
See https://github.com/sagemath/sage/issues/37897 for details.
False
>>> is_FreeAbelianMonoid(FreeAbelianMonoid(Integer(7),'a'))
True
>>> is_FreeAbelianMonoid(FreeMonoid(Integer(7),'a'))
False
>>> is_FreeAbelianMonoid(FreeMonoid(Integer(0),''))
False
```

ABELIAN MONOID ELEMENTS

AUTHORS:

- David Kohel (2005-09)

EXAMPLES:

Recall the example from abelian monoids:

```
sage: F = FreeAbelianMonoid(5,names = list("abcde"))
sage: (a,b,c,d,e) = F.gens()
sage: a*b^2*e*d
a*b^2*d*e
sage: x = b^2*e*d*a^7
sage: x
a^7*b^2*d*e
sage: x.list()
[7, 2, 0, 1, 1]
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(Integer(5),names = list("abcde"))
>>> (a,b,c,d,e) = F.gens()
>>> a*b**Integer(2)*e*d
a*b^2*d*e
>>> x = b**Integer(2)*e*d*a**Integer(7)
>>> x
a^7*b^2*d*e
>>> x.list()
[7, 2, 0, 1, 1]
```

The list is a copy, so changing the list does not change the element:

```
sage: x.list()[0] = 0
sage: x
a^7*b^2*d*e
```

```
>>> from sage.all import *
>>> x.list()[Integer(0)] = Integer(0)
>>> x
a^7*b^2*d*e
```

class sage.monoids.free_abelian_monoid_element.**FreeAbelianMonoidElement**

Bases: `MonoidElement`

Create the element `x` of the FreeAbelianMonoid `parent`.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(5, 'abcde')
sage: F
Free abelian monoid on 5 generators (a, b, c, d, e)
sage: F(1)
1
sage: F(2)
Traceback (most recent call last):
...
TypeError: argument x (= 2) is of the wrong type
sage: F(int(1))
1
sage: a, b, c, d, e = F.gens()
sage: a^2 * b^3 * a^2 * b^4
a^4*b^7
sage: F = FreeAbelianMonoid(5, 'abcde')
sage: a, b, c, d, e = F.gens()
sage: a in F
True
sage: a*b in F
True
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(Integer(5), 'abcde')
>>> F
Free abelian monoid on 5 generators (a, b, c, d, e)
>>> F(Integer(1))
1
>>> F(Integer(2))
Traceback (most recent call last):
...
TypeError: argument x (= 2) is of the wrong type
>>> F(int(Integer(1)))
1
>>> a, b, c, d, e = F.gens()
>>> a**Integer(2) * b**Integer(3) * a**Integer(2) * b**Integer(4)
a^4*b^7
>>> F = FreeAbelianMonoid(Integer(5), 'abcde')
>>> a, b, c, d, e = F.gens()
>>> a in F
True
>>> a*b in F
True
```

list()

> Return the underlying list used to represent `self`.

> If this is a monoid in an abelian monoid on n generators, then this is a list of nonnegative integers of length n.

> EXAMPLES:

```
sage: F = FreeAbelianMonoid(5, 'abcde')
sage: (a, b, c, d, e) = F.gens()
sage: a.list()
[1, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(Integer(5), 'abcde')
>>> (a, b, c, d, e) = F.gens()
>>> a.list()
[1, 0, 0, 0, 0]
```

sage.monoids.free_abelian_monoid_element.**is_FreeAbelianMonoidElement**(*x*)

    Queries whether x is an object of type `FreeAbelianMonoidElement`.

    INPUT:

        • x – an object

    OUTPUT:

        • `True` if x is an object of type `FreeAbelianMonoidElement`; `False` otherwise.

INDEXED MONOIDS

AUTHORS:

• Travis Scrimshaw (2013-10-15)

class sage.monoids.indexed_free_monoid.**IndexedFreeAbelianMonoid**(*indices*, *prefix*, *category=None*, *names=None*, ***kwds*)

Bases: *IndexedMonoid*

Free abelian monoid with an indexed set of generators.

INPUT:

• indices – the indices for the generators

For the optional arguments that control the printing, see IndexedGenerators.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: F.gen(15)^3 * F.gen(2) * F.gen(15)
F[2]*F[15]^4
sage: F.gen(1)
F[1]
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> F.gen(Integer(15))**Integer(3) * F.gen(Integer(2)) * F.gen(Integer(15))
F[2]*F[15]^4
>>> F.gen(Integer(1))
F[1]
```

Now we examine some of the printing options:

```
sage: F = FreeAbelianMonoid(index_set=Partitions(), prefix='A', bracket=False,␣
↪scalar_mult='%')
sage: F.gen([3,1,1]) * F.gen([2,2])
A[2, 2]%A[3, 1, 1]
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=Partitions(), prefix='A', bracket=False,␣
↪scalar_mult='%')
>>> F.gen([Integer(3),Integer(1),Integer(1)]) * F.gen([Integer(2),Integer(2)])
A[2, 2]%A[3, 1, 1]
```

> **✎ Todo**
>
> Implement a subclass when the index sets is finite that utilizes vectors or the polydict monomials with the index order fixed.

Element

   alias of *IndexedFreeAbelianMonoidElement*

gen(x)

   The generator indexed by `x` of `self`.

   EXAMPLES:

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: F.gen(0)
F[0]
sage: F.gen(2)
F[2]
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> F.gen(Integer(0))
F[0]
>>> F.gen(Integer(2))
F[2]
```

one()

   Return the identity element of `self`.

   EXAMPLES:

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: F.one()
1
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> F.one()
1
```

class sage.monoids.indexed_free_monoid.**IndexedFreeAbelianMonoidElement**(F, x)

   Bases: *IndexedMonoidElement*

   An element of an indexed free abelian monoid.

dict()

   Return `self` as a dictionary.

   EXAMPLES:

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: (a*c^3).dict()
{0: 1, 2: 3}
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> (a*c**Integer(3)).dict()
{0: 1, 2: 3}
```

divides(*m*)

Return whether self divides m.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: elt = a*b*c^3*d^2
sage: a.divides(elt)
True
sage: c.divides(elt)
True
sage: (a*b*d^2).divides(elt)
True
sage: (a^4).divides(elt)
False
sage: e.divides(elt)
False
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> elt = a*b*c**Integer(3)*d**Integer(2)
>>> a.divides(elt)
True
>>> c.divides(elt)
True
>>> (a*b*d**Integer(2)).divides(elt)
True
>>> (a**Integer(4)).divides(elt)
False
>>> e.divides(elt)
False
```

length()

Return the length of self.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: elt = a*c^3*b^2*a
sage: elt.length()
7
sage: len(elt)
7
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> elt = a*c**Integer(3)*b**Integer(2)*a
>>> elt.length()
7
>>> len(elt)
7
```

class sage.monoids.indexed_free_monoid.**IndexedFreeMonoid**(*indices*, *prefix*, *category=None*, *names=None*, ***kwds*)

Bases: *IndexedMonoid*

Free monoid with an indexed set of generators.

INPUT:

- indices – the indices for the generators

For the optional arguments that control the printing, see IndexedGenerators.

EXAMPLES:

```
sage: F = FreeMonoid(index_set=ZZ)
sage: F.gen(15)^3 * F.gen(2) * F.gen(15)
F[15]^3*F[2]*F[15]
sage: F.gen(1)
F[1]
```

```
>>> from sage.all import *
>>> F = FreeMonoid(index_set=ZZ)
>>> F.gen(Integer(15))**Integer(3) * F.gen(Integer(2)) * F.gen(Integer(15))
F[15]^3*F[2]*F[15]
>>> F.gen(Integer(1))
F[1]
```

Now we examine some of the printing options:

```
sage: F = FreeMonoid(index_set=ZZ, prefix='X', bracket=['|','>'])
sage: F.gen(2) * F.gen(12)
X|2>*X|12>
```

```
>>> from sage.all import *
>>> F = FreeMonoid(index_set=ZZ, prefix='X', bracket=['|','>'])
>>> F.gen(Integer(2)) * F.gen(Integer(12))
X|2>*X|12>
```

Element

alias of *IndexedFreeMonoidElement*

gen(*x*)

The generator indexed by x of self.

EXAMPLES:

```
sage: F = FreeMonoid(index_set=ZZ)
sage: F.gen(0)
F[0]
sage: F.gen(2)
F[2]
```

```
>>> from sage.all import *
>>> F = FreeMonoid(index_set=ZZ)
>>> F.gen(Integer(0))
F[0]
>>> F.gen(Integer(2))
F[2]
```

one()

Return the identity element of `self`.

EXAMPLES:

```
sage: F = FreeMonoid(index_set=ZZ)
sage: F.one()
1
```

```
>>> from sage.all import *
>>> F = FreeMonoid(index_set=ZZ)
>>> F.one()
1
```

class sage.monoids.indexed_free_monoid.**IndexedFreeMonoidElement**(F, x)

Bases: *IndexedMonoidElement*

An element of an indexed free abelian monoid.

length()

Return the length of `self`.

EXAMPLES:

```
sage: F = FreeMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: elt = a*c^3*b^2*a
sage: elt.length()
7
sage: len(elt)
7
```

```
>>> from sage.all import *
>>> F = FreeMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> elt = a*c**Integer(3)*b**Integer(2)*a
>>> elt.length()
7
>>> len(elt)
7
```

class sage.monoids.indexed_free_monoid.**IndexedMonoid**(*indices*, *prefix*, *category=None*, *names=None*, ***kwds*)

> Bases: `Parent`, `IndexedGenerators`, `UniqueRepresentation`
>
> Base class for monoids with an indexed set of generators.
>
> INPUT:
>
> > • `indices` – the indices for the generators
>
> For the optional arguments that control the printing, see `IndexedGenerators`.
>
> **cardinality**()
>
> > Return the cardinality of `self`, which is ∞ unless this is the trivial monoid.
> >
> > EXAMPLES:
> >
> > ```
> > sage: F = FreeMonoid(index_set=ZZ)
> > sage: F.cardinality()
> > +Infinity
> > sage: F = FreeMonoid(index_set=())
> > sage: F.cardinality()
> > 1
> >
> > sage: F = FreeAbelianMonoid(index_set=ZZ)
> > sage: F.cardinality()
> > +Infinity
> > sage: F = FreeAbelianMonoid(index_set=())
> > sage: F.cardinality()
> > 1
> > ```
> >
> > ```
> > >>> from sage.all import *
> > >>> F = FreeMonoid(index_set=ZZ)
> > >>> F.cardinality()
> > +Infinity
> > >>> F = FreeMonoid(index_set=())
> > >>> F.cardinality()
> > 1
> >
> > >>> F = FreeAbelianMonoid(index_set=ZZ)
> > >>> F.cardinality()
> > +Infinity
> > >>> F = FreeAbelianMonoid(index_set=())
> > >>> F.cardinality()
> > 1
> > ```
>
> **gens**()
>
> > Return the monoid generators of `self`.
> >
> > EXAMPLES:
> >
> > ```
> > sage: F = FreeAbelianMonoid(index_set=ZZ)
> > sage: F.monoid_generators()
> > Lazy family (Generator map from Integer Ring to
> >  Free abelian monoid indexed by Integer Ring(i))_{i in Integer Ring}
> > sage: F = FreeAbelianMonoid(index_set=tuple('abcde'))
> > ```
> >
> > (continues on next page)

```
sage: sorted(F.monoid_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> F.monoid_generators()
Lazy family (Generator map from Integer Ring to
 Free abelian monoid indexed by Integer Ring(i))_{i in Integer Ring}
>>> F = FreeAbelianMonoid(index_set=tuple('abcde'))
>>> sorted(F.monoid_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

monoid_generators()

> Return the monoid generators of `self`.
>
> EXAMPLES:

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: F.monoid_generators()
Lazy family (Generator map from Integer Ring to
 Free abelian monoid indexed by Integer Ring(i))_{i in Integer Ring}
sage: F = FreeAbelianMonoid(index_set=tuple('abcde'))
sage: sorted(F.monoid_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> F.monoid_generators()
Lazy family (Generator map from Integer Ring to
 Free abelian monoid indexed by Integer Ring(i))_{i in Integer Ring}
>>> F = FreeAbelianMonoid(index_set=tuple('abcde'))
>>> sorted(F.monoid_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

class sage.monoids.indexed_free_monoid.**IndexedMonoidElement**(F, x)

> Bases: `MonoidElement`
>
> An element of an indexed monoid.
>
> This is an abstract class which uses the (abstract) method `_sorted_items()` for all of its functions. So to implement an element of an indexed monoid, one just needs to implement `_sorted_items()`, which returns a list of pairs `(i, p)` where `i` is the index and `p` is the corresponding power, sorted in some order. For example, in the free monoid there is no such choice, but for the free abelian monoid, one could want lex order or have the highest powers first.
>
> Indexed monoid elements are ordered lexicographically with respect to the result of `_sorted_items()` (which for abelian free monoids is influenced by the order on the indexing set).
>
> **is_one**()
>
> > Return if `self` is the identity element.
> >
> > EXAMPLES:

```
sage: F = FreeMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: (b*a*c^3*a).is_one()
False
sage: F.one().is_one()
True
```

```
>>> from sage.all import *
>>> F = FreeMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> (b*a*c**Integer(3)*a).is_one()
False
>>> F.one().is_one()
True
```

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: (b*c^3*a).is_one()
False
sage: F.one().is_one()
True
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> (b*c**Integer(3)*a).is_one()
False
>>> F.one().is_one()
True
```

leading_support()

> Return the support of the leading generator of `self`.
>
> EXAMPLES:

```
sage: F = FreeMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: (b*a*c^3*a).leading_support()
1
```

```
>>> from sage.all import *
>>> F = FreeMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> (b*a*c**Integer(3)*a).leading_support()
1
```

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: (b*c^3*a).leading_support()
0
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> (b*c**Integer(3)*a).leading_support()
0
```

support()

> Return a list of the objects indexing self with nonzero exponents.
>
> EXAMPLES:
>
> ```
> sage: F = FreeMonoid(index_set=ZZ)
> sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
> sage: (b*a*c^3*b).support()
> [0, 1, 2]
> ```
>
> ```
> >>> from sage.all import *
> >>> F = FreeMonoid(index_set=ZZ)
> >>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
> >>> (b*a*c**Integer(3)*b).support()
> [0, 1, 2]
> ```
>
> ```
> sage: F = FreeAbelianMonoid(index_set=ZZ)
> sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
> sage: (a*c^3).support()
> [0, 2]
> ```
>
> ```
> >>> from sage.all import *
> >>> F = FreeAbelianMonoid(index_set=ZZ)
> >>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
> >>> (a*c**Integer(3)).support()
> [0, 2]
> ```

to_word_list()

> Return self as a word represented as a list whose entries are indices of self.
>
> EXAMPLES:
>
> ```
> sage: F = FreeMonoid(index_set=ZZ)
> sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
> sage: (b*a*c^3*a).to_word_list()
> [1, 0, 2, 2, 2, 0]
> ```
>
> ```
> >>> from sage.all import *
> >>> F = FreeMonoid(index_set=ZZ)
> >>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
> >>> (b*a*c**Integer(3)*a).to_word_list()
> [1, 0, 2, 2, 2, 0]
> ```
>
> ```
> sage: F = FreeAbelianMonoid(index_set=ZZ)
> sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
> sage: (b*c^3*a).to_word_list()
> [0, 1, 2, 2, 2]
> ```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> (b*c**Integer(3)*a).to_word_list()
[0, 1, 2, 2, 2]
```

trailing_support()

> Return the support of the trailing generator of `self`.

> EXAMPLES:

```
sage: F = FreeMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: (b*a*c^3*a).trailing_support()
0
```

```
>>> from sage.all import *
>>> F = FreeMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> (b*a*c**Integer(3)*a).trailing_support()
0
```

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: a,b,c,d,e = [F.gen(i) for i in range(5)]
sage: (b*c^3*a).trailing_support()
2
```

```
>>> from sage.all import *
>>> F = FreeAbelianMonoid(index_set=ZZ)
>>> a,b,c,d,e = [F.gen(i) for i in range(Integer(5))]
>>> (b*c**Integer(3)*a).trailing_support()
2
```

FREE STRING MONOIDS

AUTHORS:

- David Kohel <kohel@maths.usyd.edu.au>, 2007-01

Sage supports a wide range of specific free string monoids.

class sage.monoids.string_monoid.**AlphabeticStringMonoid**

Bases: *StringMonoid_class*

The free alphabetic string monoid on generators A-Z.

EXAMPLES:

```
sage: S = AlphabeticStrings(); S
Free alphabetic string monoid on A-Z
sage: S.gen(0)
A
sage: S.gen(25)
Z
sage: S([ i for i in range(26) ])
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
>>> from sage.all import *
>>> S = AlphabeticStrings(); S
Free alphabetic string monoid on A-Z
>>> S.gen(Integer(0))
A
>>> S.gen(Integer(25))
Z
>>> S([ i for i in range(Integer(26)) ])
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

characteristic_frequency(*table_name='beker_piper'*)

Return a table of the characteristic frequency probability distribution of the English alphabet. In written English, various letters of the English alphabet occur more frequently than others. For example, the letter "E" appears more often than other vowels such as "A", "I", "O", and "U". In long works of written English such as books, the probability of a letter occurring tends to stabilize around a value. We call this value the characteristic frequency probability of the letter under consideration. When this probability is considered for each letter of the English alphabet, the resulting probabilities for all letters of this alphabet is referred to as the characteristic frequency probability distribution. Various studies report slightly different values for the characteristic frequency probability of an English letter. For instance, [Lew2000] reports that "E" has a characteristic frequency probability of 0.12702, while [BP1982] reports this value as 0.127. The concepts of

characteristic frequency probability and characteristic frequency probability distribution can also be applied to non-empty alphabets other than the English alphabet.

The output of this method is different from that of the method *frequency_distribution()*. One can think of the characteristic frequency probability of an element in an alphabet A as the expected probability of that element occurring. Let S be a string encoded using elements of A. The frequency probability distribution corresponding to S provides us with the frequency probability of each element of A as observed occurring in S. Thus one distribution provides expected probabilities, while the other provides observed probabilities.

INPUT:

- `table_name` – (default: `'beker_piper'`) the table of characteristic frequency probability distribution to use. The following tables are supported:

    - `'beker_piper'` – the table of characteristic frequency probability distribution by Beker and Piper [BP1982]. This is the default table to use.

    - `'lewand'` – the table of characteristic frequency probability distribution by Lewand as described on page 36 of [Lew2000].

OUTPUT:

- A table of the characteristic frequency probability distribution of the English alphabet. This is a dictionary of letter/probability pairs.

EXAMPLES:

The characteristic frequency probability distribution table of Beker and Piper [BP1982]:

```
sage: A = AlphabeticStrings()
sage: table = A.characteristic_frequency(table_name='beker_piper')
sage: sorted(table.items())

[('A', 0.0820000000000000),
('B', 0.0150000000000000),
('C', 0.0280000000000000),
('D', 0.0430000000000000),
('E', 0.127000000000000),
('F', 0.0220000000000000),
('G', 0.0200000000000000),
('H', 0.0610000000000000),
('I', 0.0700000000000000),
('J', 0.00200000000000000),
('K', 0.00800000000000000),
('L', 0.0400000000000000),
('M', 0.0240000000000000),
('N', 0.0670000000000000),
('O', 0.0750000000000000),
('P', 0.0190000000000000),
('Q', 0.00100000000000000),
('R', 0.0600000000000000),
('S', 0.0630000000000000),
('T', 0.0910000000000000),
('U', 0.0280000000000000),
('V', 0.0100000000000000),
('W', 0.0230000000000000),
('X', 0.00100000000000000),
```

(continues on next page)

```
('Y', 0.0200000000000000),
('Z', 0.00100000000000000)]
```

```
>>> from sage.all import *
>>> A = AlphabeticStrings()
>>> table = A.characteristic_frequency(table_name='beker_piper')
>>> sorted(table.items())
<BLANKLINE>
[('A', 0.0820000000000000),
('B', 0.0150000000000000),
('C', 0.0280000000000000),
('D', 0.0430000000000000),
('E', 0.127000000000000),
('F', 0.0220000000000000),
('G', 0.0200000000000000),
('H', 0.0610000000000000),
('I', 0.0700000000000000),
('J', 0.00200000000000000),
('K', 0.0080000000000000),
('L', 0.0400000000000000),
('M', 0.0240000000000000),
('N', 0.0670000000000000),
('O', 0.0750000000000000),
('P', 0.0190000000000000),
('Q', 0.00100000000000000),
('R', 0.0600000000000000),
('S', 0.0630000000000000),
('T', 0.0910000000000000),
('U', 0.0280000000000000),
('V', 0.0100000000000000),
('W', 0.0230000000000000),
('X', 0.00100000000000000),
('Y', 0.0200000000000000),
('Z', 0.00100000000000000)]
```

The characteristic frequency probability distribution table of Lewand [Lew2000]:

```
sage: table = A.characteristic_frequency(table_name='lewand')
sage: sorted(table.items())

[('A', 0.0816700000000000),
('B', 0.0149200000000000),
('C', 0.0278200000000000),
('D', 0.0425300000000000),
('E', 0.127020000000000),
('F', 0.0222800000000000),
('G', 0.0201500000000000),
('H', 0.0609400000000000),
('I', 0.0696600000000000),
('J', 0.00153000000000000),
('K', 0.00772000000000000),
('L', 0.0402500000000000),
```

```
('M', 0.0240600000000000),
('N', 0.0674900000000000),
('O', 0.0750700000000000),
('P', 0.0192900000000000),
('Q', 0.000950000000000000),
('R', 0.0598700000000000),
('S', 0.0632700000000000),
('T', 0.0905600000000000),
('U', 0.0275800000000000),
('V', 0.00978000000000000),
('W', 0.0236000000000000),
('X', 0.00150000000000000),
('Y', 0.0197400000000000),
('Z', 0.000740000000000000)]
```

```
>>> from sage.all import *
>>> table = A.characteristic_frequency(table_name='lewand')
>>> sorted(table.items())
<BLANKLINE>
[('A', 0.0816700000000000),
('B', 0.0149200000000000),
('C', 0.0278200000000000),
('D', 0.0425300000000000),
('E', 0.127020000000000),
('F', 0.0222800000000000),
('G', 0.0201500000000000),
('H', 0.0609400000000000),
('I', 0.0696600000000000),
('J', 0.00153000000000000),
('K', 0.00772000000000000),
('L', 0.0402500000000000),
('M', 0.0240600000000000),
('N', 0.0674900000000000),
('O', 0.0750700000000000),
('P', 0.0192900000000000),
('Q', 0.000950000000000000),
('R', 0.0598700000000000),
('S', 0.0632700000000000),
('T', 0.0905600000000000),
('U', 0.0275800000000000),
('V', 0.00978000000000000),
('W', 0.0236000000000000),
('X', 0.00150000000000000),
('Y', 0.0197400000000000),
('Z', 0.000740000000000000)]
```

Illustrating the difference between *characteristic_frequency()* and *frequency_distribution()*:

```
sage: A = AlphabeticStrings()
sage: M = A.encoding("abcd")
sage: FD = M.frequency_distribution().function()
sage: sorted(FD.items())
```

```
[(A, 0.250000000000000),
 (B, 0.250000000000000),
 (C, 0.250000000000000),
 (D, 0.250000000000000)]
sage: CF = A.characteristic_frequency()
sage: sorted(CF.items())

[('A', 0.0820000000000000),
 ('B', 0.0150000000000000),
 ('C', 0.0280000000000000),
 ('D', 0.0430000000000000),
 ('E', 0.127000000000000),
 ('F', 0.0220000000000000),
 ('G', 0.0200000000000000),
 ('H', 0.0610000000000000),
 ('I', 0.0700000000000000),
 ('J', 0.0020000000000000),
 ('K', 0.0080000000000000),
 ('L', 0.0400000000000000),
 ('M', 0.0240000000000000),
 ('N', 0.0670000000000000),
 ('O', 0.0750000000000000),
 ('P', 0.0190000000000000),
 ('Q', 0.0010000000000000),
 ('R', 0.0600000000000000),
 ('S', 0.0630000000000000),
 ('T', 0.0910000000000000),
 ('U', 0.0280000000000000),
 ('V', 0.0100000000000000),
 ('W', 0.0230000000000000),
 ('X', 0.0010000000000000),
 ('Y', 0.0200000000000000),
 ('Z', 0.0010000000000000)]
```

```
>>> from sage.all import *
>>> A = AlphabeticStrings()
>>> M = A.encoding("abcd")
>>> FD = M.frequency_distribution().function()
>>> sorted(FD.items())
<BLANKLINE>
[(A, 0.250000000000000),
 (B, 0.250000000000000),
 (C, 0.250000000000000),
 (D, 0.250000000000000)]
>>> CF = A.characteristic_frequency()
>>> sorted(CF.items())
<BLANKLINE>
[('A', 0.0820000000000000),
 ('B', 0.0150000000000000),
 ('C', 0.0280000000000000),
 ('D', 0.0430000000000000),
```

```
('E', 0.127000000000000),
('F', 0.0220000000000000),
('G', 0.0200000000000000),
('H', 0.0610000000000000),
('I', 0.0700000000000000),
('J', 0.00200000000000000),
('K', 0.00800000000000000),
('L', 0.0400000000000000),
('M', 0.0240000000000000),
('N', 0.0670000000000000),
('O', 0.0750000000000000),
('P', 0.0190000000000000),
('Q', 0.00100000000000000),
('R', 0.0600000000000000),
('S', 0.0630000000000000),
('T', 0.0910000000000000),
('U', 0.0280000000000000),
('V', 0.0100000000000000),
('W', 0.0230000000000000),
('X', 0.00100000000000000),
('Y', 0.0200000000000000),
('Z', 0.00100000000000000)]
```

encoding(S)

> The encoding of the string S in the alphabetic string monoid, obtained by the monoid homomorphism

```
A -> A, ..., Z -> Z, a -> A, ..., z -> Z
```

> and stripping away all other characters. It should be noted that this is a non-injective monoid homomorphism.
>
> EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: s = S.encoding("The cat in the hat."); s
THECATINTHEHAT
sage: s.decoding()
'THECATINTHEHAT'
```

```
>>> from sage.all import *
>>> S = AlphabeticStrings()
>>> s = S.encoding("The cat in the hat."); s
THECATINTHEHAT
>>> s.decoding()
'THECATINTHEHAT'
```

sage.monoids.string_monoid.**AlphabeticStrings**

> alias of *AlphabeticStringMonoid*

class sage.monoids.string_monoid.**BinaryStringMonoid**

> Bases: *StringMonoid_class*
>
> The free binary string monoid on generators $\{0, 1\}$.

---

encoding(*S*, *padic=False*)

> The binary encoding of the string `S`, as a binary string element.

> The default is to keep the standard ASCII byte encoding, e.g.

```
A = 65 -> 01000001
B = 66 -> 01000010
.
.
.
Z = 90 -> 01001110
```

> rather than a 2-adic representation 65 -> 10000010.

> Set `padic=True` to reverse the bit string.

> EXAMPLES:

```
sage: S = BinaryStrings()
sage: S.encoding('A')
01000001
sage: S.encoding('A',padic=True)
10000010
sage: S.encoding(' ',padic=True)
00000100
```

```
>>> from sage.all import *
>>> S = BinaryStrings()
>>> S.encoding('A')
01000001
>>> S.encoding('A',padic=True)
10000010
>>> S.encoding(' ',padic=True)
00000100
```

sage.monoids.string_monoid.**BinaryStrings**

> alias of *BinaryStringMonoid*

class sage.monoids.string_monoid.**HexadecimalStringMonoid**

> Bases: *StringMonoid_class*

> The free hexadecimal string monoid on generators $\{0, 1, \ldots, 9, a, b, c, d, e, f\}$.

> **encoding**(*S*, *padic=False*)

>> The encoding of the string `S` as a hexadecimal string element.

>> The default is to keep the standard right-to-left byte encoding, e.g.

```
A = '\x41' -> 41
B = '\x42' -> 42
.
.
.
Z = '\x5a' -> 5a
```

> rather than a left-to-right representation A = 65 -> 14. Although standard (e.g., in the Python constructor 'xhh'), this can be confusing when the string reads left-to-right.

Set `padic=True` to reverse the character encoding.

EXAMPLES:

```
sage: S = HexadecimalStrings()
sage: S.encoding('A')
41
sage: S.encoding('A',padic=True)
14
sage: S.encoding(' ',padic=False)
20
sage: S.encoding(' ',padic=True)
02
```

```
>>> from sage.all import *
>>> S = HexadecimalStrings()
>>> S.encoding('A')
41
>>> S.encoding('A',padic=True)
14
>>> S.encoding(' ',padic=False)
20
>>> S.encoding(' ',padic=True)
02
```

sage.monoids.string_monoid.**HexadecimalStrings**

>   alias of *HexadecimalStringMonoid*

class sage.monoids.string_monoid.**OctalStringMonoid**

>   Bases: *StringMonoid_class*
>
>   The free octal string monoid on generators $\{0, 1, \ldots, 7\}$.

sage.monoids.string_monoid.**OctalStrings**

>   alias of *OctalStringMonoid*

class sage.monoids.string_monoid.**Radix64StringMonoid**

>   Bases: *StringMonoid_class*
>
>   The free radix 64 string monoid on 64 generators.

sage.monoids.string_monoid.**Radix64Strings**

>   alias of *Radix64StringMonoid*

class sage.monoids.string_monoid.**StringMonoid_class**(*n*, *alphabet=()*)

>   Bases: *FreeMonoid*
>
>   A free string monoid on n generators.
>
>   **alphabet**()
>
>   **gen**(*i=0*)
>
> >   The i-th generator of the monoid.
> >
> >   INPUT:
> >
> >   • i – integer (default: 0)
> >
> >   EXAMPLES:

```
sage: S = BinaryStrings()
sage: S.gen(0)
0
sage: S.gen(1)
1
sage: S.gen(2)
Traceback (most recent call last):
...
IndexError: Argument i (= 2) must be between 0 and 1.
sage: S = HexadecimalStrings()
sage: S.gen(0)
0
sage: S.gen(12)
c
sage: S.gen(16)
Traceback (most recent call last):
...
IndexError: Argument i (= 16) must be between 0 and 15.
```

```
>>> from sage.all import *
>>> S = BinaryStrings()
>>> S.gen(Integer(0))
0
>>> S.gen(Integer(1))
1
>>> S.gen(Integer(2))
Traceback (most recent call last):
...
IndexError: Argument i (= 2) must be between 0 and 1.
>>> S = HexadecimalStrings()
>>> S.gen(Integer(0))
0
>>> S.gen(Integer(12))
c
>>> S.gen(Integer(16))
Traceback (most recent call last):
...
IndexError: Argument i (= 16) must be between 0 and 15.
```

one()

    Return the identity element of `self`.

    EXAMPLES:

```
sage: b = BinaryStrings(); b
Free binary string monoid
sage: b.one() * b('1011')
1011
sage: b.one() * b('110') == b('110')
True
sage: b('10101') * b.one() == b('101011')
False
```

```
>>> from sage.all import *
>>> b = BinaryStrings(); b
Free binary string monoid
>>> b.one() * b('1011')
1011
>>> b.one() * b('110') == b('110')
True
>>> b('10101') * b.one() == b('101011')
False
```

STRING MONOID ELEMENTS

AUTHORS:

- David Kohel <kohel@maths.usyd.edu.au>, 2007-01

Elements of free string monoids, internal representation subject to change.

These are special classes of free monoid elements with distinct printing.

The internal representation of elements does not use the exponential compression of FreeMonoid elements (a feature), and could be packed into words.

class sage.monoids.string_monoid_element.**StringMonoidElement**(*S*, *x*, *check=True*)

Bases: *FreeMonoidElement*

Element of a free string monoid.

character_count()

Return the count of each unique character.

EXAMPLES:

Count the character frequency in an object comprised of capital letters of the English alphabet:

```
sage: M = AlphabeticStrings().encoding("abcabf")
sage: sorted(M.character_count().items())
[(A, 2), (B, 2), (C, 1), (F, 1)]
```

```
>>> from sage.all import *
>>> M = AlphabeticStrings().encoding("abcabf")
>>> sorted(M.character_count().items())
[(A, 2), (B, 2), (C, 1), (F, 1)]
```

In an object comprised of binary numbers:

```
sage: M = BinaryStrings().encoding("abcabf")
sage: sorted(M.character_count().items())
[(0, 28), (1, 20)]
```

```
>>> from sage.all import *
>>> M = BinaryStrings().encoding("abcabf")
>>> sorted(M.character_count().items())
[(0, 28), (1, 20)]
```

In an object comprised of octal numbers:

```
sage: A = OctalStrings()
sage: M = A([1, 2, 3, 2, 5, 3])
sage: sorted(M.character_count().items())
[(1, 1), (2, 2), (3, 2), (5, 1)]
```

```
>>> from sage.all import *
>>> A = OctalStrings()
>>> M = A([Integer(1), Integer(2), Integer(3), Integer(2), Integer(5),␣
→Integer(3)])
>>> sorted(M.character_count().items())
[(1, 1), (2, 2), (3, 2), (5, 1)]
```

In an object comprised of hexadecimal numbers:

```
sage: A = HexadecimalStrings()
sage: M = A([1, 2, 4, 6, 2, 4, 15])
sage: sorted(M.character_count().items())
[(1, 1), (2, 2), (4, 2), (6, 1), (f, 1)]
```

```
>>> from sage.all import *
>>> A = HexadecimalStrings()
>>> M = A([Integer(1), Integer(2), Integer(4), Integer(6), Integer(2),␣
→Integer(4), Integer(15)])
>>> sorted(M.character_count().items())
[(1, 1), (2, 2), (4, 2), (6, 1), (f, 1)]
```

In an object comprised of radix-64 characters:

```
sage: A = Radix64Strings()
sage: M = A([1, 2, 63, 45, 45, 10]); M
BC/ttK
sage: sorted(M.character_count().items())
[(B, 1), (C, 1), (K, 1), (t, 2), (/, 1)]
```

```
>>> from sage.all import *
>>> A = Radix64Strings()
>>> M = A([Integer(1), Integer(2), Integer(63), Integer(45), Integer(45),␣
→Integer(10)]); M
BC/ttK
>>> sorted(M.character_count().items())
[(B, 1), (C, 1), (K, 1), (t, 2), (/, 1)]
```

coincidence_index(*prec=0*)

Return the probability of two randomly chosen characters being equal.

decoding(*padic=False*)

The byte string associated to a binary or hexadecimal string monoid element.

EXAMPLES:

```
sage: S = HexadecimalStrings()
sage: s = S.encoding("A..Za..z"); s
412e2e5a612e2e7a
```

(continues on next page)

```
sage: s.decoding()
'A..Za..z'
sage: s = S.encoding("A..Za..z",padic=True); s
14e2e2a516e2e2a7
sage: s.decoding()
'\x14\xe2\xe2\xa5\x16\xe2\xe2\xa7'
sage: s.decoding(padic=True)
'A..Za..z'
sage: S = BinaryStrings()
sage: s = S.encoding("A..Za..z"); s
01000001001011100010111001011010011000010010111000101110011111010
sage: s.decoding()
'A..Za..z'
sage: s = S.encoding("A..Za..z",padic=True); s
10000010011101000111010001011010100001100111010001110100010111110
sage: s.decoding()
'\x82ttZ\x86tt^'
sage: s.decoding(padic=True)
'A..Za..z'
```

```
>>> from sage.all import *
>>> S = HexadecimalStrings()
>>> s = S.encoding("A..Za..z"); s
412e2e5a612e2e7a
>>> s.decoding()
'A..Za..z'
>>> s = S.encoding("A..Za..z",padic=True); s
14e2e2a516e2e2a7
>>> s.decoding()
'\x14\xe2\xe2\xa5\x16\xe2\xe2\xa7'
>>> s.decoding(padic=True)
'A..Za..z'
>>> S = BinaryStrings()
>>> s = S.encoding("A..Za..z"); s
01000001001011100010111001011010011000010010111000101110011111010
>>> s.decoding()
'A..Za..z'
>>> s = S.encoding("A..Za..z",padic=True); s
10000010011101000111010001011010100001100111010001110100010111110
>>> s.decoding()
'\x82ttZ\x86tt^'
>>> s.decoding(padic=True)
'A..Za..z'
```

frequency_distribution(*length=1*, *prec=0*)

Return the probability space of character frequencies.

The output of this method is different from that of the method *characteristic_frequency()*.

One can think of the characteristic frequency probability of an element in an alphabet A as the expected probability of that element occurring. Let S be a string encoded using elements of A. The frequency probability distribution corresponding to S provides us with the frequency probability of each element of A as observed occurring in S. Thus one distribution provides expected probabilities, while the other provides observed

probabilities.

INPUT:

- `length` – (default: `1`) if `length=1` then consider the probability space of monogram frequency, i.e. probability distribution of single characters. If `length=2` then consider the probability space of digram frequency, i.e. probability distribution of pairs of characters. This method currently supports the generation of probability spaces for monogram frequency (`length=1`) and digram frequency (`length=2`).

- `prec` – (default: `0`) a nonnegative integer representing the precision (in number of bits) of a floating-point number. The default value `prec=0` means that we use 53 bits to represent the mantissa of a floating-point number. For more information on the precision of floating-point numbers, see the function `Real-Field()` or refer to the module `real_mpfr`.

EXAMPLES:

Capital letters of the English alphabet:

```
sage: M = AlphabeticStrings().encoding("abcd")
sage: L = M.frequency_distribution().function()
sage: sorted(L.items())

[(A, 0.250000000000000),
(B, 0.250000000000000),
(C, 0.250000000000000),
(D, 0.250000000000000)]
```

```
>>> from sage.all import *
>>> M = AlphabeticStrings().encoding("abcd")
>>> L = M.frequency_distribution().function()
>>> sorted(L.items())
<BLANKLINE>
[(A, 0.250000000000000),
(B, 0.250000000000000),
(C, 0.250000000000000),
(D, 0.250000000000000)]
```

The binary number system:

```
sage: M = BinaryStrings().encoding("abcd")
sage: L = M.frequency_distribution().function()
sage: sorted(L.items())
[(0, 0.593750000000000), (1, 0.406250000000000)]
```

```
>>> from sage.all import *
>>> M = BinaryStrings().encoding("abcd")
>>> L = M.frequency_distribution().function()
>>> sorted(L.items())
[(0, 0.593750000000000), (1, 0.406250000000000)]
```

The hexadecimal number system:

```
sage: M = HexadecimalStrings().encoding("abcd")
sage: L = M.frequency_distribution().function()
sage: sorted(L.items())
```

```
[(1, 0.125000000000000),
(2, 0.125000000000000),
(3, 0.125000000000000),
(4, 0.125000000000000),
(6, 0.500000000000000)]
```

```
>>> from sage.all import *
>>> M = HexadecimalStrings().encoding("abcd")
>>> L = M.frequency_distribution().function()
>>> sorted(L.items())
<BLANKLINE>
[(1, 0.125000000000000),
(2, 0.125000000000000),
(3, 0.125000000000000),
(4, 0.125000000000000),
(6, 0.500000000000000)]
```

Get the observed frequency probability distribution of digrams in the string "ABCD". This string consists of the following digrams: "AB", "BC", and "CD". Now find out the frequency probability of each of these digrams as they occur in the string "ABCD":

```
sage: M = AlphabeticStrings().encoding("abcd")
sage: D = M.frequency_distribution(length=2).function()
sage: sorted(D.items())
[(AB, 0.333333333333333), (BC, 0.333333333333333), (CD, 0.333333333333333)]
```

```
>>> from sage.all import *
>>> M = AlphabeticStrings().encoding("abcd")
>>> D = M.frequency_distribution(length=Integer(2)).function()
>>> sorted(D.items())
[(AB, 0.333333333333333), (BC, 0.333333333333333), (CD, 0.333333333333333)]
```

sage.monoids.string_monoid_element.**is_AlphabeticStringMonoidElement**(x)

sage.monoids.string_monoid_element.**is_BinaryStringMonoidElement**(x)

sage.monoids.string_monoid_element.**is_HexadecimalStringMonoidElement**(x)

sage.monoids.string_monoid_element.**is_OctalStringMonoidElement**(x)

sage.monoids.string_monoid_element.**is_Radix64StringMonoidElement**(x)

sage.monoids.string_monoid_element.**is_StringMonoidElement**(x)

NINE

UTILITY FUNCTIONS ON STRINGS

sage.monoids.string_ops.**coincidence_discriminant**(*S*, *n=2*)

> INPUT:
>
> > • S –tuple of strings; e.g. produced as decimation of transposition ciphertext, or a sample plaintext
>
> OUTPUT:
>
> A measure of the difference of probability of association of character pairs, relative to their independent one-character probabilities.
>
> EXAMPLES:

```
sage: S = strip_encoding("The cat in the hat.")
sage: coincidence_discriminant([ S[i:i+2] for i in range(len(S)-1) ])
0.0827001855677322
```

```
>>> from sage.all import *
>>> S = strip_encoding("The cat in the hat.")
>>> coincidence_discriminant([ S[i:i+Integer(2)] for i in range(len(S)-
↪Integer(1)) ])
0.0827001855677322
```

sage.monoids.string_ops.**coincidence_index**(*S*, *n=1*)

> Return the coincidence index of the string S.
>
> EXAMPLES:

```
sage: S = strip_encoding("The cat in the hat.")
sage: coincidence_index(S)
0.120879120879121
```

```
>>> from sage.all import *
>>> S = strip_encoding("The cat in the hat.")
>>> coincidence_index(S)
0.120879120879121
```

sage.monoids.string_ops.**frequency_distribution**(*S*, *n=1*, *field=None*)

> The probability space of frequencies of n-character substrings of S.
>
> EXAMPLES:

```
sage: frequency_distribution('banana not a nana nor ananas', 2)
Discrete probability space defined by {' a': 0.0740740740740741,
 ' n': 0.111111111111111,
 'a ': 0.111111111111111,
 'an': 0.185185185185185,
 'as': 0.0370370370370370,
 'ba': 0.0370370370370370,
 'na': 0.222222222222222,
 'no': 0.0740740740740741,
 'or': 0.0370370370370370,
 'ot': 0.0370370370370370,
 'r ': 0.0370370370370370,
 't ': 0.0370370370370370}
```

```
>>> from sage.all import *
>>> frequency_distribution('banana not a nana nor ananas', Integer(2))
Discrete probability space defined by {' a': 0.0740740740740741,
 ' n': 0.111111111111111,
 'a ': 0.111111111111111,
 'an': 0.185185185185185,
 'as': 0.0370370370370370,
 'ba': 0.0370370370370370,
 'na': 0.222222222222222,
 'no': 0.0740740740740741,
 'or': 0.0370370370370370,
 'ot': 0.0370370370370370,
 'r ': 0.0370370370370370,
 't ': 0.0370370370370370}
```

sage.monoids.string_ops.**strip_encoding**(S)

Return the upper case string of S stripped of all non-alphabetic characters.

EXAMPLES:

```
sage: S = "The cat in the hat."
sage: strip_encoding(S)
'THECATINTHEHAT'
```

```
>>> from sage.all import *
>>> S = "The cat in the hat."
>>> strip_encoding(S)
'THECATINTHEHAT'
```

HECKE MONOIDS

sage.monoids.hecke_monoid.**HeckeMonoid**()

Return the 0-Hecke monoid of the Coxeter group W.

INPUT:

- `W` – a finite Coxeter group

Let $s_1, \ldots, s_n$ be the simple reflections of W. The 0-Hecke monoid is the monoid generated by projections $\pi_1, \ldots, \pi_n$ satisfying the same braid and commutation relations as the s_i. It is of same cardinality as W.

> **ⓘ Note**
>
> This is currently a very basic implementation as the submonoid of sorting maps on W generated by the simple projections of W. It's only functional for W finite.

> **↪ See also**
>
> - CoxeterGroups
>
> - CoxeterGroups.ParentMethods.simple_projections
>
> - IwahoriHeckeAlgebra

EXAMPLES:

```
sage: from sage.monoids.hecke_monoid import HeckeMonoid
sage: W = SymmetricGroup(4)
sage: H = HeckeMonoid(W); H
0-Hecke monoid of the Symmetric group of order 4! as a permutation group
sage: pi = H.monoid_generators(); pi
Finite family {1: ..., 2: ..., 3: ...}
sage: all(pi[i]^2 == pi[i] for i in pi.keys())
True
sage: pi[1] * pi[2] * pi[1] == pi[2] * pi[1] * pi[2]
True
sage: pi[2] * pi[3] * pi[2] == pi[3] * pi[2] * pi[3]
True
sage: pi[1] * pi[3] == pi[3] * pi[1]
True
```

```
sage: H.cardinality()
24
```

```
>>> from sage.all import *
>>> from sage.monoids.hecke_monoid import HeckeMonoid
>>> W = SymmetricGroup(Integer(4))
>>> H = HeckeMonoid(W); H
0-Hecke monoid of the Symmetric group of order 4! as a permutation group
>>> pi = H.monoid_generators(); pi
Finite family {1: ..., 2: ..., 3: ...}
>>> all(pi[i]**Integer(2) == pi[i] for i in pi.keys())
True
>>> pi[Integer(1)] * pi[Integer(2)] * pi[Integer(1)] == pi[Integer(2)] *␣
↪pi[Integer(1)] * pi[Integer(2)]
True
>>> pi[Integer(2)] * pi[Integer(3)] * pi[Integer(2)] == pi[Integer(3)] *␣
↪pi[Integer(2)] * pi[Integer(3)]
True
>>> pi[Integer(1)] * pi[Integer(3)] == pi[Integer(3)] * pi[Integer(1)]
True
>>> H.cardinality()
24
```

AUTOMATIC SEMIGROUPS

Semigroups defined by generators living in an ambient semigroup and represented by an automaton.

AUTHORS:

- Nicolas M. Thiéry

- Aladin Virmaux

class sage.monoids.automatic_semigroup.**AutomaticMonoid**(*generators*, *ambient*, *one*, *mul*, *category*)

Bases: *AutomaticSemigroup*

gens()

Return the family of monoid generators of self.

EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(28)
sage: M = R.submonoid(Family({1: R(3), 2: R(5)}))
sage: M.monoid_generators()
Finite family {1: 3, 2: 5}
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(28))
>>> M = R.submonoid(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}))
>>> M.monoid_generators()
Finite family {1: 3, 2: 5}
```

Note that the monoid generators do not include the unit, unlike the semigroup generators:

```
sage: M.semigroup_generators()
Family (1, 3, 5)
```

```
>>> from sage.all import *
>>> M.semigroup_generators()
Family (1, 3, 5)
```

monoid_generators()

Return the family of monoid generators of self.

EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(28)
sage: M = R.submonoid(Family({1: R(3), 2: R(5)}))
sage: M.monoid_generators()
Finite family {1: 3, 2: 5}
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(28))
>>> M = R.submonoid(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}))
>>> M.monoid_generators()
Finite family {1: 3, 2: 5}
```

Note that the monoid generators do not include the unit, unlike the semigroup generators:

```
sage: M.semigroup_generators()
Family (1, 3, 5)
```

```
>>> from sage.all import *
>>> M.semigroup_generators()
Family (1, 3, 5)
```

one()

> Return the unit of `self`.
>
> EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(21)
sage: M = R.submonoid(())
sage: M.one()
1
sage: M.one().parent() is M
True
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(21))
>>> M = R.submonoid(())
>>> M.one()
1
>>> M.one().parent() is M
True
```

semigroup_generators()

> Return the generators of `self` as a semigroup.
>
> The generators of a monoid M as a semigroup are the generators of M as a monoid and the unit.
>
> EXAMPLES:

```
sage: M = Monoids().free([1,2,3])                                          #␣
↪needs sage.combinat
sage: M.semigroup_generators()                                            #␣
↪needs sage.combinat
Family (1, F[1], F[2], F[3])
```

```
>>> from sage.all import *
>>> M = Monoids().free([Integer(1),Integer(2),Integer(3)])                ␣
↪                      # needs sage.combinat
>>> M.semigroup_generators()                                          # needs␣
↪sage.combinat
Family (1, F[1], F[2], F[3])
```

class sage.monoids.automatic_semigroup.**AutomaticSemigroup**(*generators*, *ambient*, *one*, *mul*,
                                                              category)

Bases: `UniqueRepresentation`, `Parent`

Semigroups defined by generators living in an ambient semigroup.

This implementation lazily constructs all the elements of the semigroup, and the right Cayley graph relations between them, and uses the latter as an automaton.

EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(12)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: M in Monoids()
True
sage: M.one()
1
sage: M.one() in M
True
sage: g = M._generators; g
Finite family {1: 3, 2: 5}
sage: g[1]*g[2]
3
sage: M.some_elements()
[1, 3, 5, 9]

sage: M.list()
[1, 3, 5, 9]

sage: M.idempotents()
[1, 9]
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(12))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}), one=R.one())
>>> M in Monoids()
True
```

```
>>> M.one()
1
>>> M.one() in M
True
>>> g = M._generators; g
Finite family {1: 3, 2: 5}
>>> g[Integer(1)]*g[Integer(2)]
3
>>> M.some_elements()
[1, 3, 5, 9]

>>> M.list()
[1, 3, 5, 9]

>>> M.idempotents()
[1, 9]
```

As can be seen above, elements are represented by default the corresponding element in the ambient monoid. One can also represent the elements by their reduced word:

```
sage: M.repr_element_method("reduced_word")
sage: M.list()
[[], [1], [2], [1, 1]]
```

```
>>> from sage.all import *
>>> M.repr_element_method("reduced_word")
>>> M.list()
[[], [1], [2], [1, 1]]
```

In case the reduced word has not yet been calculated, the element will be represented by the corresponding element in the ambient monoid:

```
sage: R = IntegerModRing(13)
sage: N = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: N.repr_element_method("reduced_word")
sage: n = N.an_element()
sage: n
[1]
sage: n*n
9
```

```
>>> from sage.all import *
>>> R = IntegerModRing(Integer(13))
>>> N = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}), one=R.one())
>>> N.repr_element_method("reduced_word")
>>> n = N.an_element()
>>> n
[1]
>>> n*n
9
```

Calling *construct()*, *cardinality()*, or *list()*, or iterating through the monoid will trigger its full con-

struction and, as a side effect, compute all the reduced words. The order of the elements, and the induced choice of reduced word is currently length-lexicographic (i.e. the chosen reduced word is of minimal length, and then minimal lexicographically w.r.t. the order of the indices of the generators):

```
sage: M.cardinality()
4
sage: M.list()
[[], [1], [2], [1, 1]]
sage: g = M._generators

sage: g[1]*g[2]
[1]

sage: g[1].transition(1)
[1, 1]
sage: g[1] * g[1]
[1, 1]
sage: g[1] * g[1] * g[1]
[1]
sage: g[1].transition(2)
[1]
sage: g[1] * g[2]
[1]

sage: [ x.lift() for x in M.list() ]
[1, 3, 5, 9]

sage: # needs sage.graphs
sage: G = M.cayley_graph(side='twosided'); G
Looped multi-digraph on 4 vertices
sage: G.edges(sort=True, key=str)
[([1, 1], [1, 1], (2, 'left')),
 ([1, 1], [1, 1], (2, 'right')),
 ([1, 1], [1], (1, 'left')),
 ([1, 1], [1], (1, 'right')),
 ([1], [1, 1], (1, 'left')),
 ([1], [1, 1], (1, 'right')),
 ([1], [1], (2, 'left')),
 ([1], [1], (2, 'right')),
 ([2], [1], (1, 'left')),
 ([2], [1], (1, 'right')),
 ([2], [], (2, 'left')),
 ([2], [], (2, 'right')),
 ([], [1], (1, 'left')),
 ([], [1], (1, 'right')),
 ([], [2], (2, 'left')),
 ([], [2], (2, 'right'))]
sage: list(map(sorted, M.j_classes()))
[[[1], [1, 1]], [[], [2]]]
sage: M.j_classes_of_idempotents()
[[[1, 1]], [[]]]
sage: M.j_transversal_of_idempotents()
[[1, 1], []]
```

```
sage: list(map(attrcall('pseudo_order'), M.list()))                           #␣
↪needs sage.graphs
[[1, 0], [3, 1], [2, 0], [2, 1]]
```

```
>>> from sage.all import *
>>> M.cardinality()
4
>>> M.list()
[[], [1], [2], [1, 1]]
>>> g = M._generators

>>> g[Integer(1)]*g[Integer(2)]
[1]

>>> g[Integer(1)].transition(Integer(1))
[1, 1]
>>> g[Integer(1)] * g[Integer(1)]
[1, 1]
>>> g[Integer(1)] * g[Integer(1)] * g[Integer(1)]
[1]
>>> g[Integer(1)].transition(Integer(2))
[1]
>>> g[Integer(1)] * g[Integer(2)]
[1]

>>> [ x.lift() for x in M.list() ]
[1, 3, 5, 9]

>>> # needs sage.graphs
>>> G = M.cayley_graph(side='twosided'); G
Looped multi-digraph on 4 vertices
>>> G.edges(sort=True, key=str)
[([1, 1], [1, 1], (2, 'left')),
 ([1, 1], [1, 1], (2, 'right')),
 ([1, 1], [1], (1, 'left')),
 ([1, 1], [1], (1, 'right')),
 ([1], [1, 1], (1, 'left')),
 ([1], [1, 1], (1, 'right')),
 ([1], [1], (2, 'left')),
 ([1], [1], (2, 'right')),
 ([2], [1], (1, 'left')),
 ([2], [1], (1, 'right')),
 ([2], [], (2, 'left')),
 ([2], [], (2, 'right')),
 ([], [1], (1, 'left')),
 ([], [1], (1, 'right')),
 ([], [2], (2, 'left')),
 ([], [2], (2, 'right'))]
>>> list(map(sorted, M.j_classes()))
[[[1], [1, 1]], [[], [2]]]
>>> M.j_classes_of_idempotents()
```

```
[[[1, 1]], [[]]]
>>> M.j_transversal_of_idempotents()
[[1, 1], []]

>>> list(map(attrcall('pseudo_order'), M.list()))                            #␣
↪needs sage.graphs
[[1, 0], [3, 1], [2, 0], [2, 1]]
```

We can also use it to get submonoids from groups. We check that in the symmetric group, a transposition and a long cycle generate the whole group:

```
sage: # needs sage.groups
sage: G5 = SymmetricGroup(5)
sage: N = AutomaticSemigroup(Family({1: G5([2,1,3,4,5]), 2: G5([2,3,4,5,1])}),
....:                         one=G5.one())
sage: N.repr_element_method("reduced_word")
sage: N.cardinality() == G5.cardinality()
True
sage: N.retract(G5((1,4,3,5,2)))
[1, 2, 1, 2, 2, 1, 2, 1, 2, 2]
sage: N.from_reduced_word([1, 2, 1, 2, 2, 1, 2, 1, 2, 2]).lift()
(1,4,3,5,2)
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> G5 = SymmetricGroup(Integer(5))
>>> N = AutomaticSemigroup(Family({Integer(1): G5([Integer(2),Integer(1),
↪Integer(3),Integer(4),Integer(5)]), Integer(2): G5([Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(1)])}),
...                         one=G5.one())
>>> N.repr_element_method("reduced_word")
>>> N.cardinality() == G5.cardinality()
True
>>> N.retract(G5((Integer(1),Integer(4),Integer(3),Integer(5),Integer(2))))
[1, 2, 1, 2, 2, 1, 2, 1, 2, 2]
>>> N.from_reduced_word([Integer(1), Integer(2), Integer(1), Integer(2),␣
↪Integer(2), Integer(1), Integer(2), Integer(1), Integer(2), Integer(2)]).lift()
(1,4,3,5,2)
```

We can also create a semigroup of matrices, where we define the multiplication as matrix multiplication:

```
sage: # needs sage.modules
sage: M1 = matrix([[0,0,1],[1,0,0],[0,1,0]])
sage: M2 = matrix([[0,0,0],[1,1,0],[0,0,1]])
sage: M1.set_immutable()
sage: M2.set_immutable()
sage: def prod_m(x, y):
....:     z=x*y
....:     z.set_immutable()
....:     return z
sage: Mon = AutomaticSemigroup([M1,M2], mul=prod_m,
....:                         category=Monoids().Finite().Subobjects())
```

```
sage: Mon.cardinality()
24
sage: C = Mon.cayley_graph()                                              #␣
→needs sage.graphs
sage: C.is_directed_acyclic()                                            #␣
→needs sage.graphs
False
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> M1 = matrix([[Integer(0),Integer(0),Integer(1)],[Integer(1),Integer(0),
→Integer(0)],[Integer(0),Integer(1),Integer(0)]])
>>> M2 = matrix([[Integer(0),Integer(0),Integer(0)],[Integer(1),Integer(1),
→Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> M1.set_immutable()
>>> M2.set_immutable()
>>> def prod_m(x, y):
...     z=x*y
...     z.set_immutable()
...     return z
>>> Mon = AutomaticSemigroup([M1,M2], mul=prod_m,
...                          category=Monoids().Finite().Subobjects())
>>> Mon.cardinality()
24
>>> C = Mon.cayley_graph()                                                #␣
→needs sage.graphs
>>> C.is_directed_acyclic()                                               #␣
→needs sage.graphs
False
```

Let us construct and play with the 0-Hecke Monoid:

```
sage: # needs sage.graphs sage.modules
sage: W = WeylGroup(['A',4]); W.rename('W')
sage: ambient_monoid = FiniteSetMaps(W, action='right')
sage: pi = W.simple_projections(length_increasing=True).map(ambient_monoid)
sage: M = AutomaticSemigroup(pi, one=ambient_monoid.one()); M
A submonoid of (Maps from W to itself) with 4 generators
sage: M.repr_element_method("reduced_word")
sage: sorted(M._elements_set, key=str)
[[1], [2], [3], [4], []]
sage: M.construct(n=10)
sage: sorted(M._elements_set, key=str)
[[1, 2], [1, 3], [1, 4], [1], [2, 1], [2, 3], [2], [3], [4], []]
sage: elt = M.from_reduced_word([3,1,2,4,2])
sage: M.construct(up_to=elt)
sage: len(M._elements_set)
36
sage: M.cardinality()
120
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.modules
>>> W = WeylGroup(['A',Integer(4)]); W.rename('W')
>>> ambient_monoid = FiniteSetMaps(W, action='right')
>>> pi = W.simple_projections(length_increasing=True).map(ambient_monoid)
>>> M = AutomaticSemigroup(pi, one=ambient_monoid.one()); M
A submonoid of (Maps from W to itself) with 4 generators
>>> M.repr_element_method("reduced_word")
>>> sorted(M._elements_set, key=str)
[[1], [2], [3], [4], []]
>>> M.construct(n=Integer(10))
>>> sorted(M._elements_set, key=str)
[[1, 2], [1, 3], [1, 4], [1], [2, 1], [2, 3], [2], [3], [4], []]
>>> elt = M.from_reduced_word([Integer(3),Integer(1),Integer(2),Integer(4),
↪Integer(2)])
>>> M.construct(up_to=elt)
>>> len(M._elements_set)
36
>>> M.cardinality()
120
```

We check that the 0-Hecke monoid is J-trivial and contains 2^4 idempotents:

```
sage: len(M.idempotents())                                              #␣
↪needs sage.graphs sage.modules
16
sage: all(len(j) == 1 for j in M.j_classes())                           #␣
↪needs sage.graphs sage.modules
True
```

```
>>> from sage.all import *
>>> len(M.idempotents())                                                #␣
↪needs sage.graphs sage.modules
16
>>> all(len(j) == Integer(1) for j in M.j_classes())                    ␣
↪       # needs sage.graphs sage.modules
True
```

> **ⓘ Note**
>
> Unlike what the name of the class may suggest, this currently implements only a subclass of automatic semi-groups; essentially the finite ones. See Wikipedia article Automatic_semigroup.

> **⚠ Warning**
>
> *AutomaticSemigroup* is designed primarily for finite semigroups. This property is not checked automatically (this would be too costly, if not undecidable). Use with care for an infinite semigroup, as certain features may require constructing all of it:

```
sage: M = AutomaticSemigroup([2], category = Monoids().Subobjects()); M
A submonoid of (Integer Ring) with 1 generators
sage: M.retract(2)
2
sage: M.retract(3)    # not tested: runs forever trying to find 3
```

```
>>> from sage.all import *
>>> M = AutomaticSemigroup([Integer(2)], category = Monoids().Subobjects()); M
A submonoid of (Integer Ring) with 1 generators
>>> M.retract(Integer(2))
2
>>> M.retract(Integer(3))    # not tested: runs forever trying to find 3
```

class Element(*ambient_element*, *parent*)

> Bases: `ElementWrapper`

> **lift**()

>> Lift the element `self` into its ambient semigroup.

>> EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(18)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}))
sage: M.repr_element_method("reduced_word")
sage: m = M.an_element(); m
[1]
sage: type(m)
<class 'sage.monoids.automatic_semigroup.AutomaticSemigroup_with_category.
↪element_class'>
sage: m.lift()
3
sage: type(m.lift())
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(18))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}))
>>> M.repr_element_method("reduced_word")
>>> m = M.an_element(); m
[1]
>>> type(m)
<class 'sage.monoids.automatic_semigroup.AutomaticSemigroup_with_category.
↪element_class'>
>>> m.lift()
3
>>> type(m.lift())
<class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
```

> **reduced_word**()

>> Return the length-lexicographic shortest word of `self`.

OUTPUT: list of indexes of the generators

Obtaining the reduced word requires having constructed the Cayley graph of the semigroup up to `self`.
If this is not the case, an error is raised.

EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(15)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: M.construct()
sage: for m in M: print((m, m.reduced_word()))
(1, [])
(3, [1])
(5, [2])
(9, [1, 1])
(0, [1, 2])
(10, [2, 2])
(12, [1, 1, 1])
(6, [1, 1, 1, 1])
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(15))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}), one=R.one())
>>> M.construct()
>>> for m in M: print((m, m.reduced_word()))
(1, [])
(3, [1])
(5, [2])
(9, [1, 1])
(0, [1, 2])
(10, [2, 2])
(12, [1, 1, 1])
(6, [1, 1, 1, 1])
```

transition(*i*)

The multiplication on the right by a generator.

INPUT:

- `i` – an element from the indexing set of the generators

This method computes `self * self._generators[i]`.

EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(17)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: M.repr_element_method("reduced_word")
sage: M.construct()
sage: a = M.an_element()
sage: a.transition(1)
[1, 1]
sage: a.transition(2)
```

```
[1, 2]
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(17))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):
↪R(Integer(5))}), one=R.one())
>>> M.repr_element_method("reduced_word")
>>> M.construct()
>>> a = M.an_element()
>>> a.transition(Integer(1))
[1, 1]
>>> a.transition(Integer(2))
[1, 2]
```

ambient()

Return the ambient semigroup of `self`.

EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(12)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: M.ambient()
Ring of integers modulo 12

sage: # needs sage.modules
sage: M1 = matrix([[0,0,1],[1,0,0],[0,1,0]])
sage: M2 = matrix([[0,0,0],[1,1,0],[0,0,1]])
sage: M1.set_immutable()
sage: M2.set_immutable()
sage: def prod_m(x, y):
....:     z=x*y
....:     z.set_immutable()
....:     return z
sage: Mon = AutomaticSemigroup([M1,M2], mul=prod_m)
sage: Mon.ambient()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(12))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):
↪R(Integer(5))}), one=R.one())
>>> M.ambient()
Ring of integers modulo 12

>>> # needs sage.modules
>>> M1 = matrix([[Integer(0),Integer(0),Integer(1)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0)]])
>>> M2 = matrix([[Integer(0),Integer(0),Integer(0)],[Integer(1),Integer(1),
↪Integer(0)],[Integer(0),Integer(0),Integer(1)]])
```

```
>>> M1.set_immutable()
>>> M2.set_immutable()
>>> def prod_m(x, y):
...     z=x*y
...     z.set_immutable()
...     return z
>>> Mon = AutomaticSemigroup([M1,M2], mul=prod_m)
>>> Mon.ambient()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

an_element()

Return the first given generator of `self`.

EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(16)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: M.an_element()
3
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(16))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):
↪R(Integer(5))}), one=R.one())
>>> M.an_element()
3
```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(12)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: M.cardinality()
4
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(12))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):
↪R(Integer(5))}), one=R.one())
>>> M.cardinality()
4
```

construct(*up_to=None*, *n=None*)

Construct the elements of `self`.

INPUT:

- `up_to` – an element of `self` or of the ambient semigroup

> • n – integer or `None` (default: `None`)

This construct all the elements of this semigroup, their reduced words, and the right Cayley graph. If n is specified, only the n first elements of the semigroup are constructed. If `element` is specified, only the elements up to `ambient_element` are constructed.

EXAMPLES:

```
sage: # needs sage.groups sage.modules
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: W = WeylGroup(['A',3]); W.rename('W')
sage: ambient_monoid = FiniteSetMaps(W, action='right')
sage: pi = W.simple_projections(length_increasing=True).map(ambient_monoid)
sage: M = AutomaticSemigroup(pi, one=ambient_monoid.one()); M
A submonoid of (Maps from W to itself) with 3 generators
sage: M.repr_element_method("reduced_word")
sage: sorted(M._elements_set, key=str)
[[1], [2], [3], []]
sage: elt = M.from_reduced_word([2,3,1,2])
sage: M.construct(up_to=elt)
sage: len(M._elements_set)
19
sage: M.cardinality()
24
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.modules
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> W = WeylGroup(['A',Integer(3)]); W.rename('W')
>>> ambient_monoid = FiniteSetMaps(W, action='right')
>>> pi = W.simple_projections(length_increasing=True).map(ambient_monoid)
>>> M = AutomaticSemigroup(pi, one=ambient_monoid.one()); M
A submonoid of (Maps from W to itself) with 3 generators
>>> M.repr_element_method("reduced_word")
>>> sorted(M._elements_set, key=str)
[[1], [2], [3], []]
>>> elt = M.from_reduced_word([Integer(2),Integer(3),Integer(1),Integer(2)])
>>> M.construct(up_to=elt)
>>> len(M._elements_set)
19
>>> M.cardinality()
24
```

from_reduced_word(l)

> Return the element of `self` obtained from the reduced word `l`.
>
> INPUT:
>
> • `l` – list of indices of the generators

> ⓘ **Note**
>
> We do not save the given reduced word `l` as an attribute of the element, as some elements above in the branches may have not been explored by the iterator yet.

EXAMPLES:

```
sage: # needs sage.groups
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: G4 = SymmetricGroup(4)
sage: M = AutomaticSemigroup(Family({1:G4((1,2)), 2:G4((1,2,3,4))}),
....:                        one=G4.one())
sage: M.from_reduced_word([2, 1, 2, 2, 1]).lift()
(1,3)
sage: M.from_reduced_word([2, 1, 2, 2, 1]) == M.retract(G4((3,1)))
True
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> G4 = SymmetricGroup(Integer(4))
>>> M = AutomaticSemigroup(Family({Integer(1):G4((Integer(1),Integer(2))),
→Integer(2):G4((Integer(1),Integer(2),Integer(3),Integer(4)))}),
...                        one=G4.one())
>>> M.from_reduced_word([Integer(2), Integer(1), Integer(2), Integer(2),
→Integer(1)]).lift()
(1,3)
>>> M.from_reduced_word([Integer(2), Integer(1), Integer(2), Integer(2),
→Integer(1)]) == M.retract(G4((Integer(3),Integer(1))))
True
```

gens()

> Return the family of generators of `self`.
>
> EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(28)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}))
sage: M.semigroup_generators()
Finite family {1: 3, 2: 5}
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(28))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):
→R(Integer(5))}))
>>> M.semigroup_generators()
Finite family {1: 3, 2: 5}
```

lift(*x*)

> Lift an element of `self` into its ambient space.
>
> EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(15)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: a = M.an_element()
```

```
sage: a.lift() in R
True
sage: a.lift()
3
sage: [m.lift() for m in M]
[1, 3, 5, 9, 0, 10, 12, 6]
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(15))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}), one=R.one())
>>> a = M.an_element()
>>> a.lift() in R
True
>>> a.lift()
3
>>> [m.lift() for m in M]
[1, 3, 5, 9, 0, 10, 12, 6]
```

list()

> Return the list of elements of self.
>
> EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(12)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: M.repr_element_method("reduced_word")
sage: M.list()
[[], [1], [2], [1, 1]]
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(12))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}), one=R.one())
>>> M.repr_element_method("reduced_word")
>>> M.list()
[[], [1], [2], [1, 1]]
```

product(*x*, *y*)

> Return the product of two elements in self. It is done by retracting the multiplication in the ambient semigroup.
>
> EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(12)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: a = M[1]
sage: b = M[2]
```

```
sage: a*b
[1]
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(12))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}), one=R.one())
>>> a = M[Integer(1)]
>>> b = M[Integer(2)]
>>> a*b
[1]
```

repr_element_method(*style='ambient'*)

> Set the representation of the elements of the monoid.
>
> INPUT:
>
> • `style` – "ambient" or "reduced_word"
>
> EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(17)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}), one=R.one())
sage: M.list()
[1, 3, 5, 9, 15, 8, 10, 11, 7, 6, 13, 16, 4, 14, 12, 2]
sage: M.repr_element_method("reduced_word")
sage: M.list()
[[], [1], [2], [1, 1], [1, 2], [2, 2], [1, 1, 1], [1, 1, 2], [1, 2, 2],
 [2, 2, 2], [1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 2], [1, 1, 1, 1, 2],
 [1, 1, 1, 2, 2], [1, 1, 1, 1, 2, 2]]
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(17))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):␣
↪R(Integer(5))}), one=R.one())
>>> M.list()
[1, 3, 5, 9, 15, 8, 10, 11, 7, 6, 13, 16, 4, 14, 12, 2]
>>> M.repr_element_method("reduced_word")
>>> M.list()
[[], [1], [2], [1, 1], [1, 2], [2, 2], [1, 1, 1], [1, 1, 2], [1, 2, 2],
 [2, 2, 2], [1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 2], [1, 1, 1, 1, 2],
 [1, 1, 1, 2, 2], [1, 1, 1, 1, 2, 2]]
```

retract(*ambient_element*, *check=True*)

> Retract an element of the ambient semigroup into `self`.
>
> EXAMPLES:

```
sage: # needs sage.groups
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
```

```
sage: S5 = SymmetricGroup(5); S5.rename('S5')
sage: M = AutomaticSemigroup(Family({1:S5((1,2)), 2:S5((1,2,3,4))}),
....:                        one=S5.one())
sage: m = M.retract(S5((3,1))); m
(1,3)
sage: m.parent() is M
True
sage: M.retract(S5((4,5)), check=False)
(4,5)
sage: M.retract(S5((4,5)))
Traceback (most recent call last):
...
ValueError: (4,5) not in A subgroup of (S5) with 2 generators
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> S5 = SymmetricGroup(Integer(5)); S5.rename('S5')
>>> M = AutomaticSemigroup(Family({Integer(1):S5((Integer(1),Integer(2))),
→Integer(2):S5((Integer(1),Integer(2),Integer(3),Integer(4)))}),
...                        one=S5.one())
>>> m = M.retract(S5((Integer(3),Integer(1)))); m
(1,3)
>>> m.parent() is M
True
>>> M.retract(S5((Integer(4),Integer(5))), check=False)
(4,5)
>>> M.retract(S5((Integer(4),Integer(5))))
Traceback (most recent call last):
...
ValueError: (4,5) not in A subgroup of (S5) with 2 generators
```

semigroup_generators()

    Return the family of generators of `self`.

    EXAMPLES:

```
sage: from sage.monoids.automatic_semigroup import AutomaticSemigroup
sage: R = IntegerModRing(28)
sage: M = AutomaticSemigroup(Family({1: R(3), 2: R(5)}))
sage: M.semigroup_generators()
Finite family {1: 3, 2: 5}
```

```
>>> from sage.all import *
>>> from sage.monoids.automatic_semigroup import AutomaticSemigroup
>>> R = IntegerModRing(Integer(28))
>>> M = AutomaticSemigroup(Family({Integer(1): R(Integer(3)), Integer(2):
→R(Integer(5))}))
>>> M.semigroup_generators()
Finite family {1: 3, 2: 5}
```

TWELVE

MODULE OF TRACE MONOIDS (FREE PARTIALLY COMMUTATIVE MONOIDS).

EXAMPLES:

We first create a trace monoid:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: M.<a,b,c> = TraceMonoid(I=(('a','c'), ('c','a'))); M
Trace monoid on 3 generators ([a], [b], [c]) with independence relation {{a, c}}
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> M = TraceMonoid(I=(('a','c'), ('c','a')), names=('a', 'b', 'c',)); (a, b, c,) = M.
↪_first_ngens(3); M
Trace monoid on 3 generators ([a], [b], [c]) with independence relation {{a, c}}
```

Different elements can be equal because of the partially commutative multiplication:

```
sage: c * a * b == a * c * b
True
```

```
>>> from sage.all import *
>>> c * a * b == a * c * b
True
```

We check that it is a monoid:

```
sage: M in Monoids()
True
```

```
>>> from sage.all import *
>>> M in Monoids()
True
```

REFERENCES:

- Wikipedia article Trace_monoid
- https://ncatlab.org/nlab/show/trace+monoid

AUTHORS:

- Pavlo Tokariev (2019-05-31): initial version

class sage.monoids.trace_monoid.**TraceMonoid**(*M*, *I*, *names*)

Bases: `UniqueRepresentation`, *`Monoid_class`*

Return a free partially commuting monoid (trace monoid) on n generators over independence relation I.

We construct a trace monoid by specifying:

- a free monoid and independence relation
- or generator names and independence relation, FreeMonoid is constructed automatically then.

INPUT:

- `M` – a free monoid
- `I` – commutation relation between generators (or their names if the `names` are given)
- `names` – names of generators

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: F = TraceMonoid(names=('a', 'b', 'c'), I={('a','c'), ('c','a')}); F
Trace monoid on 3 generators ([a], [b], [c]) with independence relation {{a, c}}
sage: x = F.gens()
sage: x[0]*x[1]**5 * (x[0]*x[2])
[a*b^5*a*c]

sage: from sage.monoids.trace_monoid import TraceMonoid
sage: M.<a,b,c> = TraceMonoid(I=(('a','c'), ('c','a')))
sage: latex(M)
\langle a, b, c \mid ac=ca \rangle
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> F = TraceMonoid(names=('a', 'b', 'c'), I={('a','c'), ('c','a')}); F
Trace monoid on 3 generators ([a], [b], [c]) with independence relation {{a, c}}
>>> x = F.gens()
>>> x[Integer(0)]*x[Integer(1)]**Integer(5) * (x[Integer(0)]*x[Integer(2)])
[a*b^5*a*c]

>>> from sage.monoids.trace_monoid import TraceMonoid
>>> M = TraceMonoid(I=(('a','c'), ('c','a')), names=('a', 'b', 'c',)); (a, b, c,)
→= M._first_ngens(3)
>>> latex(M)
\langle a, b, c \mid ac=ca \rangle
```

Element

alias of *`TraceMonoidElement`*

cardinality()

Return the cardinality of `self`, which is infinite except for the trivial monoid.

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: M.<a,b,c> = TraceMonoid(I=(('a','c'), ('c','a')))
sage: M.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> M = TraceMonoid(I=(('a','c'), ('c','a')), names=('a', 'b', 'c',)); (a, b,
→c,) = M._first_ngens(3)
>>> M.cardinality()
+Infinity
```

dependence()

Return dependence relation over the monoid.

OUTPUT: set of non-commuting generator pairs

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: M.<a,b,c> = TraceMonoid(I=(('a','c'), ('c','a')))
sage: sorted(M.dependence())
[(a, a), (a, b), (b, a), (b, b), (b, c), (c, b), (c, c)]
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> M = TraceMonoid(I=(('a','c'), ('c','a')), names=('a', 'b', 'c',)); (a, b,
→c,) = M._first_ngens(3)
>>> sorted(M.dependence())
[(a, a), (a, b), (b, a), (b, b), (b, c), (c, b), (c, c)]
```

dependence_graph()

Return graph of dependence relation.

OUTPUT: dependence graph with generators as vertices

dependence_polynomial(*t=None*)

Return dependence polynomial.

The polynomial is defined as follows: $\sum i(-1)^i c_i t^i$, where c_i equals to number of full subgraphs of size i in the independence graph.

OUTPUT: a rational function in `t` with coefficients in the integer ring

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: M.dependence_polynomial()                                              #
→needs sage.graphs
1/(2*t^2 - 4*t + 1)
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
→first_ngens(4)
>>> M.dependence_polynomial()                                                #
→needs sage.graphs
1/(2*t^2 - 4*t + 1)
```

gen(*i=0*)

Return the *i*-th generator of the monoid.

INPUT:

- i – integer (default: 0)

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: M.<a,b,c> = TraceMonoid(I=(('a','c'), ('c','a')))
sage: M.gen(1)
[b]
sage: M.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> M = TraceMonoid(I=(('a','c'), ('c','a')), names=('a', 'b', 'c',)); (a, b,␣
↪c,) = M._first_ngens(3)
>>> M.gen(Integer(1))
[b]
>>> M.gen(Integer(4))
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

independence()

Return independence relation over the monoid.

OUTPUT: set of commuting generator pairs

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: F.<a,b,c> = FreeMonoid()
sage: I = frozenset(((a,c), (c,a)))
sage: M.<ac,bc,cc> = TraceMonoid(F, I=I)
sage: M.independence() == frozenset([frozenset([a,c])])
True
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> F = FreeMonoid(names=('a', 'b', 'c',)); (a, b, c,) = F._first_ngens(3)
>>> I = frozenset(((a,c), (c,a)))
>>> M = TraceMonoid(F, I=I, names=('ac', 'bc', 'cc',)); (ac, bc, cc,) = M._
↪first_ngens(3)
>>> M.independence() == frozenset([frozenset([a,c])])
True
```

independence_graph()

Return the digraph of independence relations.

OUTPUT: independence graph with generators as vertices

ngens()

    Return the number of generators of `self`.

    EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: M.<a,b,c> = TraceMonoid(I=(('a','c'), ('c','a')))
sage: M.ngens()
3
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> M = TraceMonoid(I=(('a','c'), ('c','a')), names=('a', 'b', 'c',)); (a, b,
→c,) = M._first_ngens(3)
>>> M.ngens()
3
```

number_of_words(*length*)

    Return number of unique words of defined length.

    INPUT:

        • `length` – integer; defines size of words what number should be computed

    OUTPUT: words number as integer

    EXAMPLES:

    Get number of words of size 3

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: M.number_of_words(3)                                              #
→needs sage.graphs
48
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
→first_ngens(4)
>>> M.number_of_words(Integer(3))                                       
→       # needs sage.graphs
48
```

one()

    Return the neutral element of `self`.

    EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: M.<a,b,c> = TraceMonoid(I=(('a','c'), ('c','a')))
sage: M.one()
1
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> M = TraceMonoid(I=(('a','c'), ('c','a')), names=('a', 'b', 'c',)); (a, b,␣
→c,) = M._first_ngens(3)
>>> M.one()
1
```

words(*length*)

Return all lexicographic forms of defined length.

INPUT:

- `length` – integer; defines size of words

OUTPUT: set of traces of size `length`

EXAMPLES:

All words of size 2:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: sorted(M.words(2))
[[a^2], [a*b], [a*c], [a*d], [b*a], [b^2], [b*c],
 [b*d], [c*a], [c^2], [c*d], [d*b], [d*c], [d^2]]
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
→first_ngens(4)
>>> sorted(M.words(Integer(2)))
[[a^2], [a*b], [a*c], [a*d], [b*a], [b^2], [b*c],
 [b*d], [c*a], [c^2], [c*d], [d*b], [d*c], [d^2]]
```

Get number of words of size 3:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: len(M.words(3))
48
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
→first_ngens(4)
>>> len(M.words(Integer(3)))
48
```

class sage.monoids.trace_monoid.**TraceMonoidElement**

Bases: `ElementWrapper`, `MonoidElement`

Element of a trace monoid, also known as a trace.

Elements of trace monoid is actually a equivalence classes of related free monoid over some equivalence relation that in the case is presented as independence relation.

Representative

We transform each trace to its lexicographic form for the representative in the ambient free monoid. This is also used for comparisons.

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: x = b * a * d * a * c * b
sage: x^3
[b*a^2*d*b^2*c*a^2*d*b^2*c*a^2*d*b*c]
sage: x^0
1
sage: x.lex_normal_form()
b*a^2*d*b*c
sage: x.foata_normal_form()
(b, a*d, a, b*c)
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._first_
→ngens(4)
>>> x = b * a * d * a * c * b
>>> x**Integer(3)
[b*a^2*d*b^2*c*a^2*d*b^2*c*a^2*d*b*c]
>>> x**Integer(0)
1
>>> x.lex_normal_form()
b*a^2*d*b*c
>>> x.foata_normal_form()
(b, a*d, a, b*c)
```

alphabet()

> Return alphabet of `self`.
>
> OUTPUT: a set of free monoid generators
>
> EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: x = b*a*d*a*c*b
sage: x.alphabet()
{b, a, d, c}
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
```

```
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
↪first_ngens(4)
>>> x = b*a*d*a*c*b
>>> x.alphabet()
{b, a, d, c}
```

dependence_graph()

> Return dependence graph of the trace.
>
> It is a directed graph where all dependent (non-commutative) generators are connected by edges which direction depend on the generator position in the trace.
>
> OUTPUT: directed graph of generator indexes
>
> EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: x = b * a * d * a * c * b
sage: x.dependence_graph()                                              #␣
↪needs sage.graphs
Digraph on 6 vertices
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
↪first_ngens(4)
>>> x = b * a * d * a * c * b
>>> x.dependence_graph()                                                #␣
↪needs sage.graphs
Digraph on 6 vertices
```

foata_normal_form()

> Return the Foata normal form of `self`.
>
> OUTPUT: tuple of free monoid elements
>
> EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: x = b * a * d * a * c * b
sage: x.foata_normal_form()
(b, a*d, a, b*c)
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
↪first_ngens(4)
>>> x = b * a * d * a * c * b
```

```
>>> x.foata_normal_form()
(b, a*d, a, b*c)
```

hasse_diagram(*algorithm='naive'*)

Return Hasse diagram of the trace.

Hasse diagram is a dependence graph without transitive edges.

INPUT:

- `algorithm` – string (default: `'naive'`); defines algorithm that will be used to compute Hasse diagram; there are two variants: `'naive'` and `'min'`.

OUTPUT: directed graph of generator indexes

> **See also**
>
> `naive_hasse_digram()`, *min_hasse_diagram()*.

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: x = b * a * d * a * c * b
sage: x.hasse_diagram()                                                    #␣
↪needs sage.graphs
Digraph on 6 vertices
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
↪first_ngens(4)
>>> x = b * a * d * a * c * b
>>> x.hasse_diagram()                                                      #␣
↪needs sage.graphs
Digraph on 6 vertices
```

lex_normal_form()

Return the lexicographic normal form of `self`.

OUTPUT: a free monoid element

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: (a*b).lex_normal_form()
a*b
sage: (b*a).lex_normal_form()
b*a
```

```
sage: (d*a).lex_normal_form()
a*d
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
→first_ngens(4)
>>> (a*b).lex_normal_form()
a*b
>>> (b*a).lex_normal_form()
b*a
>>> (d*a).lex_normal_form()
a*d
```

min_hasse_diagram()

Return Hasse diagram of the trace.

OUTPUT: directed graph of generator indexes

> **↪ See also**
>
> hasse_digram(), *naive_hasse_diagram()*.

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: x = b * a * d * a * c * b
sage: x.min_hasse_diagram()                                              #␣
→needs sage.graphs
Digraph on 6 vertices
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
→first_ngens(4)
>>> x = b * a * d * a * c * b
>>> x.min_hasse_diagram()                                                #␣
→needs sage.graphs
Digraph on 6 vertices
```

multiplicative_order()

Return the multiplicative order of self, which is ∞ for any element not the identity.

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
```

```
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: a.multiplicative_order()
+Infinity
sage: M.one().multiplicative_order()
1
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
↪first_ngens(4)
>>> a.multiplicative_order()
+Infinity
>>> M.one().multiplicative_order()
1
```

naive_hasse_diagram()

> Return Hasse diagram of `self`.
>
> ALGORITHM:
>
> In loop check for every two pair of edges if they have common vertex, remove their transitive edge.
>
> OUTPUT: directed graph of generator indexes
>
> > ⤷ **See also**
> >
> > `hasse_digram()`, *min_hasse_diagram()*.
>
> EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
sage: M.<a,b,c,d> = TraceMonoid(I=I)
sage: x = b * a * d * a * c * b
sage: x.naive_hasse_diagram()                                              #␣
↪needs sage.graphs
Digraph on 6 vertices
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> I = (('a','d'), ('d','a'), ('b','c'), ('c','b'))
>>> M = TraceMonoid(I=I, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = M._
↪first_ngens(4)
>>> x = b * a * d * a * c * b
>>> x.naive_hasse_diagram()                                                #␣
↪needs sage.graphs
Digraph on 6 vertices
```

projection(*letters*)

> Return a trace that formed from `self` by erasing `letters`.
>
> INPUT:

- `letters` – set of generators; defines set of letters that will be used to filter the trace

OUTPUT: a trace

EXAMPLES:

```
sage: from sage.monoids.trace_monoid import TraceMonoid
sage: F.<a,b,c,d> = FreeMonoid()
sage: I = ((a,d), (d,a), (b,c), (c,b))
sage: M.<ac,bc,cc,dc> = TraceMonoid(F, I=I)
sage: x = M(b*a*d*a*c*b)
sage: x.projection({a,b})
[b*a^2*b]
sage: x.projection({b,d,c})
[b*d*b*c]
```

```
>>> from sage.all import *
>>> from sage.monoids.trace_monoid import TraceMonoid
>>> F = FreeMonoid(names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = F._first_
→ngens(4)
>>> I = ((a,d), (d,a), (b,c), (c,b))
>>> M = TraceMonoid(F, I=I, names=('ac', 'bc', 'cc', 'dc',)); (ac, bc, cc, dc,
→) = M._first_ngens(4)
>>> x = M(b*a*d*a*c*b)
>>> x.projection({a,b})
[b*a^2*b]
>>> x.projection({b,d,c})
[b*d*b*c]
```

THIRTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

m

A

alphabet() (*sage.monoids.string_monoid.String-Monoid_class method*), 40

alphabet() (*sage.monoids.trace_monoid.Trace-MonoidElement method*), 77

AlphabeticStringMonoid (*class in sage.monoids.string_monoid*), 33

AlphabeticStrings (*in module sage.monoids.string_monoid*), 38

ambient() (*sage.monoids.automatic_semigroup.Automatic-Semigroup method*), 64

an_element() (*sage.monoids.automatic_semigroup.Au-tomaticSemigroup method*), 65

AutomaticMonoid (*class in sage.monoids.auto-matic_semigroup*), 53

AutomaticSemigroup (*class in sage.monoids.auto-matic_semigroup*), 55

AutomaticSemigroup.Element (*class in sage.monoids.automatic_semigroup*), 62

B

BinaryStringMonoid (*class in sage.monoids.string_monoid*), 38

BinaryStrings (*in module sage.monoids.string_monoid*), 39

C

cardinality() (*sage.monoids.automatic_semigroup.Au-tomaticSemigroup method*), 65

cardinality() (*sage.monoids.free_abelian_monoid.Free-AbelianMonoid_class method*), 15

cardinality() (*sage.monoids.free_monoid.FreeMonoid method*), 6

cardinality() (*sage.monoids.indexed_free_monoid.In-dexedMonoid method*), 28

cardinality() (*sage.monoids.trace_monoid.Trace-Monoid method*), 72

character_count() (*sage.monoids.string_monoid_ele-ment.StringMonoidElement method*), 43

characteristic_frequency() (*sage.monoids.string_monoid.AlphabeticString-Monoid method*), 33

coincidence_discriminant() (*in module sage.monoids.string_ops*), 49

coincidence_index() (*in module sage.monoids.string_ops*), 49

coincidence_index() (*sage.monoids.string_monoid_element.String-MonoidElement method*), 44

construct() (*sage.monoids.automatic_semigroup.Auto-maticSemigroup method*), 65

create_key() (*sage.monoids.free_abelian_monoid.Free-AbelianMonoidFactory method*), 15

create_object() (*sage.monoids.free_abelian_monoid.Free-AbelianMonoidFactory method*), 15

D

decoding() (*sage.monoids.string_monoid_ele-ment.StringMonoidElement method*), 44

dependence() (*sage.monoids.trace_monoid.Trace-Monoid method*), 73

dependence_graph() (*sage.monoids.trace_monoid.Trace-Monoid method*), 73

dependence_graph() (*sage.monoids.trace_monoid.Trace-MonoidElement method*), 78

dependence_polynomial() (*sage.monoids.trace_monoid.TraceMonoid method*), 73

dict() (*sage.monoids.indexed_free_monoid.IndexedFree-AbelianMonoidElement method*), 24

divides() (*sage.monoids.indexed_free_monoid.Indexed-FreeAbelianMonoidElement method*), 25

E

Element (*sage.monoids.free_abelian_monoid.Free-AbelianMonoid_class attribute*), 15

Element (*sage.monoids.free_monoid.FreeMonoid at-tribute*), 6

Element (*sage.monoids.indexed_free_monoid.Indexed-FreeAbelianMonoid attribute*), 24

Element (*sage.monoids.indexed_free_monoid.Indexed-FreeMonoid attribute*), 26

Element (*sage.monoids.trace_monoid.TraceMonoid attribute*), 72