Data Structures

Release 10.6

The Sage Development Team

Jun 27, 2025

CONTENTS

CHAPTER

ONE

BINARY TREES

This implements a binary tree in Cython.

AUTHORS:

- Tom Boothby (2007-02-15). Initial version free for any use (public domain).

class sage.misc.binary_tree.**BinaryTree**

Bases: object

A simple binary tree with integer keys.

contains(*key*)

Return whether a node with the given key exists in the tree.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.contains(1)
False
sage: t.insert(1, 1)
sage: t.contains(1)
True
```

```
>>> from sage.all import *
>>> from sage.misc.binary_tree import BinaryTree
>>> t = BinaryTree()
>>> t.contains(Integer(1))
False
>>> t.insert(Integer(1), Integer(1))
>>> t.contains(Integer(1))
True
```

delete(*key*)

Remove a the node corresponding to key, and return the value associated with it.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(3,3)
sage: t.insert(1,1)
sage: t.insert(2,2)
```

(continues on next page)

(continued from previous page)

```
sage: t.insert(0,0)
sage: t.insert(5,5)
sage: t.insert(6,6)
sage: t.insert(4,4)
sage: t.delete(0)
0
sage: t.delete(3)
3
sage: t.delete(5)
5
sage: t.delete(2)
2
sage: t.delete(6)
6
sage: t.delete(1)
1
sage: t.delete(0)
sage: t.get_max()
4
sage: t.get_min()
4
```

```
>>> from sage.all import *
>>> from sage.misc.binary_tree import BinaryTree
>>> t = BinaryTree()
>>> t.insert(Integer(3),Integer(3))
>>> t.insert(Integer(1),Integer(1))
>>> t.insert(Integer(2),Integer(2))
>>> t.insert(Integer(0),Integer(0))
>>> t.insert(Integer(5),Integer(5))
>>> t.insert(Integer(6),Integer(6))
>>> t.insert(Integer(4),Integer(4))
>>> t.delete(Integer(0))
0
>>> t.delete(Integer(3))
3
>>> t.delete(Integer(5))
5
>>> t.delete(Integer(2))
2
>>> t.delete(Integer(6))
6
>>> t.delete(Integer(1))
1
>>> t.delete(Integer(0))
>>> t.get_max()
4
>>> t.get_min()
4
```

get(*key*)

Return the value associated with the key given.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(0, Matrix([[0,0], [1,1]]))                                    #␣
↪needs sage.modules
sage: t.insert(0, 1)
sage: t.get(0)                                                               #␣
↪needs sage.modules
[0 0]
[1 1]
```

```
>>> from sage.all import *
>>> from sage.misc.binary_tree import BinaryTree
>>> t = BinaryTree()
>>> t.insert(Integer(0), Matrix([[Integer(0),Integer(0)], [Integer(1),
↪Integer(1)]]))                                    # needs sage.modules
>>> t.insert(Integer(0), Integer(1))
>>> t.get(Integer(0))                                                        ␣
↪     # needs sage.modules
[0 0]
[1 1]
```

get_max()

Return the value of the node with the maximal key value.

get_min()

Return the value of the node with the minimal key value.

insert(*key*, *value=None*)

Insert a key-value pair into the BinaryTree.

Duplicate keys are ignored.

The first parameter, key, should be an int, or coercible (one-to-one) into an int.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(1)
sage: t.insert(0)
sage: t.insert(2)
sage: t.insert(0,1)
sage: t.get(0)
0
```

```
>>> from sage.all import *
>>> from sage.misc.binary_tree import BinaryTree
>>> t = BinaryTree()
>>> t.insert(Integer(1))
>>> t.insert(Integer(0))
>>> t.insert(Integer(2))
>>> t.insert(Integer(0),Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> t.get(Integer(0))
0
```

is_empty()

Return whether the tree has no nodes.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.is_empty()
True
sage: t.insert(0,0)
sage: t.is_empty()
False
```

```
>>> from sage.all import *
>>> from sage.misc.binary_tree import BinaryTree
>>> t = BinaryTree()
>>> t.is_empty()
True
>>> t.insert(Integer(0),Integer(0))
>>> t.is_empty()
False
```

keys(*order='inorder'*)

Return the keys sorted according to "order" parameter.

The order can be one of "inorder", "preorder", or "postorder"

pop_max()

Return the value of the node with the maximal key value, and remove that node from the tree.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(4,'e')
sage: t.insert(2,'c')
sage: t.insert(0,'a')
sage: t.insert(1,'b')
sage: t.insert(3,'d')
sage: t.insert(5,'f')
sage: while not t.is_empty():
....:     print(t.pop_max())
f
e
d
c
b
a
```

```
>>> from sage.all import *
>>> from sage.misc.binary_tree import BinaryTree
>>> t = BinaryTree()
>>> t.insert(Integer(4),'e')
>>> t.insert(Integer(2),'c')
>>> t.insert(Integer(0),'a')
>>> t.insert(Integer(1),'b')
>>> t.insert(Integer(3),'d')
>>> t.insert(Integer(5),'f')
>>> while not t.is_empty():
...     print(t.pop_max())
f
e
d
c
b
a
```

pop_min()

Return the value of the node with the minimal key value, and remove that node from the tree.

EXAMPLES:

```
sage: from sage.misc.binary_tree import BinaryTree
sage: t = BinaryTree()
sage: t.insert(4,'e')
sage: t.insert(2,'c')
sage: t.insert(0,'a')
sage: t.insert(1,'b')
sage: t.insert(3,'d')
sage: t.insert(5,'f')
sage: while not t.is_empty():
....:     print(t.pop_min())
a
b
c
d
e
f
```

```
>>> from sage.all import *
>>> from sage.misc.binary_tree import BinaryTree
>>> t = BinaryTree()
>>> t.insert(Integer(4),'e')
>>> t.insert(Integer(2),'c')
>>> t.insert(Integer(0),'a')
>>> t.insert(Integer(1),'b')
>>> t.insert(Integer(3),'d')
>>> t.insert(Integer(5),'f')
>>> while not t.is_empty():
...     print(t.pop_min())
a
b
```

(continues on next page)

(continued from previous page)

```
c
d
e
f
```

values(*order='inorder'*)

Return the keys sorted according to "order" parameter.

The order can be one of "inorder", "preorder", or "postorder"

class sage.misc.binary_tree.**Test**

Bases: object

binary_tree(*values=100*, *cycles=100000*)

Perform a sequence of random operations, given random inputs to stress test the binary tree structure.

This was useful during development to find memory leaks / segfaults. Cycles should be at least 100 times as large as values, or the delete, contains, and get methods won't hit very often.

INPUT:

- values – number of possible values to use
- cycles – number of operations to perform

random()

CHAPTER

TWO

BITSETS

A Python interface to the fast bitsets in Sage. Bitsets are fast binary sets that store elements by toggling bits in an array of numbers. A bitset can store values between 0 and `capacity - 1`, inclusive (where `capacity` is finite, but arbitrary). The storage cost is linear in `capacity`.

Warning

This class is most likely to be useful as a way to store Cython bitsets in Python data structures, acting on them using the Cython inline functions. If you want to use these classes for a Python set type, the Python `set` or `frozenset` data types may be faster.

class sage.data_structures.bitset.**Bitset**

Bases: *FrozenBitset*

A bitset class which leverages inline Cython functions for creating and manipulating bitsets. See the class documentation of *FrozenBitset* for details on the parameters of the constructor and how to interpret the string representation of a *Bitset*.

A bitset can be thought of in two ways. First, as a set of elements from the universe of the n natural numbers $0, 1, \ldots, n-1$ (where the capacity n can be specified), with typical set operations such as intersection, union, symmetric difference, etc. Secondly, a bitset can be thought of as a binary vector with typical binary operations such as `and`, `or`, `xor`, etc. This class supports both interfaces.

The interface in this class mirrors the interface in the `set` data type of Python.

Warning

This class is most likely to be useful as a way to store Cython bitsets in Python data structures, acting on them using the Cython inline functions. If you want to use this class for a Python set type, the Python `set` data type may be faster.

See also

- *FrozenBitset*
- Python's set types

EXAMPLES:

```
sage: a = Bitset('1101')
sage: loads(dumps(a)) == a
True
sage: a = Bitset('1101' * 32)
sage: loads(dumps(a)) == a
True
```

```
>>> from sage.all import *
>>> a = Bitset('1101')
>>> loads(dumps(a)) == a
True
>>> a = Bitset('1101' * Integer(32))
>>> loads(dumps(a)) == a
True
```

add(*n*)

Update the bitset by adding `n`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.add(5)
sage: a
110001
sage: a.add(100)
sage: sorted(list(a))
[0, 1, 5, 100]
sage: a.capacity()
101
```

```
>>> from sage.all import *
>>> a = Bitset('110')
>>> a.add(Integer(5))
>>> a
110001
>>> a.add(Integer(100))
>>> sorted(list(a))
[0, 1, 5, 100]
>>> a.capacity()
101
```

clear()

Remove all elements from the bitset.

EXAMPLES:

```
sage: a = Bitset('011')
sage: a.clear()
sage: a
000
sage: a = Bitset('011' * 32)
sage: a.clear()
```

(continues on next page)

(continued from previous page)

```
sage: set(a)
set()
```

```
>>> from sage.all import *
>>> a = Bitset('011')
>>> a.clear()
>>> a
000
>>> a = Bitset('011' * Integer(32))
>>> a.clear()
>>> set(a)
set()
```

difference_update(*other*)

Update the bitset to the difference of `self` and `other`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.difference_update(Bitset('0101'))
sage: a
1000
sage: a_set = set(a)
sage: a.difference_update(FrozenBitset('010101' * 10)); a
1000000000000000000000000000000000000000000000000000000000000000000000
sage: a_set.difference_update(FrozenBitset('010101' * 10))
sage: a_set == set(a)
True
sage: a.difference_update(FrozenBitset('110'))
sage: a_set.difference_update(FrozenBitset('110'))
sage: a_set == set(a)
True
sage: a.difference_update(FrozenBitset('01010' * 20)); a
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪00000000000000000000
sage: a_set.difference_update(FrozenBitset('01010' * 20))
sage: a_set == set(a)
True
sage: b = Bitset('10101' * 20)
sage: b_set = set(b)
sage: b.difference_update(FrozenBitset('1' * 5)); b
0000010101101011010110101101011010110101101011010110101101011010110101101011010110101101
↪01101011010110101101
sage: b_set.difference_update(FrozenBitset('1' * 5))
sage: b_set == set(b)
True
```

```
>>> from sage.all import *
>>> a = Bitset('110')
>>> a.difference_update(Bitset('0101'))
>>> a
1000
```

(continues on next page)

(continued from previous page)

```
>>> a_set = set(a)
>>> a.difference_update(FrozenBitset('010101' * Integer(10))); a
1000000000000000000000000000000000000000000000000000000000000000000000
>>> a_set.difference_update(FrozenBitset('010101' * Integer(10)))
>>> a_set == set(a)
True
>>> a.difference_update(FrozenBitset('110'))
>>> a_set.difference_update(FrozenBitset('110'))
>>> a_set == set(a)
True
>>> a.difference_update(FrozenBitset('01010' * Integer(20))); a
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪0000000000000000000000
>>> a_set.difference_update(FrozenBitset('01010' * Integer(20)))
>>> a_set == set(a)
True
>>> b = Bitset('10101' * Integer(20))
>>> b_set = set(b)
>>> b.difference_update(FrozenBitset('1' * Integer(5))); b
0000010101101011010110101101011010110101101011010110101101011010110101101011010110101101011010110101101
↪0110101101011010110101
>>> b_set.difference_update(FrozenBitset('1' * Integer(5)))
>>> b_set == set(b)
True
```

discard(*n*)

Update the bitset by removing `n`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.discard(1)
sage: a
100
sage: a.discard(2)
sage: a.discard(4)
sage: a
100
sage: a = Bitset('000001' * 15); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83, 89]
sage: a.discard(83); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
sage: a.discard(82); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
```

```
>>> from sage.all import *
>>> a = Bitset('110')
>>> a.discard(Integer(1))
>>> a
100
>>> a.discard(Integer(2))
>>> a.discard(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> a
100
>>> a = Bitset('000001' * Integer(15)); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83, 89]
>>> a.discard(Integer(83)); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
>>> a.discard(Integer(82)); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
```

intersection_update(*other*)

Update the bitset to the intersection of `self` and `other`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.intersection_update(Bitset('0101'))
sage: a
0100
sage: a_set = set(a)
sage: a.intersection_update(Bitset('0110' * 25))
sage: a
0100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪0000000000000000000000
sage: a_set.intersection_update(Bitset('0110' * 25))
sage: set(a) == a_set
True
```

```
>>> from sage.all import *
>>> a = Bitset('110')
>>> a.intersection_update(Bitset('0101'))
>>> a
0100
>>> a_set = set(a)
>>> a.intersection_update(Bitset('0110' * Integer(25)))
>>> a
0100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪0000000000000000000000
>>> a_set.intersection_update(Bitset('0110' * Integer(25)))
>>> set(a) == a_set
True
```

pop()

Remove and return an arbitrary element from the set.

This raises a `KeyError` if the set is empty.

EXAMPLES:

```
sage: a = Bitset('011')
sage: a.pop()
1
sage: a
001
```

(continues on next page)

(continued from previous page)

```
sage: a.pop()
2
sage: a
000
sage: a.pop()
Traceback (most recent call last):
...
KeyError: 'pop from an empty set'
sage: a = Bitset('0001'*32)
sage: a.pop()
3
sage: [a.pop() for _ in range(20)]
[7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63, 67, 71, 75, 79,␣
↪83]
```

```
>>> from sage.all import *
>>> a = Bitset('011')
>>> a.pop()
1
>>> a
001
>>> a.pop()
2
>>> a
000
>>> a.pop()
Traceback (most recent call last):
...
KeyError: 'pop from an empty set'
>>> a = Bitset('0001'*Integer(32))
>>> a.pop()
3
>>> [a.pop() for _ in range(Integer(20))]
[7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63, 67, 71, 75, 79,␣
↪83]
```

remove(*n*)

Update the bitset by removing `n`.

This raises a `KeyError` if `n` is not contained in the bitset.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.remove(1)
sage: a
100
sage: a.remove(2)
Traceback (most recent call last):
...
KeyError: 2
sage: a.remove(4)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
KeyError: 4
sage: a
100
sage: a = Bitset('000001' * 15); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83, 89]
sage: a.remove(83); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
```

```
>>> from sage.all import *
>>> a = Bitset('110')
>>> a.remove(Integer(1))
>>> a
100
>>> a.remove(Integer(2))
Traceback (most recent call last):
...
KeyError: 2
>>> a.remove(Integer(4))
Traceback (most recent call last):
...
KeyError: 4
>>> a
100
>>> a = Bitset('000001' * Integer(15)); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 83, 89]
>>> a.remove(Integer(83)); sorted(list(a))
[5, 11, 17, 23, 29, 35, 41, 47, 53, 59, 65, 71, 77, 89]
```

symmetric_difference_update(*other*)

Update the bitset to the symmetric difference of `self` and `other`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.symmetric_difference_update(Bitset('0101'))
sage: a
1001
sage: a_set = set(a)
sage: a.symmetric_difference_update(FrozenBitset('010101' * 10)); a
1100010101010101010101010101010101010101010101010101010101010101
sage: a_set.symmetric_difference_update(FrozenBitset('010101' * 10))
sage: a_set == set(a)
True
sage: a.symmetric_difference_update(FrozenBitset('01010' * 20)); a
1001011111000001111100000111110000011111000001111100000111110101001010010100
↪10010100101001010010
sage: a_set.symmetric_difference_update(FrozenBitset('01010' * 20))
sage: a_set == set(a)
True
sage: b = Bitset('10101' * 20)
sage: b_set = set(b)
```

(continues on next page)

(continued from previous page)

```
sage: b.symmetric_difference_update( FrozenBitset('1' * 5)); b
0101010101101011010110101101011010110101101011010110101101011010110101101011010110101101
↪011010110101101010110101
sage: b_set.symmetric_difference_update( FrozenBitset('1' * 5))
sage: b_set == set(b)
True
```

```
>>> from sage.all import *
>>> a = Bitset('110')
>>> a.symmetric_difference_update(Bitset('0101'))
>>> a
1001
>>> a_set = set(a)
>>> a.symmetric_difference_update(FrozenBitset('010101' * Integer(10))); a
1100010101010101010101010101010101010101010101010101010101010101
>>> a_set.symmetric_difference_update(FrozenBitset('010101' * Integer(10)))
>>> a_set == set(a)
True
>>> a.symmetric_difference_update(FrozenBitset('01010' * Integer(20))); a
1001011111000001111100000111110000011111000001111100000111110101001010010100101
↪100101001010010100101010
>>> a_set.symmetric_difference_update(FrozenBitset('01010' * Integer(20)))
>>> a_set == set(a)
True
>>> b = Bitset('10101' * Integer(20))
>>> b_set = set(b)
>>> b.symmetric_difference_update( FrozenBitset('1' * Integer(5))); b
0101010101101011010110101101011010110101101011010110101101011010110101101011010110101101
↪011010110101101010110101
>>> b_set.symmetric_difference_update( FrozenBitset('1' * Integer(5)))
>>> b_set == set(b)
True
```

update(*other*)

Update the bitset to include items in `other`.

EXAMPLES:

```
sage: a = Bitset('110')
sage: a.update(Bitset('0101'))
sage: a
1101
sage: a_set = set(a)
sage: a.update(Bitset('00011' * 25))
sage: a
1101100011000110001100011000110001100011000110001100011000110001100011000110001100011000
↪110001100011000110001100011000110001100011000110001100011
sage: a_set.update(Bitset('00011' * 25))
sage: set(a) == a_set
True
```

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

(continues on next page)

(continued from previous page)

```
>>> a = Bitset('110')
>>> a.update(Bitset('0101'))
>>> a
1101
>>> a_set = set(a)
>>> a.update(Bitset('00011' * Integer(25)))
>>> a
1101100011000110001100011000110001100011000110001100011000110001100011000110001100011000
↪1100011000110001100011000110001100011000110001100011
>>> a_set.update(Bitset('00011' * Integer(25)))
>>> set(a) == a_set
True
```

class sage.data_structures.bitset.**FrozenBitset**

Bases: object

A frozen bitset class which leverages inline Cython functions for creating and manipulating bitsets.

A bitset can be thought of in two ways. First, as a set of elements from the universe of the n natural numbers $0, 1, \ldots, n-1$ (where the capacity n can be specified), with typical set operations such as intersection, union, symmetric difference, etc. Secondly, a bitset can be thought of as a binary vector with typical binary operations such as `and`, `or`, `xor`, etc. This class supports both interfaces.

The interface in this class mirrors the interface in the `frozenset` data type of Python. See the Python documentation on set types for more details on Python's `set` and `frozenset` classes.

Warning

This class is most likely to be useful as a way to store Cython bitsets in Python data structures, acting on them using the Cython inline functions. If you want to use this class for a Python set type, the Python `frozenset` data type may be faster.

INPUT:

- `iter` – initialization parameter (default: `None`); valid inputs are:
  - *Bitset* and *FrozenBitset* – If this is a *Bitset* or *FrozenBitset*, then it is copied
  - `None` – if `None`, then the bitset is set to the empty set
  - `string` – if a nonempty string, then the bitset is initialized by including an element if the index of the string is `1`. If the string is empty, then raise a `ValueError`.
  - `iterable` – if an iterable, then it is assumed to contain a list of nonnegative integers and those integers are placed in the set
- `capacity` – (default: `None`) the maximum capacity of the bitset. If this is not specified, then it is automatically calculated from the passed iterable. It must be at least one.

OUTPUT: none

The string representation of a *FrozenBitset* `FB` can be understood as follows. Let $B = b_0 b_1 b_2 \cdots b_k$ be the string representation of the bitset `FB`, where each $b_i \in \{0, 1\}$. We read the b_i from left to right. If $b_i = 1$, then the nonnegative integer i is in the bitset `FB`. Similarly, if $b_i = 0$, then i is not in `FB`. In other words, `FB` is a subset of $\{0, 1, 2, \ldots, k\}$ and the membership in `FB` of each i is determined by the binary value b_i.

See also

- *Bitset*
- Python's set types

EXAMPLES:

The default bitset, which has capacity 1:

```
sage: FrozenBitset()
0
sage: FrozenBitset(None)
0
```

```
>>> from sage.all import *
>>> FrozenBitset()
0
>>> FrozenBitset(None)
0
```

Trying to create an empty bitset fails:

```
sage: FrozenBitset([])
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
sage: FrozenBitset(list())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
sage: FrozenBitset(())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
sage: FrozenBitset(tuple())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
sage: FrozenBitset("")
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
```

```
>>> from sage.all import *
>>> FrozenBitset([])
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
>>> FrozenBitset(list())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
```

(continues on next page)

(continued from previous page)

```
>>> FrozenBitset(())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
>>> FrozenBitset(tuple())
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
>>> FrozenBitset("")
Traceback (most recent call last):
...
ValueError: Bitsets must not be empty
```

We can create the all-zero bitset as follows:

```
sage: FrozenBitset(capacity=10)
0000000000
sage: FrozenBitset([], capacity=10)
0000000000
```

```
>>> from sage.all import *
>>> FrozenBitset(capacity=Integer(10))
0000000000
>>> FrozenBitset([], capacity=Integer(10))
0000000000
```

We can initialize a *FrozenBitset* with a *Bitset* or another *FrozenBitset*, and compare them for equality. As they are logically the same bitset, the equality test should return `True`. Furthermore, each bitset is a subset of the other.

```
sage: def bitcmp(a, b, c):  # custom function for comparing bitsets
....:     print(a == b == c)
....:     print((a <= b, b <= c, a <= c))
....:     print((a >= b, b >= c, a >= c))
....:     print((a != b, b != c, a != c))
sage: a = Bitset("1010110"); b = FrozenBitset(a); c = FrozenBitset(b)
sage: a; b; c
1010110
1010110
1010110
sage: a < b, b < c, a < c
(False, False, False)
sage: a > b, b > c, a > c
(False, False, False)
sage: bitcmp(a, b, c)
True
(True, True, True)
(True, True, True)
(False, False, False)
```

```
>>> from sage.all import *
>>> def bitcmp(a, b, c):  # custom function for comparing bitsets
```

(continues on next page)

(continued from previous page)

```
...       print(a == b == c)
...       print((a <= b, b <= c, a <= c))
...       print((a >= b, b >= c, a >= c))
...       print((a != b, b != c, a != c))
>>> a = Bitset("1010110"); b = FrozenBitset(a); c = FrozenBitset(b)
>>> a; b; c
1010110
1010110
1010110
>>> a < b, b < c, a < c
(False, False, False)
>>> a > b, b > c, a > c
(False, False, False)
>>> bitcmp(a, b, c)
True
(True, True, True)
(True, True, True)
(False, False, False)
```

Try a random bitset:

```
sage: a = Bitset(randint(0, 1) for n in range(1, randint(1, 10^4)))
sage: b = FrozenBitset(a); c = FrozenBitset(b)
sage: bitcmp(a, b, c)
True
(True, True, True)
(True, True, True)
(False, False, False)
```

```
>>> from sage.all import *
>>> a = Bitset(randint(Integer(0), Integer(1)) for n in range(Integer(1),␣
↪randint(Integer(1), Integer(10)**Integer(4))))
>>> b = FrozenBitset(a); c = FrozenBitset(b)
>>> bitcmp(a, b, c)
True
(True, True, True)
(True, True, True)
(False, False, False)
```

A bitset with a hard-coded bitstring:

```
sage: FrozenBitset('101')
101
```

```
>>> from sage.all import *
>>> FrozenBitset('101')
101
```

For a string, only those positions with `1` would be initialized to `1` in the corresponding position in the bitset. All other characters in the string, including `0`, are set to `0` in the resulting bitset.

```
sage: FrozenBitset('a')
0
sage: FrozenBitset('abc')
000
sage: FrozenBitset('abc1')
0001
sage: FrozenBitset('0abc1')
00001
sage: FrozenBitset('0abc10')
000010
sage: FrozenBitset('0a*c10')
000010
```

```
>>> from sage.all import *
>>> FrozenBitset('a')
0
>>> FrozenBitset('abc')
000
>>> FrozenBitset('abc1')
0001
>>> FrozenBitset('0abc1')
00001
>>> FrozenBitset('0abc10')
000010
>>> FrozenBitset('0a*c10')
000010
```

Represent the first 10 primes as a bitset. The primes are stored as a list and as a tuple. We then recover the primes from its bitset representation, and query the bitset for its length (how many elements it contains) and whether an element is in the bitset. Note that the length of a bitset is different from its capacity. The length counts the number of elements currently in the bitset, while the capacity is the number of elements that the bitset can hold.

```
sage: p = primes_first_n(10); p                                                 #␣
↪needs sage.libs.pari
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: tuple(p)                                                                  #␣
↪needs sage.libs.pari
(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
sage: F = FrozenBitset(p); F; FrozenBitset(tuple(p))                            #␣
↪needs sage.libs.pari
00110101000101000101000100000 1
00110101000101000101000100000 1
```

```
>>> from sage.all import *
>>> p = primes_first_n(Integer(10)); p                                           ␣
↪     # needs sage.libs.pari
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
>>> tuple(p)                                                                   #␣
↪needs sage.libs.pari
(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
>>> F = FrozenBitset(p); F; FrozenBitset(tuple(p))                             #␣
↪needs sage.libs.pari
```

(continues on next page)

(continued from previous page)

```
00110101000101000101000100000 1
00110101000101000101000100000 1
```

Recover the primes from the bitset:

```
sage: for b in F:                                                           #␣
↪needs sage.libs.pari
....:     print(b)
2
3
...
29
sage: list(F)                                                               #␣
↪needs sage.libs.pari
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

```
>>> from sage.all import *
>>> for b in F:                                                             #␣
↪needs sage.libs.pari
...     print(b)
2
3
...
29
>>> list(F)                                                                 #␣
↪needs sage.libs.pari
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Query the bitset:

```
sage: # needs sage.libs.pari
sage: len(F)
10
sage: len(list(F))
10
sage: F.capacity()
30
sage: s = str(F); len(s)
30
sage: 2 in F
True
sage: 1 in F
False
```

```
>>> from sage.all import *
>>> # needs sage.libs.pari
>>> len(F)
10
>>> len(list(F))
10
>>> F.capacity()
30
```

(continues on next page)

(continued from previous page)

```
>>> s = str(F); len(s)
30
>>> Integer(2) in F
True
>>> Integer(1) in F
False
```

A random iterable, with all duplicate elements removed:

```
sage: L = [randint(0, 100) for n in range(1, randint(1, 10^4))]
sage: FrozenBitset(L) == FrozenBitset(list(set(L)))
True
sage: FrozenBitset(tuple(L)) == FrozenBitset(tuple(set(L)))
True
```

```
>>> from sage.all import *
>>> L = [randint(Integer(0), Integer(100)) for n in range(Integer(1),␣
↪randint(Integer(1), Integer(10)**Integer(4)))]
>>> FrozenBitset(L) == FrozenBitset(list(set(L)))
True
>>> FrozenBitset(tuple(L)) == FrozenBitset(tuple(set(L)))
True
```

capacity()

Return the size of the underlying bitset.

The maximum value that can be stored in the current underlying bitset is `self.capacity() - 1`.

EXAMPLES:

```
sage: FrozenBitset('11000').capacity()
5
sage: FrozenBitset('110' * 32).capacity()
96
sage: FrozenBitset(range(20), capacity=450).capacity()
450
```

```
>>> from sage.all import *
>>> FrozenBitset('11000').capacity()
5
>>> FrozenBitset('110' * Integer(32)).capacity()
96
>>> FrozenBitset(range(Integer(20)), capacity=Integer(450)).capacity()
450
```

complement()

Return the complement of `self`.

EXAMPLES:

```
sage: ~FrozenBitset('10101')
01010
sage: ~FrozenBitset('11111'*10)
```

(continues on next page)

(continued from previous page)

```
00000000000000000000000000000000000000000000000000
sage: x = FrozenBitset('10'*40)
sage: x == ~x
False
sage: x == ~~x
True
sage: x|(~x) == FrozenBitset('11'*40)
True
sage: ~x == FrozenBitset('01'*40)
True
```

```
>>> from sage.all import *
>>> ~FrozenBitset('10101')
01010
>>> ~FrozenBitset('11111'*Integer(10))
00000000000000000000000000000000000000000000000000
>>> x = FrozenBitset('10'*Integer(40))
>>> x == ~x
False
>>> x == ~~x
True
>>> x|(~x) == FrozenBitset('11'*Integer(40))
True
>>> ~x == FrozenBitset('01'*Integer(40))
True
```

difference(*other*)

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

EXAMPLES:

```
sage: FrozenBitset('10101').difference(FrozenBitset('11100'))
00001
sage: FrozenBitset('11111' * 10).difference(FrozenBitset('010101' * 10))
10101010101010101010101010101010101010101010101010100000000000
```

```
>>> from sage.all import *
>>> FrozenBitset('10101').difference(FrozenBitset('11100'))
00001
>>> FrozenBitset('11111' * Integer(10)).difference(FrozenBitset('010101' *␣
↪Integer(10)))
10101010101010101010101010101010101010101010101010100000000000
```

intersection(*other*)

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

EXAMPLES:

```
sage: FrozenBitset('10101').intersection(FrozenBitset('11100'))
10100
sage: FrozenBitset('11111' * 10).intersection(FrozenBitset('010101' * 10))
01010101010101010101010101010101010101010101010101000000000000
```

```
>>> from sage.all import *
>>> FrozenBitset('10101').intersection(FrozenBitset('11100'))
10100
>>> FrozenBitset('11111' * Integer(10)).intersection(FrozenBitset('010101' *␣
↪Integer(10)))
01010101010101010101010101010101010101010101010101010000000000
```

isdisjoint(*other*)

Test to see if `self` is disjoint from `other`.

EXAMPLES:

```
sage: FrozenBitset('11').isdisjoint(FrozenBitset('01'))
False
sage: FrozenBitset('01').isdisjoint(FrozenBitset('001'))
True
sage: FrozenBitset('00101').isdisjoint(FrozenBitset('110' * 35))
False
```

```
>>> from sage.all import *
>>> FrozenBitset('11').isdisjoint(FrozenBitset('01'))
False
>>> FrozenBitset('01').isdisjoint(FrozenBitset('001'))
True
>>> FrozenBitset('00101').isdisjoint(FrozenBitset('110' * Integer(35)))
False
```

isempty()

Test if the bitset is empty.

OUTPUT: boolean

EXAMPLES:

```
sage: FrozenBitset().isempty()
True
sage: FrozenBitset([1]).isempty()
False
sage: FrozenBitset([], capacity=110).isempty()
True
sage: FrozenBitset(range(99)).isempty()
False
```

```
>>> from sage.all import *
>>> FrozenBitset().isempty()
True
>>> FrozenBitset([Integer(1)]).isempty()
False
>>> FrozenBitset([], capacity=Integer(110)).isempty()
True
>>> FrozenBitset(range(Integer(99))).isempty()
False
```

issubset(*other*)

Test to see if `self` is a subset of `other`.

EXAMPLES:

```
sage: FrozenBitset('11').issubset(FrozenBitset('01'))
False
sage: FrozenBitset('01').issubset(FrozenBitset('11'))
True
sage: FrozenBitset('01').issubset(FrozenBitset('01' * 45))
True
```

```
>>> from sage.all import *
>>> FrozenBitset('11').issubset(FrozenBitset('01'))
False
>>> FrozenBitset('01').issubset(FrozenBitset('11'))
True
>>> FrozenBitset('01').issubset(FrozenBitset('01' * Integer(45)))
True
```

issuperset(*other*)

Test to see if `self` is a superset of `other`.

EXAMPLES:

```
sage: FrozenBitset('11').issuperset(FrozenBitset('01'))
True
sage: FrozenBitset('01').issuperset(FrozenBitset('11'))
False
sage: FrozenBitset('01').issuperset(FrozenBitset('10' * 45))
False
```

```
>>> from sage.all import *
>>> FrozenBitset('11').issuperset(FrozenBitset('01'))
True
>>> FrozenBitset('01').issuperset(FrozenBitset('11'))
False
>>> FrozenBitset('01').issuperset(FrozenBitset('10' * Integer(45)))
False
```

symmetric_difference(*other*)

Return the symmetric difference of `self` and `other`.

EXAMPLES:

```
sage: FrozenBitset('10101').symmetric_difference(FrozenBitset('11100'))
01001
sage: FrozenBitset('11111' * 10).symmetric_difference(FrozenBitset('010101' *␣
↪10))
1010101010101010101010101010101010101010101010101010100101010101
```

```
>>> from sage.all import *
>>> FrozenBitset('10101').symmetric_difference(FrozenBitset('11100'))
01001
```

(continues on next page)

(continued from previous page)

```
>>> FrozenBitset('11111' * Integer(10)).symmetric_difference(FrozenBitset(
↪'010101' * Integer(10)))
10101010101010101010101010101010101010101010101010100101010101
```

union(*other*)

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

EXAMPLES:

```
sage: FrozenBitset('10101').union(FrozenBitset('11100'))
11101
sage: FrozenBitset('10101' * 10).union(FrozenBitset('01010' * 10))
11111111111111111111111111111111111111111111111111
```

```
>>> from sage.all import *
>>> FrozenBitset('10101').union(FrozenBitset('11100'))
11101
>>> FrozenBitset('10101' * Integer(10)).union(FrozenBitset('01010' *␣
↪Integer(10)))
11111111111111111111111111111111111111111111111111
```

sage.data_structures.bitset.**test_bitset**(*py_a*, *py_b*, *n*)

Test the Cython bitset functions so we can have some relevant doctests.

sage.data_structures.bitset.**test_bitset_copy_flex**(*py_a*)

sage.data_structures.bitset.**test_bitset_pop**(*py_a*)

Test for the bitset_pop function.

sage.data_structures.bitset.**test_bitset_remove**(*py_a*, *n*)

Test the bitset_remove function.

sage.data_structures.bitset.**test_bitset_set_first_n**(*py_a*, *n*)

Test the bitset function set_first_n.

sage.data_structures.bitset.**test_bitset_unpickle**(*data*)

This (artificially) tests pickling of bitsets across systems.

INPUT:

- `data` – tuple of data as would be produced by the internal, Cython-only, method `bitset_pickle`

OUTPUT: list form of the bitset corresponding to the pickled data

EXAMPLES:

We compare 64-bit and 32-bit encoding. Both should unpickle on any system:

```
sage: from sage.data_structures.bitset import test_bitset_unpickle
sage: test_bitset_unpickle((0, 100, 2, 8, (33, 6001)))
[0, 5, 64, 68, 69, 70, 72, 73, 74, 76]
sage: test_bitset_unpickle((0, 100, 4, 4, (33, 0, 6001, 0)))
[0, 5, 64, 68, 69, 70, 72, 73, 74, 76]
```

```
>>> from sage.all import *
>>> from sage.data_structures.bitset import test_bitset_unpickle
>>> test_bitset_unpickle((Integer(0), Integer(100), Integer(2), Integer(8),␣
↪(Integer(33), Integer(6001))))
[0, 5, 64, 68, 69, 70, 72, 73, 74, 76]
>>> test_bitset_unpickle((Integer(0), Integer(100), Integer(4), Integer(4),␣
↪(Integer(33), Integer(0), Integer(6001), Integer(0))))
[0, 5, 64, 68, 69, 70, 72, 73, 74, 76]
```

CHAPTER

THREE

SEQUENCES OF BOUNDED INTEGERS

This module provides *BoundedIntegerSequence*, which implements sequences of bounded integers and is for many (but not all) operations faster than representing the same sequence as a Python `tuple`.

The underlying data structure is similar to `Bitset`, which means that certain operations are implemented by using fast shift operations from MPIR. The following boilerplate functions can be cimported in Cython modules:

- `cdef bint biseq_init(biseq_t R, mp_size_t l, mp_size_t itemsize) except -1`

  Allocate memory for a bounded integer sequence of length `l` with items fitting in `itemsize` bits.

- `cdef inline void biseq_dealloc(biseq_t S)`

  Deallocate the memory used by `S`.

- `cdef bint biseq_init_copy(biseq_t R, biseq_t S)`

  Initialize `R` as a copy of `S`.

- `cdef tuple biseq_pickle(biseq_t S)`

  Return a triple `(bitset_data, itembitsize, length)` defining `S`.

- `cdef bint biseq_unpickle(biseq_t R, tuple bitset_data, mp_bitcnt_t itembitsize, mp_size_t length) except -1`

  Initialise `R` from data returned by `biseq_pickle`.

- `cdef bint biseq_init_list(biseq_t R, list data, size_t bound) except -1`

  Convert a list to a bounded integer sequence, which must not be allocated.

- `cdef inline Py_hash_t biseq_hash(biseq_t S)`

  Hash value for `S`.

- `cdef inline bint biseq_richcmp(biseq_t S1, biseq_t S2, int op)`

  Comparison of `S1` and `S2`. This takes into account the bound, the length, and the list of items of the two sequences.

- `cdef bint biseq_init_concat(biseq_t R, biseq_t S1, biseq_t S2) except -1`

  Concatenate `S1` and `S2` and write the result to `R`. Does not test whether the sequences have the same bound!

- `cdef inline bint biseq_startswith(biseq_t S1, biseq_t S2)`

  Is `S1=S2+something`? Does not check whether the sequences have the same bound!

- `cdef mp_size_t biseq_contains(biseq_t S1, biseq_t S2, mp_size_t start) except -2`

  Return the position in `S1` of `S2` as a subsequence of `S1[start:]`, or `-1` if `S2` is not a subsequence. Does not check whether the sequences have the same bound!

- `cdef mp_size_t biseq_starswith_tail(biseq_t S1, biseq_t S2, mp_size_t start) except -2:`

  Return the smallest number `i` such that the bounded integer sequence `S1` starts with the sequence `S2[i:]`, where `start <= i < S1.length`, or return `-1` if no such `i` exists.

- `cdef mp_size_t biseq_index(biseq_t S, size_t item, mp_size_t start) except -2`

  Return the position in `S` of the item in `S[start:]`, or `-1` if `S[start:]` does not contain the item.

- `cdef size_t biseq_getitem(biseq_t S, mp_size_t index)`

  Return `S[index]`, without checking margins.

- `cdef size_t biseq_getitem_py(biseq_t S, mp_size_t index)`

  Return `S[index]` as Python `int`, without checking margins.

- `cdef biseq_inititem(biseq_t S, mp_size_t index, size_t item)`

  Set `S[index] = item`, without checking margins and assuming that `S[index]` has previously been zero.

- `cdef inline void biseq_clearitem(biseq_t S, mp_size_t index)`

  Set `S[index] = 0`, without checking margins.

- `cdef bint biseq_init_slice(biseq_t R, biseq_t S, mp_size_t start, mp_size_t stop, mp_size_t step) except -1`

  Initialise `R` with `S[start:stop:step]`.

AUTHORS:

- Simon King, Jeroen Demeyer (2014-10): initial version (Issue #15820)

class sage.data_structures.bounded_integer_sequences.**BoundedIntegerSequence**

Bases: `object`

A sequence of nonnegative uniformly bounded integers.

INPUT:

- `bound` – nonnegative integer. When zero, a `ValueError` will be raised. Otherwise, the given bound is replaced by the power of two that is at least the given bound.
- `data` – list of integers

EXAMPLES:

We showcase the similarities and differences between bounded integer sequences and lists respectively tuples.

To distinguish from tuples or lists, we use pointed brackets for the string representation of bounded integer sequences:

```
sage: from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
sage: S = BoundedIntegerSequence(21, [2, 7, 20]); S
<2, 7, 20>
```

```
>>> from sage.all import *
>>> from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
>>> S = BoundedIntegerSequence(Integer(21), [Integer(2), Integer(7),␣
↪Integer(20)]); S
<2, 7, 20>
```

Each bounded integer sequence has a bound that is a power of two, such that all its item are less than this bound:

```
sage: S.bound()
32
sage: BoundedIntegerSequence(16, [2, 7, 20])
Traceback (most recent call last):
...
OverflowError: list item 20 larger than 15
```

```
>>> from sage.all import *
>>> S.bound()
32
>>> BoundedIntegerSequence(Integer(16), [Integer(2), Integer(7), Integer(20)])
Traceback (most recent call last):
...
OverflowError: list item 20 larger than 15
```

Bounded integer sequences are iterable, and we see that we can recover the originally given list:

```
sage: L = [randint(0,31) for i in range(5000)]
sage: S = BoundedIntegerSequence(32, L)
sage: list(L) == L
True
```

```
>>> from sage.all import *
>>> L = [randint(Integer(0),Integer(31)) for i in range(Integer(5000))]
>>> S = BoundedIntegerSequence(Integer(32), L)
>>> list(L) == L
True
```

Getting items and slicing works in the same way as for lists:

```
sage: n = randint(0,4999)
sage: S[n] == L[n]
True
sage: m = randint(0,1000)
sage: n = randint(3000,4500)
sage: s = randint(1, 7)
sage: list(S[m:n:s]) == L[m:n:s]
True
sage: list(S[n:m:-s]) == L[n:m:-s]
True
```

```
>>> from sage.all import *
>>> n = randint(Integer(0),Integer(4999))
>>> S[n] == L[n]
True
>>> m = randint(Integer(0),Integer(1000))
>>> n = randint(Integer(3000),Integer(4500))
>>> s = randint(Integer(1), Integer(7))
>>> list(S[m:n:s]) == L[m:n:s]
True
>>> list(S[n:m:-s]) == L[n:m:-s]
```

(continues on next page)

(continued from previous page)

```
True
```

The *index()* method works different for bounded integer sequences and tuples or lists. If one asks for the index of an item, the behaviour is the same. But we can also ask for the index of a sub-sequence:

```
sage: L.index(L[200]) == S.index(L[200])
True
sage: S.index(S[100:2000])     # random
100
```

```
>>> from sage.all import *
>>> L.index(L[Integer(200)]) == S.index(L[Integer(200)])
True
>>> S.index(S[Integer(100):Integer(2000)])     # random
100
```

Similarly, containment tests work for both items and sub-sequences:

```
sage: S[200] in S
True
sage: S[200:400] in S
True
sage: S[200]+S.bound() in S
False
```

```
>>> from sage.all import *
>>> S[Integer(200)] in S
True
>>> S[Integer(200):Integer(400)] in S
True
>>> S[Integer(200)]+S.bound() in S
False
```

Bounded integer sequences are immutable, and thus copies are identical. This is the same for tuples, but of course not for lists:

```
sage: T = tuple(S)
sage: copy(T) is T
True
sage: copy(S) is S
True
sage: copy(L) is L
False
```

```
>>> from sage.all import *
>>> T = tuple(S)
>>> copy(T) is T
True
>>> copy(S) is S
True
>>> copy(L) is L
False
```

Concatenation works in the same way for lists, tuples and bounded integer sequences:

```
sage: M = [randint(0,31) for i in range(5000)]
sage: T = BoundedIntegerSequence(32, M)
sage: list(S+T)==L+M
True
sage: list(T+S)==M+L
True
sage: (T+S == S+T) == (M+L == L+M)
True
```

```
>>> from sage.all import *
>>> M = [randint(Integer(0),Integer(31)) for i in range(Integer(5000))]
>>> T = BoundedIntegerSequence(Integer(32), M)
>>> list(S+T)==L+M
True
>>> list(T+S)==M+L
True
>>> (T+S == S+T) == (M+L == L+M)
True
```

However, comparison works different for lists and bounded integer sequences. Bounded integer sequences are first compared by bound, then by length, and eventually by *reverse* lexicographical ordering:

```
sage: S = BoundedIntegerSequence(21, [4,1,6,2,7,20,9])
sage: T = BoundedIntegerSequence(51, [4,1,6,2,7,20])
sage: S < T   # compare by bound, not length
True
sage: T < S
False
sage: S.bound() < T.bound()
True
sage: len(S) > len(T)
True
```

```
>>> from sage.all import *
>>> S = BoundedIntegerSequence(Integer(21), [Integer(4),Integer(1),Integer(6),
↪Integer(2),Integer(7),Integer(20),Integer(9)])
>>> T = BoundedIntegerSequence(Integer(51), [Integer(4),Integer(1),Integer(6),
↪Integer(2),Integer(7),Integer(20)])
>>> S < T   # compare by bound, not length
True
>>> T < S
False
>>> S.bound() < T.bound()
True
>>> len(S) > len(T)
True
```

```
sage: T = BoundedIntegerSequence(21, [0,0,0,0,0,0,0,0])
sage: S < T    # compare by length, not lexicographically
True
sage: T < S
```

(continues on next page)

(continued from previous page)

```
False
sage: list(T) < list(S)
True
sage: len(T) > len(S)
True
```

```
>>> from sage.all import *
>>> T = BoundedIntegerSequence(Integer(21), [Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)])
>>> S < T    # compare by length, not lexicographically
True
>>> T < S
False
>>> list(T) < list(S)
True
>>> len(T) > len(S)
True
```

```
sage: T = BoundedIntegerSequence(21, [4,1,5,2,8,20,9])
sage: T > S   # compare by reverse lexicographic ordering...
True
sage: S > T
False
sage: len(S) == len(T)
True
sage: list(S) > list(T) # direct lexicographic ordering is different
True
```

```
>>> from sage.all import *
>>> T = BoundedIntegerSequence(Integer(21), [Integer(4),Integer(1),Integer(5),
↪Integer(2),Integer(8),Integer(20),Integer(9)])
>>> T > S   # compare by reverse lexicographic ordering...
True
>>> S > T
False
>>> len(S) == len(T)
True
>>> list(S) > list(T) # direct lexicographic ordering is different
True
```

bound()

Return the bound of this bounded integer sequence.

All items of this sequence are nonnegative integers less than the returned bound. The bound is a power of two.

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
sage: S = BoundedIntegerSequence(21, [4,1,6,2,7,20,9])
sage: T = BoundedIntegerSequence(51, [4,1,6,2,7,20,9])
```

(continues on next page)

(continued from previous page)

```
sage: S.bound()
32
sage: T.bound()
64
```

```
>>> from sage.all import *
>>> from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
>>> S = BoundedIntegerSequence(Integer(21), [Integer(4),Integer(1),Integer(6),
↪Integer(2),Integer(7),Integer(20),Integer(9)])
>>> T = BoundedIntegerSequence(Integer(51), [Integer(4),Integer(1),Integer(6),
↪Integer(2),Integer(7),Integer(20),Integer(9)])
>>> S.bound()
32
>>> T.bound()
64
```

index(*other*)

The index of a given item or sub-sequence of `self`.

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
sage: S = BoundedIntegerSequence(21, [4,1,6,2,6,20,9,0])
sage: S.index(6)
2
sage: S.index(5)
Traceback (most recent call last):
...
ValueError: 5 is not in sequence
sage: S.index(BoundedIntegerSequence(21, [6, 2, 6]))
2
sage: S.index(BoundedIntegerSequence(21, [6, 2, 7]))
Traceback (most recent call last):
...
ValueError: not a sub-sequence
```

```
>>> from sage.all import *
>>> from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
>>> S = BoundedIntegerSequence(Integer(21), [Integer(4),Integer(1),Integer(6),
↪Integer(2),Integer(6),Integer(20),Integer(9),Integer(0)])
>>> S.index(Integer(6))
2
>>> S.index(Integer(5))
Traceback (most recent call last):
...
ValueError: 5 is not in sequence
>>> S.index(BoundedIntegerSequence(Integer(21), [Integer(6), Integer(2),␣
↪Integer(6)]))
2
```

(continues on next page)

(continued from previous page)

```
>>> S.index(BoundedIntegerSequence(Integer(21), [Integer(6), Integer(2),␣
↪Integer(7)]))
Traceback (most recent call last):
...
ValueError: not a sub-sequence
```

The bound of (sub-)sequences matters:

```
sage: S.index(BoundedIntegerSequence(51, [6, 2, 6]))
Traceback (most recent call last):
...
ValueError: not a sub-sequence
sage: S.index(0)
7
sage: S.index(S.bound())
Traceback (most recent call last):
...
ValueError: 32 is not in sequence
```

```
>>> from sage.all import *
>>> S.index(BoundedIntegerSequence(Integer(51), [Integer(6), Integer(2),␣
↪Integer(6)]))
Traceback (most recent call last):
...
ValueError: not a sub-sequence
>>> S.index(Integer(0))
7
>>> S.index(S.bound())
Traceback (most recent call last):
...
ValueError: 32 is not in sequence
```

list()

Convert this bounded integer sequence to a list.

NOTE:

A conversion to a list is also possible by iterating over the sequence.

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
sage: L = [randint(0,26) for i in range(5000)]
sage: S = BoundedIntegerSequence(32, L)
sage: S.list() == list(S) == L
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
>>> L = [randint(Integer(0),Integer(26)) for i in range(Integer(5000))]
>>> S = BoundedIntegerSequence(Integer(32), L)
```

(continues on next page)

(continued from previous page)

```
>>> S.list() == list(S) == L
True
```

The discussion at Issue #15820 explains why the following is a good test:

```
sage: (BoundedIntegerSequence(21, [0,0]) + BoundedIntegerSequence(21, [0,0])).
↪list()
[0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> (BoundedIntegerSequence(Integer(21), [Integer(0),Integer(0)]) +␣
↪BoundedIntegerSequence(Integer(21), [Integer(0),Integer(0)])).list()
[0, 0, 0, 0]
```

maximal_overlap(*other*)

Return `self`'s maximal trailing sub-sequence that `other` starts with.

Return `None` if there is no overlap.

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
sage: X = BoundedIntegerSequence(21, [4,1,6,2,7,2,3])
sage: S = BoundedIntegerSequence(21, [0,0,0,0,0,0,0])
sage: T = BoundedIntegerSequence(21, [2,7,2,3,0,0,0,0,0,0,0,1])
sage: (X+S).maximal_overlap(T)
<2, 7, 2, 3, 0, 0, 0, 0, 0, 0, 0>
sage: print((X+S).maximal_overlap(BoundedIntegerSequence(21, [2,7,2,3,0,0,0,0,
↪0,1])))
None
sage: (X+S).maximal_overlap(BoundedIntegerSequence(21, [0,0]))
<0, 0>
sage: B1 = BoundedIntegerSequence(4,[1,2,3,2,3,2,3])
sage: B2 = BoundedIntegerSequence(4,[2,3,2,3,2,3,1])
sage: B1.maximal_overlap(B2)
<2, 3, 2, 3, 2, 3>
```

```
>>> from sage.all import *
>>> from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
>>> X = BoundedIntegerSequence(Integer(21), [Integer(4),Integer(1),Integer(6),
↪Integer(2),Integer(7),Integer(2),Integer(3)])
>>> S = BoundedIntegerSequence(Integer(21), [Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0)])
>>> T = BoundedIntegerSequence(Integer(21), [Integer(2),Integer(7),Integer(2),
↪Integer(3),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(1)])
>>> (X+S).maximal_overlap(T)
<2, 7, 2, 3, 0, 0, 0, 0, 0, 0, 0>
>>> print((X+S).maximal_overlap(BoundedIntegerSequence(Integer(21),␣
↪[Integer(2),Integer(7),Integer(2),Integer(3),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(1)])))
```

(continues on next page)

(continued from previous page)

```
None
>>> (X+S).maximal_overlap(BoundedIntegerSequence(Integer(21), [Integer(0),
↪Integer(0)]))
<0, 0>
>>> B1 = BoundedIntegerSequence(Integer(4),[Integer(1),Integer(2),Integer(3),
↪Integer(2),Integer(3),Integer(2),Integer(3)])
>>> B2 = BoundedIntegerSequence(Integer(4),[Integer(2),Integer(3),Integer(2),
↪Integer(3),Integer(2),Integer(3),Integer(1)])
>>> B1.maximal_overlap(B2)
<2, 3, 2, 3, 2, 3>
```

startswith(*other*)

Tells whether `self` starts with a given bounded integer sequence

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
sage: L = [randint(0,26) for i in range(5000)]
sage: S = BoundedIntegerSequence(27, L)
sage: L0 = L[:1000]
sage: T = BoundedIntegerSequence(27, L0)
sage: S.startswith(T)
True
sage: L0[-1] = (L0[-1] + 1) % 27
sage: T = BoundedIntegerSequence(27, L0)
sage: S.startswith(T)
False
sage: L0[-1] = (L0[-1] - 1) % 27
sage: L0[0] = (L0[0] + 1) % 27
sage: T = BoundedIntegerSequence(27, L0)
sage: S.startswith(T)
False
sage: L0[0] = (L0[0] - 1) % 27
```

```
>>> from sage.all import *
>>> from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
>>> L = [randint(Integer(0),Integer(26)) for i in range(Integer(5000))]
>>> S = BoundedIntegerSequence(Integer(27), L)
>>> L0 = L[:Integer(1000)]
>>> T = BoundedIntegerSequence(Integer(27), L0)
>>> S.startswith(T)
True
>>> L0[-Integer(1)] = (L0[-Integer(1)] + Integer(1)) % Integer(27)
>>> T = BoundedIntegerSequence(Integer(27), L0)
>>> S.startswith(T)
False
>>> L0[-Integer(1)] = (L0[-Integer(1)] - Integer(1)) % Integer(27)
>>> L0[Integer(0)] = (L0[Integer(0)] + Integer(1)) % Integer(27)
>>> T = BoundedIntegerSequence(Integer(27), L0)
>>> S.startswith(T)
```

(continues on next page)

(continued from previous page)

```
False
>>> L0[Integer(0)] = (L0[Integer(0)] - Integer(1)) % Integer(27)
```

The bounds of the sequences must be compatible, or *startswith()* returns `False`:

```
sage: T = BoundedIntegerSequence(51, L0)
sage: S.startswith(T)
False
```

```
>>> from sage.all import *
>>> T = BoundedIntegerSequence(Integer(51), L0)
>>> S.startswith(T)
False
```

sage.data_structures.bounded_integer_sequences.**NewBISEQ**(*bitset_data*, *itembitsize*, *length*)

Helper function for unpickling of *BoundedIntegerSequence*.

EXAMPLES:

```
sage: from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
sage: L = [randint(0,26) for i in range(5000)]
sage: S = BoundedIntegerSequence(32, L)
sage: loads(dumps(S)) == S    # indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.bounded_integer_sequences import␣
↪BoundedIntegerSequence
>>> L = [randint(Integer(0),Integer(26)) for i in range(Integer(5000))]
>>> S = BoundedIntegerSequence(Integer(32), L)
>>> loads(dumps(S)) == S    # indirect doctest
True
```

CHAPTER

FOUR

STREAMS

This module provides lazy implementations of basic operators on streams. The classes implemented in this module can be used to build up more complex streams for different kinds of series (Laurent, Dirichlet, etc.).

EXAMPLES:

Streams can be used as data structure for lazy Laurent series:

```
sage: L.<z> = LazyLaurentSeriesRing(ZZ)
sage: f = L(lambda n: n, valuation=0)
sage: f
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
sage: type(f._coeff_stream)
<class 'sage.data_structures.stream.Stream_function'>
```

```
>>> from sage.all import *
>>> L = LazyLaurentSeriesRing(ZZ, names=('z',)); (z,) = L._first_ngens(1)
>>> f = L(lambda n: n, valuation=Integer(0))
>>> f
z + 2*z^2 + 3*z^3 + 4*z^4 + 5*z^5 + 6*z^6 + O(z^7)
>>> type(f._coeff_stream)
<class 'sage.data_structures.stream.Stream_function'>
```

There are basic unary and binary operators available for streams. For example, we can add two streams:

```
sage: from sage.data_structures.stream import *
sage: f = Stream_function(lambda n: n, True, 0)
sage: [f[i] for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: g = Stream_function(lambda n: 1, True, 0)
sage: [g[i] for i in range(10)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: h = Stream_add(f, g, True)
sage: [h[i] for i in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import *
>>> f = Stream_function(lambda n: n, True, Integer(0))
>>> [f[i] for i in range(Integer(10))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> g = Stream_function(lambda n: Integer(1), True, Integer(0))
```

(continues on next page)

(continued from previous page)

```
>>> [g[i] for i in range(Integer(10))]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>> h = Stream_add(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We can subtract one stream from another:

```
sage: h = Stream_sub(f, g, True)
sage: [h[i] for i in range(10)]
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> from sage.all import *
>>> h = Stream_sub(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

There is a Cauchy product on streams:

```
sage: h = Stream_cauchy_mul(f, g, True)
sage: [h[i] for i in range(10)]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

```
>>> from sage.all import *
>>> h = Stream_cauchy_mul(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

We can compute the inverse corresponding to the Cauchy product:

```
sage: ginv = Stream_cauchy_invert(g)
sage: h = Stream_cauchy_mul(f, ginv, True)
sage: [h[i] for i in range(10)]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> ginv = Stream_cauchy_invert(g)
>>> h = Stream_cauchy_mul(f, ginv, True)
>>> [h[i] for i in range(Integer(10))]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Two streams can be composed:

```
sage: g = Stream_function(lambda n: n, True, 1)
sage: h = Stream_cauchy_compose(f, g, True)
sage: [h[i] for i in range(10)]
[0, 1, 4, 14, 46, 145, 444, 1331, 3926, 11434]
```

```
>>> from sage.all import *
>>> g = Stream_function(lambda n: n, True, Integer(1))
>>> h = Stream_cauchy_compose(f, g, True)
```

(continues on next page)

(continued from previous page)

```
>>> [h[i] for i in range(Integer(10))]
[0, 1, 4, 14, 46, 145, 444, 1331, 3926, 11434]
```

There is a unary negation operator:

```
sage: h = Stream_neg(f, True)
sage: [h[i] for i in range(10)]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

```
>>> from sage.all import *
>>> h = Stream_neg(f, True)
>>> [h[i] for i in range(Integer(10))]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

More generally, we can multiply by a scalar:

```
sage: h = Stream_lmul(f, 2, True)
sage: [h[i] for i in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
>>> from sage.all import *
>>> h = Stream_lmul(f, Integer(2), True)
>>> [h[i] for i in range(Integer(10))]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Finally, we can apply an arbitrary functions to the elements of a stream:

```
sage: h = Stream_map_coefficients(f, lambda n: n^2, True)
sage: [h[i] for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> from sage.all import *
>>> h = Stream_map_coefficients(f, lambda n: n**Integer(2), True)
>>> [h[i] for i in range(Integer(10))]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

AUTHORS:

- Kwankyu Lee (2019-02-24): initial version
- Tejasvi Chebrolu, Martin Rubey, Travis Scrimshaw (2021-08): refactored and expanded functionality

class sage.data_structures.stream.**CoefficientRing**(*base_ring*)

Bases: UniqueRepresentation, FractionField_generic

The class of unknown coefficients in a stream.

gen(*i*)

Return the n-th generator of self.

The name of the generator is not to be relied on.

EXAMPLES:

```
sage: from sage.data_structures.stream import CoefficientRing
sage: PF = CoefficientRing(ZZ["q"])
sage: PF.gen(0)
FESDUMMY_0
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import CoefficientRing
>>> PF = CoefficientRing(ZZ["q"])
>>> PF.gen(Integer(0))
FESDUMMY_0
```

class sage.data_structures.stream.**DominatingAction**

Bases: Action

The action defined by G acting on S by any operation such that the result is either in G if S is in the base ring of G or G is the coefficient ring of S otherwise.

This is meant specifically for use by *CoefficientRing* as part of the function solver. This is not a mathematically defined action of G on S since the result might not be in S.

class sage.data_structures.stream.**Stream**(*true_order*)

Bases: object

Abstract base class for all streams.

INPUT:

- true_order – boolean; if the approximate order is the actual order

Note

An implementation of a stream class depending on other stream classes must not access coefficients or the approximate order of these, in order not to interfere with lazy definitions for *Stream_uninitialized*.

If an approximate order or even the true order is known, it must be set after calling super().__init__.

Otherwise, a lazy attribute _approximate_order has to be defined. Any initialization code depending on the approximate orders of input streams can be put into this definition.

However, keep in mind that (trivially) this initialization code is not executed if _approximate_order is set to a value before it is accessed.

input_streams()

Return the list of streams which are used to compute the coefficients of self.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_zero
sage: z = Stream_zero()
sage: z.input_streams()
[]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_zero
>>> z = Stream_zero()
```

(continues on next page)

(continued from previous page)

```
>>> z.input_streams()
[]
```

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

The default implementation is `False`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream
sage: CS = Stream(1)
sage: CS.is_nonzero()
False
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream
>>> CS = Stream(Integer(1))
>>> CS.is_nonzero()
False
```

is_uninitialized()

Return `True` if `self` is an uninitialized stream.

The default implementation is `False`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_zero
sage: zero = Stream_zero()
sage: zero.is_uninitialized()
False
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_zero
>>> zero = Stream_zero()
>>> zero.is_uninitialized()
False
```

order()

Return the order of `self`, which is the minimum index `n` such that `self[n]` is non-zero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function
sage: f = Stream_function(lambda n: n, True, 0)
sage: f.order()
1
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function
>>> f = Stream_function(lambda n: n, True, Integer(0))
>>> f.order()
1
```

class sage.data_structures.stream.**Stream_add**(*left*, *right*, *is_sparse*)

Bases: *Stream_binaryCommutative*

Operator for addition of two coefficient streams.

INPUT:

- `left` – *Stream* of coefficients on the left side of the operator
- `right` – *Stream* of coefficients on the right side of the operator
- `is_sparse` – boolean; whether the implementation of the stream is sparse

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_add, Stream_function)
sage: f = Stream_function(lambda n: n, True, 0)
sage: g = Stream_function(lambda n: 1, True, 0)
sage: h = Stream_add(f, g, True)
sage: [h[i] for i in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: u = Stream_add(g, f, True)
sage: [u[i] for i in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_add, Stream_function)
>>> f = Stream_function(lambda n: n, True, Integer(0))
>>> g = Stream_function(lambda n: Integer(1), True, Integer(0))
>>> h = Stream_add(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> u = Stream_add(g, f, True)
>>> [u[i] for i in range(Integer(10))]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_function, Stream_add)
sage: f = Stream_function(lambda n: n, True, 0)
sage: g = Stream_function(lambda n: n^2, True, 0)
sage: h = Stream_add(f, g, True)
sage: h.get_coefficient(5)
30
sage: [h.get_coefficient(i) for i in range(10)]
[0, 2, 6, 12, 20, 30, 42, 56, 72, 90]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_function, Stream_add)
>>> f = Stream_function(lambda n: n, True, Integer(0))
```

(continues on next page)

(continued from previous page)

```
>>> g = Stream_function(lambda n: n**Integer(2), True, Integer(0))
>>> h = Stream_add(f, g, True)
>>> h.get_coefficient(Integer(5))
30
>>> [h.get_coefficient(i) for i in range(Integer(10))]
[0, 2, 6, 12, 20, 30, 42, 56, 72, 90]
```

class sage.data_structures.stream.**Stream_binary**(*left*, *right*, *is_sparse*)

Bases: *Stream_inexact*

Base class for binary operators on coefficient streams.

INPUT:

- `left` – *Stream* for the left side of the operator
- `right` – *Stream* for the right side of the operator

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_function, Stream_add,␣
↪Stream_sub)
sage: f = Stream_function(lambda n: 2*n, True, 0)
sage: g = Stream_function(lambda n: n, True, 1)
sage: h = Stream_add(f, g, True)
sage: [h[i] for i in range(10)]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
sage: h = Stream_sub(f, g, True)
sage: [h[i] for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_function, Stream_add, Stream_
↪sub)
>>> f = Stream_function(lambda n: Integer(2)*n, True, Integer(0))
>>> g = Stream_function(lambda n: n, True, Integer(1))
>>> h = Stream_add(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
>>> h = Stream_sub(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

input_streams()

Return the list of streams which are used to compute the coefficients of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_add
sage: l = Stream_function(lambda n: n, False, 1)
sage: r = Stream_function(lambda n: n^2, False, 1)
sage: M = Stream_add(l, r, False)
sage: M.input_streams()
[<sage.data_structures.stream.Stream_function object at ...>,
 <sage.data_structures.stream.Stream_function object at ...>]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_add
>>> l = Stream_function(lambda n: n, False, Integer(1))
>>> r = Stream_function(lambda n: n**Integer(2), False, Integer(1))
>>> M = Stream_add(l, r, False)
>>> M.input_streams()
[<sage.data_structures.stream.Stream_function object at ...>,
 <sage.data_structures.stream.Stream_function object at ...>]
```

is_uninitialized()

Return `True` if `self` is an uninitialized stream.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized, Stream_
↪sub, Stream_function
sage: C = Stream_uninitialized(0)
sage: F = Stream_function(lambda n: n, True, 0)
sage: B = Stream_sub(F, C, True)
sage: B.is_uninitialized()
True
sage: Bp = Stream_sub(F, F, True)
sage: Bp.is_uninitialized()
False
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized, Stream_sub,␣
↪Stream_function
>>> C = Stream_uninitialized(Integer(0))
>>> F = Stream_function(lambda n: n, True, Integer(0))
>>> B = Stream_sub(F, C, True)
>>> B.is_uninitialized()
True
>>> Bp = Stream_sub(F, F, True)
>>> Bp.is_uninitialized()
False
```

class sage.data_structures.stream.**Stream_binaryCommutative**(*left*, *right*, *is_sparse*)

Bases: *Stream_binary*

Base class for commutative binary operators on coefficient streams.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_function, Stream_add)
sage: f = Stream_function(lambda n: 2*n, True, 0)
sage: g = Stream_function(lambda n: n, True, 1)
sage: h = Stream_add(f, g, True)
sage: [h[i] for i in range(10)]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
sage: u = Stream_add(g, f, True)
sage: [u[i] for i in range(10)]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

(continues on next page)

(continued from previous page)

```
sage: h == u
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_function, Stream_add)
>>> f = Stream_function(lambda n: Integer(2)*n, True, Integer(0))
>>> g = Stream_function(lambda n: n, True, Integer(1))
>>> h = Stream_add(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
>>> u = Stream_add(g, f, True)
>>> [u[i] for i in range(Integer(10))]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
>>> h == u
True
```

class sage.data_structures.stream.**Stream_cauchy_compose**(*f*, *g*, *is_sparse*)

Bases: *Stream_binary*

Return `f` composed by `g`.

This is the composition $(f \circ g)(z) = f(g(z))$.

INPUT:

- `f` – a *Stream*
- `g` – a *Stream* with positive order

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_cauchy_compose, Stream_
↪function
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_function(lambda n: 1, True, 1)
sage: h = Stream_cauchy_compose(f, g, True)
sage: [h[i] for i in range(10)]
[0, 1, 3, 8, 20, 48, 112, 256, 576, 1280]
sage: u = Stream_cauchy_compose(g, f, True)
sage: [u[i] for i in range(10)]
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_cauchy_compose, Stream_function
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_function(lambda n: Integer(1), True, Integer(1))
>>> h = Stream_cauchy_compose(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[0, 1, 3, 8, 20, 48, 112, 256, 576, 1280]
>>> u = Stream_cauchy_compose(g, f, True)
>>> [u[i] for i in range(Integer(10))]
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- n – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_cauchy_
↪compose
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_function(lambda n: n^2, True, 1)
sage: h = Stream_cauchy_compose(f, g, True)
sage: h[5]  # indirect doctest
527
sage: [h[i] for i in range(10)]  # indirect doctest
[0, 1, 6, 28, 124, 527, 2172, 8755, 34704, 135772]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_cauchy_
↪compose
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_function(lambda n: n**Integer(2), True, Integer(1))
>>> h = Stream_cauchy_compose(f, g, True)
>>> h[Integer(5)]  # indirect doctest
527
>>> [h[i] for i in range(Integer(10))]  # indirect doctest
[0, 1, 6, 28, 124, 527, 2172, 8755, 34704, 135772]
```

class sage.data_structures.stream.**Stream_cauchy_invert**(*series*, *approximate_order=None*)

Bases: *Stream_unary*

Operator for multiplicative inverse of the stream.

INPUT:

- series – a *Stream*
- approximate_order – None, or a lower bound on the order of the resulting stream

Instances of this class are always dense, because of mathematical necessities.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_cauchy_invert, Stream_
↪function)
sage: f = Stream_function(lambda n: 1, True, 1)
sage: g = Stream_cauchy_invert(f)
sage: [g[i] for i in range(10)]
[-1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_cauchy_invert, Stream_
↪function)
>>> f = Stream_function(lambda n: Integer(1), True, Integer(1))
>>> g = Stream_cauchy_invert(f)
>>> [g[i] for i in range(Integer(10))]
[-1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

An assumption of this class is that it is nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_cauchy_invert, Stream_
↪function)
sage: f = Stream_function(lambda n: n^2, False, 1)
sage: g = Stream_cauchy_invert(f)
sage: g.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_cauchy_invert, Stream_
↪function)
>>> f = Stream_function(lambda n: n**Integer(2), False, Integer(1))
>>> g = Stream_cauchy_invert(f)
>>> g.is_nonzero()
True
```

iterate_coefficients()

A generator for the coefficients of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_cauchy_invert, Stream_
↪function)
sage: f = Stream_function(lambda n: n^2, False, 1)
sage: g = Stream_cauchy_invert(f)
sage: n = g.iterate_coefficients()
sage: [next(n) for i in range(10)]
[1, -4, 7, -8, 8, -8, 8, -8, 8, -8]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_cauchy_invert, Stream_
↪function)
>>> f = Stream_function(lambda n: n**Integer(2), False, Integer(1))
>>> g = Stream_cauchy_invert(f)
>>> n = g.iterate_coefficients()
>>> [next(n) for i in range(Integer(10))]
[1, -4, 7, -8, 8, -8, 8, -8, 8, -8]
```

class sage.data_structures.stream.**Stream_cauchy_mul**(*left*, *right*, *is_sparse*)

Bases: *Stream_binary*

Operator for multiplication of two coefficient streams using the Cauchy product.

We are *not* assuming commutativity of the coefficient ring here, only that the coefficient ring commutes with the (implicit) variable.

INPUT:

- `left` – *Stream* of coefficients on the left side of the operator
- `right` – *Stream* of coefficients on the right side of the operator

- `is_sparse` – boolean; whether the implementation of the stream is sparse

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_cauchy_mul, Stream_function)
sage: f = Stream_function(lambda n: n, True, 0)
sage: g = Stream_function(lambda n: 1, True, 0)
sage: h = Stream_cauchy_mul(f, g, True)
sage: [h[i] for i in range(10)]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
sage: u = Stream_cauchy_mul(g, f, True)
sage: [u[i] for i in range(10)]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_cauchy_mul, Stream_function)
>>> f = Stream_function(lambda n: n, True, Integer(0))
>>> g = Stream_function(lambda n: Integer(1), True, Integer(0))
>>> h = Stream_cauchy_mul(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
>>> u = Stream_cauchy_mul(g, f, True)
>>> [u[i] for i in range(Integer(10))]
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_function, Stream_cauchy_
↪mul)
sage: f = Stream_function(lambda n: n, True, 0)
sage: g = Stream_function(lambda n: n^2, True, 0)
sage: h = Stream_cauchy_mul(f, g, True)
sage: h.get_coefficient(5)
50
sage: [h.get_coefficient(i) for i in range(10)]
[0, 0, 1, 6, 20, 50, 105, 196, 336, 540]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_function, Stream_cauchy_
↪mul)
>>> f = Stream_function(lambda n: n, True, Integer(0))
>>> g = Stream_function(lambda n: n**Integer(2), True, Integer(0))
>>> h = Stream_cauchy_mul(f, g, True)
>>> h.get_coefficient(Integer(5))
50
>>> [h.get_coefficient(i) for i in range(Integer(10))]
[0, 0, 1, 6, 20, 50, 105, 196, 336, 540]
```

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_function,
....:     Stream_cauchy_mul, Stream_cauchy_invert)
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_cauchy_mul(f, f, True)
sage: g.is_nonzero()
False
sage: fi = Stream_cauchy_invert(f)
sage: h = Stream_cauchy_mul(fi, fi, True)
sage: h.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_function,
...     Stream_cauchy_mul, Stream_cauchy_invert)
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_cauchy_mul(f, f, True)
>>> g.is_nonzero()
False
>>> fi = Stream_cauchy_invert(f)
>>> h = Stream_cauchy_mul(fi, fi, True)
>>> h.is_nonzero()
True
```

class sage.data_structures.stream.**Stream_cauchy_mul_commutative**(*left*, *right*, *is_sparse*)

Bases: *Stream_cauchy_mul*, *Stream_binaryCommutative*

Operator for multiplication of two coefficient streams using the Cauchy product for commutative multiplication of coefficients.

class sage.data_structures.stream.**Stream_derivative**(*series*, *shift*, *is_sparse*)

Bases: *Stream_unary*

Operator for taking derivatives of a non-exact stream.

Instances of this class share the cache with its input stream.

INPUT:

- `series` – a *Stream*
- `shift` – positive integer
- `is_sparse` – boolean

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_exact, Stream_derivative
sage: f = Stream_exact([1,2])
sage: Stream_derivative(f, 1, True).is_nonzero()
True
```

(continues on next page)

(continued from previous page)

```
sage: Stream_derivative(f, 2, True).is_nonzero() # it might be nice if this␣
↪gave False
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_exact, Stream_derivative
>>> f = Stream_exact([Integer(1),Integer(2)])
>>> Stream_derivative(f, Integer(1), True).is_nonzero()
True
>>> Stream_derivative(f, Integer(2), True).is_nonzero() # it might be nice if␣
↪this gave False
True
```

class sage.data_structures.stream.**Stream_dirichlet_convolve**(*left*, *right*, *is_sparse*)

Bases: *Stream_binary*

Operator for the Dirichlet convolution of two streams.

INPUT:

- `left` – *Stream* of coefficients on the left side of the operator
- `right` – *Stream* of coefficients on the right side of the operator

The coefficient of n^{-s} in the convolution of l and r equals $\sum_{k|n} l_k r_{n/k}$.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_dirichlet_convolve, Stream_
↪function, Stream_exact)
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_exact([0], constant=1)
sage: h = Stream_dirichlet_convolve(f, g, True)
sage: [h[i] for i in range(1, 10)]
[1, 3, 4, 7, 6, 12, 8, 15, 13]
sage: [sigma(n) for n in range(1, 10)]
[1, 3, 4, 7, 6, 12, 8, 15, 13]

sage: u = Stream_dirichlet_convolve(g, f, True)
sage: [u[i] for i in range(1, 10)]
[1, 3, 4, 7, 6, 12, 8, 15, 13]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_dirichlet_convolve, Stream_
↪function, Stream_exact)
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_exact([Integer(0)], constant=Integer(1))
>>> h = Stream_dirichlet_convolve(f, g, True)
>>> [h[i] for i in range(Integer(1), Integer(10))]
[1, 3, 4, 7, 6, 12, 8, 15, 13]
>>> [sigma(n) for n in range(Integer(1), Integer(10))]
[1, 3, 4, 7, 6, 12, 8, 15, 13]

>>> u = Stream_dirichlet_convolve(g, f, True)
```

(continues on next page)

(continued from previous page)

```
>>> [u[i] for i in range(Integer(1), Integer(10))]
[1, 3, 4, 7, 6, 12, 8, 15, 13]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_dirichlet_convolve,␣
↪Stream_function, Stream_exact)
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_exact([0], constant=1)
sage: h = Stream_dirichlet_convolve(f, g, True)
sage: h.get_coefficient(7)
8
sage: [h[i] for i in range(1, 10)]
[1, 3, 4, 7, 6, 12, 8, 15, 13]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_dirichlet_convolve,␣
↪Stream_function, Stream_exact)
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_exact([Integer(0)], constant=Integer(1))
>>> h = Stream_dirichlet_convolve(f, g, True)
>>> h.get_coefficient(Integer(7))
8
>>> [h[i] for i in range(Integer(1), Integer(10))]
[1, 3, 4, 7, 6, 12, 8, 15, 13]
```

class sage.data_structures.stream.**Stream_dirichlet_invert**(*series*, *is_sparse*)

Bases: *Stream_unary*

Operator for inverse with respect to Dirichlet convolution of the stream.

INPUT:

- `series` – a *Stream*

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_dirichlet_invert, Stream_
↪function)
sage: f = Stream_function(lambda n: 1, True, 1)
sage: g = Stream_dirichlet_invert(f, True)
sage: [g[i] for i in range(10)]
[0, 1, -1, -1, 0, -1, 1, -1, 0, 0]
sage: [moebius(i) for i in range(10)]                                          #␣
↪needs sage.libs.pari
[0, 1, -1, -1, 0, -1, 1, -1, 0, 0]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_dirichlet_invert, Stream_
↪function)
>>> f = Stream_function(lambda n: Integer(1), True, Integer(1))
>>> g = Stream_dirichlet_invert(f, True)
>>> [g[i] for i in range(Integer(10))]
[0, 1, -1, -1, 0, -1, 1, -1, 0, 0]
>>> [moebius(i) for i in range(Integer(10))]                                    ␣
↪    # needs sage.libs.pari
[0, 1, -1, -1, 0, -1, 1, -1, 0, 0]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_exact, Stream_dirichlet_
↪invert)
sage: f = Stream_exact([0, 3], constant=2)
sage: g = Stream_dirichlet_invert(f, True)
sage: g.get_coefficient(6)
2/27
sage: [g[i] for i in range(8)]
[0, 1/3, -2/9, -2/9, -2/27, -2/9, 2/27, -2/9]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_exact, Stream_dirichlet_
↪invert)
>>> f = Stream_exact([Integer(0), Integer(3)], constant=Integer(2))
>>> g = Stream_dirichlet_invert(f, True)
>>> g.get_coefficient(Integer(6))
2/27
>>> [g[i] for i in range(Integer(8))]
[0, 1/3, -2/9, -2/9, -2/27, -2/9, 2/27, -2/9]
```

class sage.data_structures.stream.**Stream_exact**(*initial_coefficients*, *constant=None*, *degree=None*, *order=None*)

Bases: *Stream*

A stream of eventually constant coefficients.

INPUT:

- `initial_values` – list of initial values
- `is_sparse` – boolean; specifies whether the stream is sparse
- `order` – integer (default: 0); determining the degree of the first element of `initial_values`
- `degree` – integer (optional); determining the degree of the first element which is known to be equal to `constant`
- `constant` – integer (default: 0); the coefficient of every index larger than or equal to `degree`

Warning

The convention for `order` is different to the one in `sage.rings.lazy_series_ring.LazySeriesRing`, where the input is shifted to have the prescribed order.

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

An assumption of this class is that it is nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_exact
sage: s = Stream_exact([2], order=-1, degree=2, constant=1)
sage: s.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_exact
>>> s = Stream_exact([Integer(2)], order=-Integer(1), degree=Integer(2),␣
↪constant=Integer(1))
>>> s.is_nonzero()
True
```

order()

Return the order of `self`, which is the minimum index `n` such that `self[n]` is nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_exact
sage: s = Stream_exact([1])
sage: s.order()
0
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_exact
>>> s = Stream_exact([Integer(1)])
>>> s.order()
0
```

class sage.data_structures.stream.**Stream_function**(*function*, *is_sparse*, *approximate_order*, *true_order=False*)

Bases: *Stream_inexact*

Class that creates a stream from a function on the integers.

INPUT:

- `function` – a function that generates the coefficients of the stream
- `is_sparse` – boolean; specifies whether the stream is sparse
- `approximate_order` – integer; a lower bound for the order of the stream

Note

We assume for equality that `function` is a function in the mathematical sense.

Warning

To make `sage.rings.lazy_series_ring.LazySeriesRing.define_implicitly()` work any streams used in `function` must appear in its `__closure__` as instances of *Stream*, as opposed to, for example, as instances of `LazyPowerSeries`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function
sage: f = Stream_function(lambda n: n^2, False, 1)
sage: f[3]
9
sage: [f[i] for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

sage: f = Stream_function(lambda n: 1, False, 0)
sage: n = f.iterate_coefficients()
sage: [next(n) for _ in range(10)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

sage: f = Stream_function(lambda n: n, True, 0)
sage: f[4]
4
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function
>>> f = Stream_function(lambda n: n**Integer(2), False, Integer(1))
>>> f[Integer(3)]
9
>>> [f[i] for i in range(Integer(10))]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> f = Stream_function(lambda n: Integer(1), False, Integer(0))
>>> n = f.iterate_coefficients()
>>> [next(n) for _ in range(Integer(10))]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

>>> f = Stream_function(lambda n: n, True, Integer(0))
>>> f[Integer(4)]
4
```

input_streams()

Return the list of streams which are used to compute the coefficients of `self`, as provided.

EXAMPLES:

Only streams that appear in the closure are detected:

```
sage: from sage.data_structures.stream import Stream_function, Stream_exact
sage: f = Stream_exact([1,3,5], constant=7)
sage: g = Stream_function(lambda n: f[n]^2, False, 0)
sage: g.input_streams()
[]

sage: def fun():
....:     f = Stream_exact([1,3,5], constant=7)
....:     g = Stream_function(lambda n: f[n]^2, False, 0)
....:     return g.input_streams()
sage: fun()
[<sage.data_structures.stream.Stream_exact object at 0x...>]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_exact
>>> f = Stream_exact([Integer(1),Integer(3),Integer(5)], constant=Integer(7))
>>> g = Stream_function(lambda n: f[n]**Integer(2), False, Integer(0))
>>> g.input_streams()
[]

>>> def fun():
...     f = Stream_exact([Integer(1),Integer(3),Integer(5)],␣
↪constant=Integer(7))
...     g = Stream_function(lambda n: f[n]**Integer(2), False, Integer(0))
...     return g.input_streams()
>>> fun()
[<sage.data_structures.stream.Stream_exact object at 0x...>]
```

class sage.data_structures.stream.**Stream_inexact**(*is_sparse*, *true_order*)

Bases: *Stream*

An abstract base class for the stream when we do not know it is eventually constant.

In particular, a cache is provided.

INPUT:

- `is_sparse` – boolean; whether the implementation of the stream is sparse
- `true_order` – boolean; if the approximate order is the actual order

If the cache is dense, it begins with the first nonzero term.

is_nonzero()

Return `True` if and only if the cache contains a nonzero element.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function
sage: CS = Stream_function(lambda n: 1/n, False, 1)
sage: CS.is_nonzero()
False
sage: CS[1]
1
sage: CS.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function
>>> CS = Stream_function(lambda n: Integer(1)/n, False, Integer(1))
>>> CS.is_nonzero()
False
>>> CS[Integer(1)]
1
>>> CS.is_nonzero()
True
```

iterate_coefficients()

A generator for the coefficients of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_cauchy_
↪compose
sage: f = Stream_function(lambda n: 1, False, 1)
sage: g = Stream_function(lambda n: n^3, False, 1)
sage: h = Stream_cauchy_compose(f, g, True)
sage: n = h.iterate_coefficients()
sage: [next(n) for i in range(10)]
[1, 9, 44, 207, 991, 4752, 22769, 109089, 522676, 2504295]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_cauchy_
↪compose
>>> f = Stream_function(lambda n: Integer(1), False, Integer(1))
>>> g = Stream_function(lambda n: n**Integer(3), False, Integer(1))
>>> h = Stream_cauchy_compose(f, g, True)
>>> n = h.iterate_coefficients()
>>> [next(n) for i in range(Integer(10))]
[1, 9, 44, 207, 991, 4752, 22769, 109089, 522676, 2504295]
```

class sage.data_structures.stream.**Stream_infinite_operator**(*iterator*)

Bases: *Stream*

Stream defined by applying an operator an infinite number of times.

The `iterator` returns elements s_i to compute an infinite operator. The valuation of s_i is weakly increasing as we iterate over I and there are only finitely many terms with any fixed valuation. In particular, this *assumes* the result is nonzero.

Warning

This does not check that the input is valid.

INPUT:

- `iterator` – the iterator for the factors

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_infinite_sum
sage: L.<t> = LazyLaurentSeriesRing(QQ)
sage: it = (t^n / (1 - t) for n in PositiveIntegers())
sage: f = Stream_infinite_sum(it)
sage: f.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_infinite_sum
>>> L = LazyLaurentSeriesRing(QQ, names=('t',)); (t,) = L._first_ngens(1)
>>> it = (t**n / (Integer(1) - t) for n in PositiveIntegers())
>>> f = Stream_infinite_sum(it)
>>> f.is_nonzero()
True
```

order()

Return the order of `self`, which is the minimum index `n` such that `self[n]` is nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_infinite_sum
sage: L.<t> = LazyLaurentSeriesRing(QQ)
sage: it = (t^(5+n) / (1 - t) for n in PositiveIntegers())
sage: f = Stream_infinite_sum(it)
sage: f.order()
6
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_infinite_sum
>>> L = LazyLaurentSeriesRing(QQ, names=('t',)); (t,) = L._first_ngens(1)
>>> it = (t**(Integer(5)+n) / (Integer(1) - t) for n in PositiveIntegers())
>>> f = Stream_infinite_sum(it)
>>> f.order()
6
```

class sage.data_structures.stream.**Stream_infinite_product**(*iterator*)

Bases: *Stream_infinite_operator*

Stream defined by an infinite product.

The `iterator` returns elements p_i to compute the product $\prod_{i \in I}(1 + p_i)$. See *Stream_infinite_operator* for restrictions on the p_i.

INPUT:

- `iterator` – the iterator for the factors

apply_operator(*next_obj*)

Apply the operator to `next_obj`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_infinite_product
sage: L.<t> = LazyLaurentSeriesRing(QQ)
sage: it = (t^n / (1 - t) for n in PositiveIntegers())
```

(continues on next page)

(continued from previous page)

```
sage: f = Stream_infinite_product(it)
sage: f._advance()
sage: f._advance()  # indirect doctest
sage: f._cur
1 + t + 2*t^2 + 4*t^3 + 6*t^4 + 9*t^5 + 13*t^6 + O(t^7)
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_infinite_product
>>> L = LazyLaurentSeriesRing(QQ, names=('t',)); (t,) = L._first_ngens(1)
>>> it = (t**n / (Integer(1) - t) for n in PositiveIntegers())
>>> f = Stream_infinite_product(it)
>>> f._advance()
>>> f._advance()  # indirect doctest
>>> f._cur
1 + t + 2*t^2 + 4*t^3 + 6*t^4 + 9*t^5 + 13*t^6 + O(t^7)
```

initial(*obj*)

Set the initial data.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_infinite_product
sage: L.<t> = LazyLaurentSeriesRing(QQ)
sage: it = (t^n / (1 - t) for n in PositiveIntegers())
sage: f = Stream_infinite_product(it)
sage: f._cur is None
True
sage: f._advance()  # indirect doctest
sage: f._cur
1 + t + 2*t^2 + 3*t^3 + 4*t^4 + 5*t^5 + 6*t^6 + O(t^7)
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_infinite_product
>>> L = LazyLaurentSeriesRing(QQ, names=('t',)); (t,) = L._first_ngens(1)
>>> it = (t**n / (Integer(1) - t) for n in PositiveIntegers())
>>> f = Stream_infinite_product(it)
>>> f._cur is None
True
>>> f._advance()  # indirect doctest
>>> f._cur
1 + t + 2*t^2 + 3*t^3 + 4*t^4 + 5*t^5 + 6*t^6 + O(t^7)
```

class sage.data_structures.stream.**Stream_infinite_sum**(*iterator*)

Bases: *Stream_infinite_operator*

Stream defined by an infinite sum.

The `iterator` returns elements s_i to compute the product $\sum_{i \in I} s_i$. See *Stream_infinite_operator* for restrictions on the s_i.

INPUT:

- `iterator` – the iterator for the factors

apply_operator(*next_obj*)

Apply the operator to `next_obj`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_infinite_sum
sage: L.<t> = LazyLaurentSeriesRing(QQ)
sage: it = (t^(n//2) / (1 - t) for n in PositiveIntegers())
sage: f = Stream_infinite_sum(it)
sage: f._advance()
sage: f._advance()  # indirect doctest
sage: f._cur
1 + 3*t + 4*t^2 + 4*t^3 + 4*t^4 + O(t^5)
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_infinite_sum
>>> L = LazyLaurentSeriesRing(QQ, names=('t',)); (t,) = L._first_ngens(1)
>>> it = (t**(n//Integer(2)) / (Integer(1) - t) for n in PositiveIntegers())
>>> f = Stream_infinite_sum(it)
>>> f._advance()
>>> f._advance()  # indirect doctest
>>> f._cur
1 + 3*t + 4*t^2 + 4*t^3 + 4*t^4 + O(t^5)
```

initial(*obj*)

Set the initial data.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_infinite_sum
sage: L.<t> = LazyLaurentSeriesRing(QQ)
sage: it = (t^n / (1 - t) for n in PositiveIntegers())
sage: f = Stream_infinite_sum(it)
sage: f._cur is None
True
sage: f._advance()  # indirect doctest
sage: f._cur
t + 2*t^2 + 2*t^3 + 2*t^4 + O(t^5)
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_infinite_sum
>>> L = LazyLaurentSeriesRing(QQ, names=('t',)); (t,) = L._first_ngens(1)
>>> it = (t**n / (Integer(1) - t) for n in PositiveIntegers())
>>> f = Stream_infinite_sum(it)
>>> f._cur is None
True
>>> f._advance()  # indirect doctest
>>> f._cur
t + 2*t^2 + 2*t^3 + 2*t^4 + O(t^5)
```

class sage.data_structures.stream.**Stream_integral**(*series*, *integration_constants*, *is_sparse*)

Bases: *Stream_unary*

Operator for taking integrals of a non-exact stream.

INPUT:

- `series` – a *Stream*
- `integration_constants` – list of integration constants
- `is_sparse` – boolean

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_integral
sage: f = Stream_function(lambda n: n + 1, True, -3)
sage: [f[i] for i in range(-3, 4)]
[-2, -1, 0, 1, 2, 3, 4]
sage: f2 = Stream_integral(f, [0], True)
sage: [f2.get_coefficient(i) for i in range(-3, 5)]
[0, 1, 1, 0, 1, 1, 1, 1]

sage: f = Stream_function(lambda n: (n + 1)*(n+2), True, 2)
sage: [f[i] for i in range(-1, 4)]
[0, 0, 0, 12, 20]
sage: f2 = Stream_integral(f, [-1, -1, -1], True)
sage: [f2.get_coefficient(i) for i in range(-1, 7)]
[0, -1, -1, -1/2, 0, 0, 1/5, 1/6]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_integral
>>> f = Stream_function(lambda n: n + Integer(1), True, -Integer(3))
>>> [f[i] for i in range(-Integer(3), Integer(4))]
[-2, -1, 0, 1, 2, 3, 4]
>>> f2 = Stream_integral(f, [Integer(0)], True)
>>> [f2.get_coefficient(i) for i in range(-Integer(3), Integer(5))]
[0, 1, 1, 0, 1, 1, 1, 1]

>>> f = Stream_function(lambda n: (n + Integer(1))*(n+Integer(2)), True,␣
↪Integer(2))
>>> [f[i] for i in range(-Integer(1), Integer(4))]
[0, 0, 0, 12, 20]
>>> f2 = Stream_integral(f, [-Integer(1), -Integer(1), -Integer(1)], True)
>>> [f2.get_coefficient(i) for i in range(-Integer(1), Integer(7))]
[0, -1, -1, -1/2, 0, 0, 1/5, 1/6]
```

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_integral
sage: f = Stream_function(lambda n: 2*n, True, 1)
sage: f[1]
2
sage: f.is_nonzero()
True
```

(continues on next page)

(continued from previous page)

```
sage: Stream_integral(f, [0], True).is_nonzero()
True
sage: f = Stream_function(lambda n: 0, False, 1)
sage: Stream_integral(f, [0, 0, 0], False).is_nonzero()
False
sage: Stream_integral(f, [0, 2], False).is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_integral
>>> f = Stream_function(lambda n: Integer(2)*n, True, Integer(1))
>>> f[Integer(1)]
2
>>> f.is_nonzero()
True
>>> Stream_integral(f, [Integer(0)], True).is_nonzero()
True
>>> f = Stream_function(lambda n: Integer(0), False, Integer(1))
>>> Stream_integral(f, [Integer(0), Integer(0), Integer(0)], False).is_
↪nonzero()
False
>>> Stream_integral(f, [Integer(0), Integer(2)], False).is_nonzero()
True
```

class sage.data_structures.stream.**Stream_iterator**(*iter*, *approximate_order*, *true_order=False*)

Bases: *Stream_inexact*

Class that creates a stream from an iterator.

INPUT:

- `iter` – a function that generates the coefficients of the stream
- `approximate_order` – integer; a lower bound for the order of the stream

Instances of this class are always dense.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_iterator
sage: f = Stream_iterator(iter(NonNegativeIntegers()), 0)
sage: [f[i] for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

sage: f = Stream_iterator(iter(NonNegativeIntegers()), 1)
sage: [f[i] for i in range(10)]
[0, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_iterator
>>> f = Stream_iterator(iter(NonNegativeIntegers()), Integer(0))
>>> [f[i] for i in range(Integer(10))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(continues on next page)

(continued from previous page)

```
>>> f = Stream_iterator(iter(NonNegativeIntegers()), Integer(1))
>>> [f[i] for i in range(Integer(10))]
[0, 0, 1, 2, 3, 4, 5, 6, 7, 8]
```

class sage.data_structures.stream.**Stream_lmul**(*series*, *scalar*, *is_sparse*)

Bases: *Stream_scalar*

Operator for multiplying a coefficient stream with a scalar as `self * scalar`.

INPUT:

- `series` – a *Stream*
- `scalar` – a nonzero, non-one scalar

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.data_structures.stream import (Stream_lmul, Stream_function)
sage: W = algebras.DifferentialWeyl(QQ, names=('x',))
sage: x, dx = W.gens()
sage: f = Stream_function(lambda n: x^n, True, 1)
sage: g = Stream_lmul(f, dx, True)
sage: [g[i] for i in range(5)]
[0, x*dx, x^2*dx, x^3*dx, x^4*dx]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.data_structures.stream import (Stream_lmul, Stream_function)
>>> W = algebras.DifferentialWeyl(QQ, names=('x',))
>>> x, dx = W.gens()
>>> f = Stream_function(lambda n: x**n, True, Integer(1))
>>> g = Stream_lmul(f, dx, True)
>>> [g[i] for i in range(Integer(5))]
[0, x*dx, x^2*dx, x^3*dx, x^4*dx]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_lmul, Stream_function)
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_lmul(f, 3, True)
sage: g.get_coefficient(5)
15
sage: [g.get_coefficient(i) for i in range(10)]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_lmul, Stream_function)
```

(continues on next page)

(continued from previous page)

```
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_lmul(f, Integer(3), True)
>>> g.get_coefficient(Integer(5))
15
>>> [g.get_coefficient(i) for i in range(Integer(10))]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

class sage.data_structures.stream.**Stream_map_coefficients**(*series*, *function*, *is_sparse*, *approximate_order=None*, *true_order=False*)

Bases: *Stream_unary*

The stream with `function` applied to each nonzero coefficient of `series`.

INPUT:

- `series` – a *Stream*
- `function` – a function that modifies the elements of the stream

Note

We assume for equality that `function` is a function in the mathematical sense.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_map_coefficients, Stream_
↪function)
sage: f = Stream_function(lambda n: 1, True, 1)
sage: g = Stream_map_coefficients(f, lambda n: -n, True)
sage: [g[i] for i in range(10)]
[0, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_map_coefficients, Stream_
↪function)
>>> f = Stream_function(lambda n: Integer(1), True, Integer(1))
>>> g = Stream_map_coefficients(f, lambda n: -n, True)
>>> [g[i] for i in range(Integer(10))]
[0, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_map_coefficients,␣
↪Stream_function)
sage: f = Stream_function(lambda n: n, True, -1)
sage: g = Stream_map_coefficients(f, lambda n: n^2 + 1, True)
```

(continues on next page)

(continued from previous page)

```
sage: g.get_coefficient(5)
26
sage: [g.get_coefficient(i) for i in range(-1, 10)]
[2, 0, 2, 5, 10, 17, 26, 37, 50, 65, 82]

sage: R.<x,y> = ZZ[]
sage: f = Stream_function(lambda n: n, True, -1)
sage: g = Stream_map_coefficients(f, lambda n: R(n).degree() + 1, True)
sage: [g.get_coefficient(i) for i in range(-1, 3)]
[1, 0, 1, 1]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_map_coefficients, Stream_
↪function)
>>> f = Stream_function(lambda n: n, True, -Integer(1))
>>> g = Stream_map_coefficients(f, lambda n: n**Integer(2) + Integer(1), True)
>>> g.get_coefficient(Integer(5))
26
>>> [g.get_coefficient(i) for i in range(-Integer(1), Integer(10))]
[2, 0, 2, 5, 10, 17, 26, 37, 50, 65, 82]

>>> R = ZZ['x, y']; (x, y,) = R._first_ngens(2)
>>> f = Stream_function(lambda n: n, True, -Integer(1))
>>> g = Stream_map_coefficients(f, lambda n: R(n).degree() + Integer(1), True)
>>> [g.get_coefficient(i) for i in range(-Integer(1), Integer(3))]
[1, 0, 1, 1]
```

class sage.data_structures.stream.**Stream_neg**(*series*, *is_sparse*)

Bases: *Stream_unary*

Operator for negative of the stream.

INPUT:

- `series` – a *Stream*

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_neg, Stream_function)
sage: f = Stream_function(lambda n: 1, True, 1)
sage: g = Stream_neg(f, True)
sage: [g[i] for i in range(10)]
[0, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_neg, Stream_function)
>>> f = Stream_function(lambda n: Integer(1), True, Integer(1))
>>> g = Stream_neg(f, True)
>>> [g[i] for i in range(Integer(10))]
[0, -1, -1, -1, -1, -1, -1, -1, -1, -1]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- n – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_neg, Stream_function)
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_neg(f, True)
sage: g.get_coefficient(5)
-5
sage: [g.get_coefficient(i) for i in range(10)]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_neg, Stream_function)
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_neg(f, True)
>>> g.get_coefficient(Integer(5))
-5
>>> [g.get_coefficient(i) for i in range(Integer(10))]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

is_nonzero()

Return True if and only if this stream is known to be nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_neg, Stream_function)
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_neg(f, True)
sage: g.is_nonzero()
False

sage: from sage.data_structures.stream import Stream_cauchy_invert
sage: fi = Stream_cauchy_invert(f)
sage: g = Stream_neg(fi, True)
sage: g.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_neg, Stream_function)
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_neg(f, True)
>>> g.is_nonzero()
False

>>> from sage.data_structures.stream import Stream_cauchy_invert
>>> fi = Stream_cauchy_invert(f)
>>> g = Stream_neg(fi, True)
>>> g.is_nonzero()
True
```

class sage.data_structures.stream.**Stream_plethysm**(*f*, *g*, *is_sparse*, *p*, *ring=None*, *include=None*, *exclude=None*)

Bases: *Stream_binary*

Return the plethysm of `f` composed by `g`.

This is the plethysm $f \circ g = f(g)$ when g is an element of a ring of symmetric functions.

INPUT:

- `f` – a *Stream*
- `g` – a *Stream* with positive order, unless `f` is of *Stream_exact*
- `p` – the ring of powersum symmetric functions containing `g`
- `ring` – (default: `None`) the ring the result should be in, by default `p`
- `include` – list of variables to be treated as degree one elements instead of the default degree one elements
- `exclude` – list of variables to be excluded from the default degree one elements

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.data_structures.stream import Stream_function, Stream_plethysm
sage: s = SymmetricFunctions(QQ).s()
sage: p = SymmetricFunctions(QQ).p()
sage: f = Stream_function(lambda n: s[n], True, 1)
sage: g = Stream_function(lambda n: s[[1]*n], True, 1)
sage: h = Stream_plethysm(f, g, True, p, s)
sage: [h[i] for i in range(5)]
[0,
 s[1],
 s[1, 1] + s[2],
 2*s[1, 1, 1] + s[2, 1] + s[3],
 3*s[1, 1, 1, 1] + 2*s[2, 1, 1] + s[2, 2] + s[3, 1] + s[4]]
sage: u = Stream_plethysm(g, f, True, p, s)
sage: [u[i] for i in range(5)]
[0,
 s[1],
 s[1, 1] + s[2],
 s[1, 1, 1] + s[2, 1] + 2*s[3],
 s[1, 1, 1, 1] + s[2, 1, 1] + 3*s[3, 1] + 2*s[4]]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.data_structures.stream import Stream_function, Stream_plethysm
>>> s = SymmetricFunctions(QQ).s()
>>> p = SymmetricFunctions(QQ).p()
>>> f = Stream_function(lambda n: s[n], True, Integer(1))
>>> g = Stream_function(lambda n: s[[Integer(1)]*n], True, Integer(1))
>>> h = Stream_plethysm(f, g, True, p, s)
>>> [h[i] for i in range(Integer(5))]
[0,
 s[1],
 s[1, 1] + s[2],
 2*s[1, 1, 1] + s[2, 1] + s[3],
 3*s[1, 1, 1, 1] + 2*s[2, 1, 1] + s[2, 2] + s[3, 1] + s[4]]
>>> u = Stream_plethysm(g, f, True, p, s)
>>> [u[i] for i in range(Integer(5))]
[0,
```

(continues on next page)

(continued from previous page)

```
 s[1],
 s[1, 1] + s[2],
 s[1, 1, 1] + s[2, 1] + 2*s[3],
 s[1, 1, 1, 1] + s[2, 1, 1] + 3*s[3, 1] + 2*s[4]]
```

This class also handles the plethysm of an exact stream with a stream of order 0:

```
sage: # needs sage.modules
sage: from sage.data_structures.stream import Stream_exact
sage: f = Stream_exact([s[1]], order=1)
sage: g = Stream_function(lambda n: s[n], True, 0)
sage: r = Stream_plethysm(f, g, True, p, s)
sage: [r[n] for n in range(3)]
[s[], s[1], s[2]]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.data_structures.stream import Stream_exact
>>> f = Stream_exact([s[Integer(1)]], order=Integer(1))
>>> g = Stream_function(lambda n: s[n], True, Integer(0))
>>> r = Stream_plethysm(f, g, True, p, s)
>>> [r[n] for n in range(Integer(3))]
[s[], s[1], s[2]]
```

compute_product(*n*, *la*)

Compute the product `p[la](self._right)` in degree `n`.

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.data_structures.stream import Stream_plethysm, Stream_exact,␣
 ↪Stream_function, Stream_zero
sage: s = SymmetricFunctions(QQ).s()
sage: p = SymmetricFunctions(QQ).p()
sage: f = Stream_exact([1]) # irrelevant for this test
sage: g = Stream_exact([s[2], s[3]], 0, 4, 2)
sage: h = Stream_plethysm(f, g, True, p)
sage: A = h.compute_product(7, Partition([2, 1])); A
1/12*p[2, 2, 1, 1, 1] + 1/4*p[2, 2, 2, 1] + 1/6*p[3, 2, 2]
 + 1/12*p[4, 1, 1, 1] + 1/4*p[4, 2, 1] + 1/6*p[4, 3]
sage: A == p[2, 1](s[2] + s[3]).homogeneous_component(7)
True

sage: # needs sage.modules
sage: p2 = tensor([p, p])
sage: f = Stream_exact([1]) # irrelevant for this test
sage: g = Stream_function(lambda n: sum(tensor([p[k], p[n-k]])
....:                                   for k in range(n+1)), True, 1)
sage: h = Stream_plethysm(f, g, True, p2)
sage: A = h.compute_product(7, Partition([2, 1]))
sage: B = p[2, 1](sum(g[n] for n in range(7)))
sage: B = p2.element_class(p2, {m: c for m, c in B
```

(continues on next page)

(continued from previous page)

```
....:                               if sum(mu.size() for mu in m) == 7})
sage: A == B
True

sage: # needs sage.modules
sage: f = Stream_exact([1]) # irrelevant for this test
sage: g = Stream_function(lambda n: s[n], True, 0)
sage: h = Stream_plethysm(f, g, True, p)
sage: B = p[2, 2, 1](sum(p(s[i]) for i in range(7)))
sage: all(h.compute_product(k, Partition([2, 2, 1]))
....:       == B.restrict_degree(k) for k in range(7))
True
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.data_structures.stream import Stream_plethysm, Stream_exact,␣
↪Stream_function, Stream_zero
>>> s = SymmetricFunctions(QQ).s()
>>> p = SymmetricFunctions(QQ).p()
>>> f = Stream_exact([Integer(1)]) # irrelevant for this test
>>> g = Stream_exact([s[Integer(2)], s[Integer(3)]], Integer(0), Integer(4),␣
↪Integer(2))
>>> h = Stream_plethysm(f, g, True, p)
>>> A = h.compute_product(Integer(7), Partition([Integer(2), Integer(1)])); A
1/12*p[2, 2, 1, 1, 1] + 1/4*p[2, 2, 2, 1] + 1/6*p[3, 2, 2]
 + 1/12*p[4, 1, 1, 1] + 1/4*p[4, 2, 1] + 1/6*p[4, 3]
>>> A == p[Integer(2), Integer(1)](s[Integer(2)] + s[Integer(3)]).homogeneous_
↪component(Integer(7))
True

>>> # needs sage.modules
>>> p2 = tensor([p, p])
>>> f = Stream_exact([Integer(1)]) # irrelevant for this test
>>> g = Stream_function(lambda n: sum(tensor([p[k], p[n-k]])
...                                   for k in range(n+Integer(1))), True,␣
↪Integer(1))
>>> h = Stream_plethysm(f, g, True, p2)
>>> A = h.compute_product(Integer(7), Partition([Integer(2), Integer(1)]))
>>> B = p[Integer(2), Integer(1)](sum(g[n] for n in range(Integer(7))))
>>> B = p2.element_class(p2, {m: c for m, c in B
...                           if sum(mu.size() for mu in m) == Integer(7)})
>>> A == B
True

>>> # needs sage.modules
>>> f = Stream_exact([Integer(1)]) # irrelevant for this test
>>> g = Stream_function(lambda n: s[n], True, Integer(0))
>>> h = Stream_plethysm(f, g, True, p)
>>> B = p[Integer(2), Integer(2), Integer(1)](sum(p(s[i]) for i in␣
↪range(Integer(7))))
>>> all(h.compute_product(k, Partition([Integer(2), Integer(2), Integer(1)]))
...       == B.restrict_degree(k) for k in range(Integer(7)))
```

(continues on next page)

(continued from previous page)

```
True
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.data_structures.stream import Stream_function, Stream_plethysm
sage: s = SymmetricFunctions(QQ).s()
sage: p = SymmetricFunctions(QQ).p()
sage: f = Stream_function(lambda n: s[n], True, 1)
sage: g = Stream_function(lambda n: s[[1]*n], True, 1)
sage: h = Stream_plethysm(f, g, True, p)
sage: s(h.get_coefficient(5))
4*s[1, 1, 1, 1, 1] + 4*s[2, 1, 1, 1] + 2*s[2, 2, 1] + 2*s[3, 1, 1] + s[3, 2]␣
↪+ s[4, 1] + s[5]
sage: [s(h.get_coefficient(i)) for i in range(6)]
[0,
 s[1],
 s[1, 1] + s[2],
 2*s[1, 1, 1] + s[2, 1] + s[3],
 3*s[1, 1, 1, 1] + 2*s[2, 1, 1] + s[2, 2] + s[3, 1] + s[4],
 4*s[1, 1, 1, 1, 1] + 4*s[2, 1, 1, 1] + 2*s[2, 2, 1] + 2*s[3, 1, 1] + s[3, 2]␣
↪+ s[4, 1] + s[5]]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.data_structures.stream import Stream_function, Stream_plethysm
>>> s = SymmetricFunctions(QQ).s()
>>> p = SymmetricFunctions(QQ).p()
>>> f = Stream_function(lambda n: s[n], True, Integer(1))
>>> g = Stream_function(lambda n: s[[Integer(1)]*n], True, Integer(1))
>>> h = Stream_plethysm(f, g, True, p)
>>> s(h.get_coefficient(Integer(5)))
4*s[1, 1, 1, 1, 1] + 4*s[2, 1, 1, 1] + 2*s[2, 2, 1] + 2*s[3, 1, 1] + s[3, 2]␣
↪+ s[4, 1] + s[5]
>>> [s(h.get_coefficient(i)) for i in range(Integer(6))]
[0,
 s[1],
 s[1, 1] + s[2],
 2*s[1, 1, 1] + s[2, 1] + s[3],
 3*s[1, 1, 1, 1] + 2*s[2, 1, 1] + s[2, 2] + s[3, 1] + s[4],
 4*s[1, 1, 1, 1, 1] + 4*s[2, 1, 1, 1] + 2*s[2, 2, 1] + 2*s[3, 1, 1] + s[3, 2]␣
↪+ s[4, 1] + s[5]]
```

input_streams()

Return the list of streams which are used to compute the coefficients of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_plethysm
sage: s = SymmetricFunctions(QQ).s()
sage: p = SymmetricFunctions(QQ).p()
sage: f = Stream_function(lambda n: s[n], True, 1)
sage: g = Stream_function(lambda n: s[n-1,1], True, 2)
sage: h = Stream_plethysm(f, g, True, p)
sage: h.input_streams()
[<sage.data_structures.stream.Stream_map_coefficients object at ...>]
sage: [h[i] for i in range(1, 5)]
[0,
 1/2*p[1, 1] - 1/2*p[2],
 1/3*p[1, 1, 1] - 1/3*p[3],
 1/4*p[1, 1, 1, 1] + 1/4*p[2, 2] - 1/2*p[4]]
sage: h.input_streams()
[<sage.data_structures.stream.Stream_map_coefficients object at ...>,
 <sage.data_structures.stream.Stream_cauchy_mul object at ...>]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_plethysm
>>> s = SymmetricFunctions(QQ).s()
>>> p = SymmetricFunctions(QQ).p()
>>> f = Stream_function(lambda n: s[n], True, Integer(1))
>>> g = Stream_function(lambda n: s[n-Integer(1),Integer(1)], True,␣
↪Integer(2))
>>> h = Stream_plethysm(f, g, True, p)
>>> h.input_streams()
[<sage.data_structures.stream.Stream_map_coefficients object at ...>]
>>> [h[i] for i in range(Integer(1), Integer(5))]
[0,
 1/2*p[1, 1] - 1/2*p[2],
 1/3*p[1, 1, 1] - 1/3*p[3],
 1/4*p[1, 1, 1, 1] + 1/4*p[2, 2] - 1/2*p[4]]
>>> h.input_streams()
[<sage.data_structures.stream.Stream_map_coefficients object at ...>,
 <sage.data_structures.stream.Stream_cauchy_mul object at ...>]
```

stretched_power_restrict_degree(*i*, *m*, *d*)

Return the degree `d*i` part of `p([i]*m)(g)` in terms of `self._basis`.

INPUT:

- `i`, `m` – positive integers
- `d` – integer

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.data_structures.stream import Stream_plethysm, Stream_exact,␣
↪Stream_function, Stream_zero
sage: s = SymmetricFunctions(QQ).s()
sage: p = SymmetricFunctions(QQ).p()
sage: f = Stream_exact([1]) # irrelevant for this test
sage: g = Stream_exact([s[2], s[3]], 0, 4, 2)
```

(continues on next page)

(continued from previous page)

```
sage: h = Stream_plethysm(f, g, True, p)
sage: A = h.stretched_power_restrict_degree(2, 3, 6)
sage: A == p[2,2,2](s[2] + s[3]).homogeneous_component(12)
True

sage: # needs sage.modules
sage: p2 = tensor([p, p])
sage: f = Stream_exact([1]) # irrelevant for this test
sage: g = Stream_function(lambda n: sum(tensor([p[k], p[n-k]])
....:                                   for k in range(n+1)), True, 1)
sage: h = Stream_plethysm(f, g, True, p2)
sage: A = h.stretched_power_restrict_degree(2, 3, 6)
sage: B = p[2,2,2](sum(g[n] for n in range(7)))      # long time
sage: B = p2.element_class(p2, {m: c for m, c in B  # long time
....:                           if sum(mu.size() for mu in m) == 12})
sage: A == B                          # long time
True
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.data_structures.stream import Stream_plethysm, Stream_exact,␣
↪Stream_function, Stream_zero
>>> s = SymmetricFunctions(QQ).s()
>>> p = SymmetricFunctions(QQ).p()
>>> f = Stream_exact([Integer(1)]) # irrelevant for this test
>>> g = Stream_exact([s[Integer(2)], s[Integer(3)]], Integer(0), Integer(4),␣
↪Integer(2))
>>> h = Stream_plethysm(f, g, True, p)
>>> A = h.stretched_power_restrict_degree(Integer(2), Integer(3), Integer(6))
>>> A == p[Integer(2),Integer(2),Integer(2)](s[Integer(2)] + s[Integer(3)]).
↪homogeneous_component(Integer(12))
True

>>> # needs sage.modules
>>> p2 = tensor([p, p])
>>> f = Stream_exact([Integer(1)]) # irrelevant for this test
>>> g = Stream_function(lambda n: sum(tensor([p[k], p[n-k]])
...                                   for k in range(n+Integer(1))), True,␣
↪Integer(1))
>>> h = Stream_plethysm(f, g, True, p2)
>>> A = h.stretched_power_restrict_degree(Integer(2), Integer(3), Integer(6))
>>> B = p[Integer(2),Integer(2),Integer(2)](sum(g[n] for n in␣
↪range(Integer(7))))      # long time
>>> B = p2.element_class(p2, {m: c for m, c in B  # long time
...                           if sum(mu.size() for mu in m) == Integer(12)})
>>> A == B                          # long time
True
```

class sage.data_structures.stream.**Stream_rmul**(*series*, *scalar*, *is_sparse*)

Bases: *Stream_scalar*

Operator for multiplying a coefficient stream with a scalar as `scalar * self`.

INPUT:

- `series` – a *Stream*
- `scalar` – a nonzero, non-one scalar

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.data_structures.stream import (Stream_rmul, Stream_function)
sage: W = algebras.DifferentialWeyl(QQ, names=('x',))
sage: x, dx = W.gens()
sage: f = Stream_function(lambda n: x^n, True, 1)
sage: g = Stream_rmul(f, dx, True)
sage: [g[i] for i in range(5)]
[0, x*dx + 1, x^2*dx + 2*x, x^3*dx + 3*x^2, x^4*dx + 4*x^3]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.data_structures.stream import (Stream_rmul, Stream_function)
>>> W = algebras.DifferentialWeyl(QQ, names=('x',))
>>> x, dx = W.gens()
>>> f = Stream_function(lambda n: x**n, True, Integer(1))
>>> g = Stream_rmul(f, dx, True)
>>> [g[i] for i in range(Integer(5))]
[0, x*dx + 1, x^2*dx + 2*x, x^3*dx + 3*x^2, x^4*dx + 4*x^3]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_rmul, Stream_function)
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_rmul(f, 3, True)
sage: g.get_coefficient(5)
15
sage: [g.get_coefficient(i) for i in range(10)]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_rmul, Stream_function)
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_rmul(f, Integer(3), True)
>>> g.get_coefficient(Integer(5))
15
>>> [g.get_coefficient(i) for i in range(Integer(10))]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

class sage.data_structures.stream.**Stream_scalar**(*series*, *scalar*, *is_sparse*)

Bases: *Stream_unary*

Base class for operators multiplying a coefficient stream by a scalar.

INPUT:

- `series` – a *Stream*
- `scalar` – a nonzero, non-one scalar
- `is_sparse` – boolean

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_rmul, Stream_function)
sage: f = Stream_function(lambda n: n, True, 1)
sage: g = Stream_rmul(f, 2, True)
sage: g.is_nonzero()
False

sage: from sage.data_structures.stream import Stream_cauchy_invert
sage: fi = Stream_cauchy_invert(f)
sage: g = Stream_rmul(fi, 2, True)
sage: g.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_rmul, Stream_function)
>>> f = Stream_function(lambda n: n, True, Integer(1))
>>> g = Stream_rmul(f, Integer(2), True)
>>> g.is_nonzero()
False

>>> from sage.data_structures.stream import Stream_cauchy_invert
>>> fi = Stream_cauchy_invert(f)
>>> g = Stream_rmul(fi, Integer(2), True)
>>> g.is_nonzero()
True
```

class sage.data_structures.stream.**Stream_shift**(*series*, *shift*)

Bases: *Stream*

Operator for shifting a nonzero, non-exact stream.

Instances of this class share the cache with its input stream.

INPUT:

- `series` – a *Stream*
- `shift` – integer

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

An assumption of this class is that it is nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_cauchy_invert, Stream_
↪function)
sage: f = Stream_function(lambda n: n^2, False, 1)
sage: g = Stream_cauchy_invert(f)
sage: g.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_cauchy_invert, Stream_
↪function)
>>> f = Stream_function(lambda n: n**Integer(2), False, Integer(1))
>>> g = Stream_cauchy_invert(f)
>>> g.is_nonzero()
True
```

is_uninitialized()

Return `True` if `self` is an uninitialized stream.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized, Stream_
↪shift
sage: C = Stream_uninitialized(0)
sage: S = Stream_shift(C, 5)
sage: S.is_uninitialized()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized, Stream_shift
>>> C = Stream_uninitialized(Integer(0))
>>> S = Stream_shift(C, Integer(5))
>>> S.is_uninitialized()
True
```

order()

Return the order of `self`, which is the minimum index `n` such that `self[n]` is nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_shift
sage: s = Stream_shift(Stream_function(lambda n: n, True, 0), 2)
sage: s.order()
3
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_shift
>>> s = Stream_shift(Stream_function(lambda n: n, True, Integer(0)),␣
↪Integer(2))
>>> s.order()
3
```

class sage.data_structures.stream.**Stream_sub**(*left*, *right*, *is_sparse*)

Bases: *Stream_binary*

Operator for subtraction of two coefficient streams.

INPUT:

- `left` – *Stream* of coefficients on the left side of the operator
- `right` – *Stream* of coefficients on the right side of the operator
- `is_sparse` – boolean; whether the implementation of the stream is sparse

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_sub, Stream_function)
sage: f = Stream_function(lambda n: n, True, 0)
sage: g = Stream_function(lambda n: 1, True, 0)
sage: h = Stream_sub(f, g, True)
sage: [h[i] for i in range(10)]
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
sage: u = Stream_sub(g, f, True)
sage: [u[i] for i in range(10)]
[1, 0, -1, -2, -3, -4, -5, -6, -7, -8]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_sub, Stream_function)
>>> f = Stream_function(lambda n: n, True, Integer(0))
>>> g = Stream_function(lambda n: Integer(1), True, Integer(0))
>>> h = Stream_sub(f, g, True)
>>> [h[i] for i in range(Integer(10))]
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> u = Stream_sub(g, f, True)
>>> [u[i] for i in range(Integer(10))]
[1, 0, -1, -2, -3, -4, -5, -6, -7, -8]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_function, Stream_sub)
sage: f = Stream_function(lambda n: n, True, 0)
sage: g = Stream_function(lambda n: n^2, True, 0)
sage: h = Stream_sub(f, g, True)
sage: h.get_coefficient(5)
-20
sage: [h.get_coefficient(i) for i in range(10)]
[0, 0, -2, -6, -12, -20, -30, -42, -56, -72]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_function, Stream_sub)
>>> f = Stream_function(lambda n: n, True, Integer(0))
>>> g = Stream_function(lambda n: n**Integer(2), True, Integer(0))
>>> h = Stream_sub(f, g, True)
>>> h.get_coefficient(Integer(5))
```

(continues on next page)

(continued from previous page)

```
-20
>>> [h.get_coefficient(i) for i in range(Integer(10))]
[0, 0, -2, -6, -12, -20, -30, -42, -56, -72]
```

class sage.data_structures.stream.**Stream_taylor**(*function*, *is_sparse*)

Bases: *Stream_inexact*

Class that returns a stream for the Taylor series of a function.

INPUT:

- `function` – a function that has a `derivative` method and takes a single input
- `is_sparse` – boolean; specifies whether the stream is sparse

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_taylor
sage: g(x) = sin(x)
sage: f = Stream_taylor(g, False)
sage: f[3]
-1/6
sage: [f[i] for i in range(10)]
[0, 1, 0, -1/6, 0, 1/120, 0, -1/5040, 0, 1/362880]

sage: g(y) = cos(y)
sage: f = Stream_taylor(g, False)
sage: n = f.iterate_coefficients()
sage: [next(n) for _ in range(10)]
[1, 0, -1/2, 0, 1/24, 0, -1/720, 0, 1/40320, 0]

sage: g(z) = 1 / (1 - 2*z)
sage: f = Stream_taylor(g, True)
sage: [f[i] for i in range(4)]
[1, 2, 4, 8]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_taylor
>>> __tmp__=var("x"); g = symbolic_expression(sin(x)).function(x)
>>> f = Stream_taylor(g, False)
>>> f[Integer(3)]
-1/6
>>> [f[i] for i in range(Integer(10))]
[0, 1, 0, -1/6, 0, 1/120, 0, -1/5040, 0, 1/362880]

>>> __tmp__=var("y"); g = symbolic_expression(cos(y)).function(y)
>>> f = Stream_taylor(g, False)
>>> n = f.iterate_coefficients()
>>> [next(n) for _ in range(Integer(10))]
[1, 0, -1/2, 0, 1/24, 0, -1/720, 0, 1/40320, 0]

>>> __tmp__=var("z"); g = symbolic_expression(Integer(1) / (Integer(1) -␣
↪Integer(2)*z)).function(z)
>>> f = Stream_taylor(g, True)
```

(continues on next page)

(continued from previous page)

```
>>> [f[i] for i in range(Integer(4))]
[1, 2, 4, 8]
```

get_coefficient(*n*)

Return the `n`-th coefficient of `self`.

INPUT:

- `n` – integer; the degree for the coefficient

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_taylor
sage: g(x) = exp(x)
sage: f = Stream_taylor(g, True)
sage: f.get_coefficient(5)
1/120

sage: from sage.data_structures.stream import Stream_taylor
sage: y = SR.var('y')
sage: f = Stream_taylor(sin(y), True)
sage: f.get_coefficient(0)
0
sage: f.get_coefficient(5)
1/120
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_taylor
>>> __tmp__=var("x"); g = symbolic_expression(exp(x)).function(x)
>>> f = Stream_taylor(g, True)
>>> f.get_coefficient(Integer(5))
1/120

>>> from sage.data_structures.stream import Stream_taylor
>>> y = SR.var('y')
>>> f = Stream_taylor(sin(y), True)
>>> f.get_coefficient(Integer(0))
0
>>> f.get_coefficient(Integer(5))
1/120
```

iterate_coefficients()

A generator for the coefficients of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_taylor
sage: x = polygen(QQ, 'x')
sage: f = Stream_taylor(x^3, False)
sage: it = f.iterate_coefficients()
sage: [next(it) for _ in range(10)]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

sage: y = SR.var('y')
```

(continues on next page)

(continued from previous page)

```
sage: f = Stream_taylor(y^3, False)
sage: it = f.iterate_coefficients()
sage: [next(it) for _ in range(10)]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_taylor
>>> x = polygen(QQ, 'x')
>>> f = Stream_taylor(x**Integer(3), False)
>>> it = f.iterate_coefficients()
>>> [next(it) for _ in range(Integer(10))]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

>>> y = SR.var('y')
>>> f = Stream_taylor(y**Integer(3), False)
>>> it = f.iterate_coefficients()
>>> [next(it) for _ in range(Integer(10))]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

class sage.data_structures.stream.**Stream_truncated**(*series*, *shift*, *minimal_valuation*)

Bases: *Stream_unary*

Operator for shifting a nonzero, non-exact stream that has been shifted below its minimal valuation.

Instances of this class share the cache with its input stream.

INPUT:

- `series` – a *Stream_inexact*
- `shift` – integer
- `minimal_valuation` – integer; this is also the approximate order

is_nonzero()

Return `True` if and only if this stream is known to be nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_
↪truncated
sage: def fun(n): return 1 if ZZ(n).is_power_of(2) else 0
sage: f = Stream_function(fun, False, 0)
sage: [f[i] for i in range(4)]
[0, 1, 1, 0]
sage: f._cache
[1, 1, 0]
sage: s = Stream_truncated(f, -5, 0)
sage: s.is_nonzero()
False
sage: [f[i] for i in range(7,10)]  # updates the cache of s
[0, 1, 0]
sage: s.is_nonzero()
True
```

(continues on next page)

(continued from previous page)

```
sage: f = Stream_function(fun, True, 0)
sage: [f[i] for i in range(4)]
[0, 1, 1, 0]
sage: f._cache
{1: 1, 2: 1, 3: 0}
sage: s = Stream_truncated(f, -5, 0)
sage: s.is_nonzero()
False
sage: [f[i] for i in range(7,10)]  # updates the cache of s
[0, 1, 0]
sage: s.is_nonzero()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_truncated
>>> def fun(n): return Integer(1) if ZZ(n).is_power_of(Integer(2)) else␣
↪Integer(0)
>>> f = Stream_function(fun, False, Integer(0))
>>> [f[i] for i in range(Integer(4))]
[0, 1, 1, 0]
>>> f._cache
[1, 1, 0]
>>> s = Stream_truncated(f, -Integer(5), Integer(0))
>>> s.is_nonzero()
False
>>> [f[i] for i in range(Integer(7),Integer(10))]  # updates the cache of s
[0, 1, 0]
>>> s.is_nonzero()
True

>>> f = Stream_function(fun, True, Integer(0))
>>> [f[i] for i in range(Integer(4))]
[0, 1, 1, 0]
>>> f._cache
{1: 1, 2: 1, 3: 0}
>>> s = Stream_truncated(f, -Integer(5), Integer(0))
>>> s.is_nonzero()
False
>>> [f[i] for i in range(Integer(7),Integer(10))]  # updates the cache of s
[0, 1, 0]
>>> s.is_nonzero()
True
```

order()

Return the order of `self`, which is the minimum index `n` such that `self[n]` is nonzero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_
↪truncated
sage: def fun(n): return 1 if ZZ(n).is_power_of(2) else 0
sage: s = Stream_truncated(Stream_function(fun, True, 0), -5, 0)
```

(continues on next page)

(continued from previous page)

```
sage: s.order()
3
sage: s = Stream_truncated(Stream_function(fun, False, 0), -5, 0)
sage: s.order()
3
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_truncated
>>> def fun(n): return Integer(1) if ZZ(n).is_power_of(Integer(2)) else␣
↪Integer(0)
>>> s = Stream_truncated(Stream_function(fun, True, Integer(0)), -Integer(5),␣
↪Integer(0))
>>> s.order()
3
>>> s = Stream_truncated(Stream_function(fun, False, Integer(0)), -Integer(5),
↪ Integer(0))
>>> s.order()
3
```

Check that it also worked properly with the cache partially filled:

```
sage: f = Stream_function(fun, True, 0)
sage: dummy = [f[i] for i in range(10)]
sage: s = Stream_truncated(f, -5, 0)
sage: s.order()
3
sage: f = Stream_function(fun, False, 0)
sage: dummy = [f[i] for i in range(10)]
sage: s = Stream_truncated(f, -5, 0)
sage: s.order()
3
```

```
>>> from sage.all import *
>>> f = Stream_function(fun, True, Integer(0))
>>> dummy = [f[i] for i in range(Integer(10))]
>>> s = Stream_truncated(f, -Integer(5), Integer(0))
>>> s.order()
3
>>> f = Stream_function(fun, False, Integer(0))
>>> dummy = [f[i] for i in range(Integer(10))]
>>> s = Stream_truncated(f, -Integer(5), Integer(0))
>>> s.order()
3
```

class sage.data_structures.stream.**Stream_unary**(*series*, *is_sparse*, *true_order=False*)

Bases: *Stream_inexact*

Base class for unary operators on coefficient streams.

INPUT:

- `series` – *Stream* the operator acts on
- `is_sparse` – boolean

- `true_order` – boolean (default: `False`); if the approximate order is the actual order

EXAMPLES:

```
sage: from sage.data_structures.stream import (Stream_function, Stream_cauchy_
↪invert, Stream_lmul)
sage: f = Stream_function(lambda n: 2*n, False, 1)
sage: g = Stream_cauchy_invert(f)
sage: [g[i] for i in range(10)]
[-1, 1/2, 0, 0, 0, 0, 0, 0, 0, 0]
sage: g = Stream_lmul(f, 2, True)
sage: [g[i] for i in range(10)]
[0, 4, 8, 12, 16, 20, 24, 28, 32, 36]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import (Stream_function, Stream_cauchy_
↪invert, Stream_lmul)
>>> f = Stream_function(lambda n: Integer(2)*n, False, Integer(1))
>>> g = Stream_cauchy_invert(f)
>>> [g[i] for i in range(Integer(10))]
[-1, 1/2, 0, 0, 0, 0, 0, 0, 0, 0]
>>> g = Stream_lmul(f, Integer(2), True)
>>> [g[i] for i in range(Integer(10))]
[0, 4, 8, 12, 16, 20, 24, 28, 32, 36]
```

input_streams()

Return the list of streams which are used to compute the coefficients of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_function, Stream_neg
sage: h = Stream_function(lambda n: n, False, 1)
sage: M = Stream_neg(h, False)
sage: M.input_streams()
[<sage.data_structures.stream.Stream_function object at ...>]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_function, Stream_neg
>>> h = Stream_function(lambda n: n, False, Integer(1))
>>> M = Stream_neg(h, False)
>>> M.input_streams()
[<sage.data_structures.stream.Stream_function object at ...>]
```

is_uninitialized()

Return `True` if `self` is an uninitialized stream.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized, Stream_
↪unary
sage: C = Stream_uninitialized(0)
sage: M = Stream_unary(C, True)
sage: M.is_uninitialized()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized, Stream_unary
>>> C = Stream_uninitialized(Integer(0))
>>> M = Stream_unary(C, True)
>>> M.is_uninitialized()
True
```

class sage.data_structures.stream.**Stream_uninitialized**(*approximate_order*, *true_order=False*, *name=None*)

Bases: *Stream*

Coefficient stream for an uninitialized series.

INPUT:

- `approximate_order` – integer; a lower bound for the order of the stream
- `true_order` – boolean; if the approximate order is the actual order
- `name` – string; a name that refers to the undefined stream in error messages

Instances of this class are always dense.

Todo

Should instances of this class share the cache with its `_target`?

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized
sage: from sage.data_structures.stream import Stream_exact
sage: one = Stream_exact([1])
sage: C = Stream_uninitialized(0)
sage: C._target
sage: C.define(one)
sage: C[4]
0
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized
>>> from sage.data_structures.stream import Stream_exact
>>> one = Stream_exact([Integer(1)])
>>> C = Stream_uninitialized(Integer(0))
>>> C._target
>>> C.define(one)
>>> C[Integer(4)]
0
```

define(*target*)

Define `self` via `self = target`.

INPUT:

- `target` – a stream

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized, Stream_
↪exact, Stream_cauchy_mul, Stream_add
sage: x = Stream_exact([1], order=1)
sage: C = Stream_uninitialized(1)
sage: C.define(Stream_add(x, Stream_cauchy_mul(C, C, True), True))
sage: C[6]
42
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized, Stream_
↪exact, Stream_cauchy_mul, Stream_add
>>> x = Stream_exact([Integer(1)], order=Integer(1))
>>> C = Stream_uninitialized(Integer(1))
>>> C.define(Stream_add(x, Stream_cauchy_mul(C, C, True), True))
>>> C[Integer(6)]
42
```

define_implicitly(*series*, *initial_values*, *equations*, *base_ring*, *coefficient_ring*, *terms_of_degree*, *max_lookahead=1*)

Define `self` via `equations == 0`.

INPUT:

- `series` – a list of series
- `equations` – a list of equations defining the series
- `initial_values` – a list specifying `self[0], self[1], ...`
- `base_ring` – the base ring
- `coefficient_ring` – the ring containing the elements of the stream (after substitution)
- `terms_of_degree` – a function returning the list of terms of a given degree
- `max_lookahead` – a positive integer specifying how many elements beyond the approximate order of each equation to extract linear equations from

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized, Stream_
↪exact, Stream_cauchy_mul, Stream_add, Stream_sub
sage: terms_of_degree = lambda n, R: [R.one()]
sage: x = Stream_exact([1], order=1)
sage: C = Stream_uninitialized(1)
sage: D = Stream_add(x, Stream_cauchy_mul(C, C, True), True)
sage: eq = Stream_sub(C, D, True)
sage: C.define_implicitly([C], [], [eq], QQ, QQ, terms_of_degree)
sage: C[6]
42
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized, Stream_
↪exact, Stream_cauchy_mul, Stream_add, Stream_sub
>>> terms_of_degree = lambda n, R: [R.one()]
>>> x = Stream_exact([Integer(1)], order=Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> C = Stream_uninitialized(Integer(1))
>>> D = Stream_add(x, Stream_cauchy_mul(C, C, True), True)
>>> eq = Stream_sub(C, D, True)
>>> C.define_implicitly([C], [], [eq], QQ, QQ, terms_of_degree)
>>> C[Integer(6)]
42
```

input_streams()

Return the list of streams which are used to compute the coefficients of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized, Stream_
↪function
sage: h = Stream_function(lambda n: n, False, 1)
sage: M = Stream_uninitialized(0)
sage: M.input_streams()
[]
sage: M._target = h
sage: [h[i] for i in range(5)]
[0, 1, 2, 3, 4]
sage: M.input_streams()
[<sage.data_structures.stream.Stream_function object at ...>]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized, Stream_
↪function
>>> h = Stream_function(lambda n: n, False, Integer(1))
>>> M = Stream_uninitialized(Integer(0))
>>> M.input_streams()
[]
>>> M._target = h
>>> [h[i] for i in range(Integer(5))]
[0, 1, 2, 3, 4]
>>> M.input_streams()
[<sage.data_structures.stream.Stream_function object at ...>]
```

is_uninitialized()

Return `True` if `self` is an uninitialized stream.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized
sage: C = Stream_uninitialized(0)
sage: C.is_uninitialized()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized
>>> C = Stream_uninitialized(Integer(0))
>>> C.is_uninitialized()
True
```

A more subtle uninitialized series:

```
sage: L.<z> = LazyPowerSeriesRing(QQ)
sage: T = L.undefined(1)
sage: D = L.undefined(0)
sage: T.define(z * exp(T) * D)
sage: T._coeff_stream.is_uninitialized()
True
```

```
>>> from sage.all import *
>>> L = LazyPowerSeriesRing(QQ, names=('z',)); (z,) = L._first_ngens(1)
>>> T = L.undefined(Integer(1))
>>> D = L.undefined(Integer(0))
>>> T.define(z * exp(T) * D)
>>> T._coeff_stream.is_uninitialized()
True
```

iterate_coefficients()

A generator for the coefficients of `self`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_uninitialized
sage: from sage.data_structures.stream import Stream_exact
sage: z = Stream_exact([1], order=1)
sage: C = Stream_uninitialized(0)
sage: C._target
sage: C._target = z
sage: n = C.iterate_coefficients()
sage: [next(n) for _ in range(10)]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_uninitialized
>>> from sage.data_structures.stream import Stream_exact
>>> z = Stream_exact([Integer(1)], order=Integer(1))
>>> C = Stream_uninitialized(Integer(0))
>>> C._target
>>> C._target = z
>>> n = C.iterate_coefficients()
>>> [next(n) for _ in range(Integer(10))]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

class sage.data_structures.stream.**Stream_zero**

Bases: *Stream*

A coefficient stream that is exactly equal to zero.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_zero
sage: s = Stream_zero()
sage: s[5]
0
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_zero
>>> s = Stream_zero()
>>> s[Integer(5)]
0
```

order()

Return the order of `self`, which is `infinity`.

EXAMPLES:

```
sage: from sage.data_structures.stream import Stream_zero
sage: s = Stream_zero()
sage: s.order()
+Infinity
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import Stream_zero
>>> s = Stream_zero()
>>> s.order()
+Infinity
```

class sage.data_structures.stream.**VariablePool**(*ring*)

Bases: `UniqueRepresentation`

A class to keep track of used and unused variables in an `InfinitePolynomialRing`.

INPUT:

- `ring` – `InfinitePolynomialRing`

del_variable(*v*)

Remove `v` from the pool.

EXAMPLES:

```
sage: from sage.data_structures.stream import VariablePool
sage: R.<a> = InfinitePolynomialRing(QQ)
sage: P = VariablePool(R)
sage: v = P.new_variable(); v
a_1

sage: P.del_variable(v)
sage: v = P.new_variable(); v
a_1
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import VariablePool
>>> R = InfinitePolynomialRing(QQ, names=('a',)); (a,) = R._first_ngens(1)
>>> P = VariablePool(R)
>>> v = P.new_variable(); v
a_1

>>> P.del_variable(v)
```

(continues on next page)

(continued from previous page)

```
>>> v = P.new_variable(); v
a_1
```

new_variable(*data=None*)

Return an unused variable.

EXAMPLES:

```
sage: from sage.data_structures.stream import VariablePool
sage: R.<a> = InfinitePolynomialRing(QQ)
sage: P = VariablePool(R)
sage: P.new_variable()
a_0
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import VariablePool
>>> R = InfinitePolynomialRing(QQ, names=('a',)); (a,) = R._first_ngens(1)
>>> P = VariablePool(R)
>>> P.new_variable()
a_0
```

variables()

Return the dictionary mapping variables to data.

EXAMPLES:

```
sage: from sage.data_structures.stream import VariablePool
sage: R.<a> = InfinitePolynomialRing(QQ)
sage: P = VariablePool(R)
sage: P.new_variable("my new variable")
a_2
sage: P.variables()
{a_0: None, a_1: None, a_2: 'my new variable'}
```

```
>>> from sage.all import *
>>> from sage.data_structures.stream import VariablePool
>>> R = InfinitePolynomialRing(QQ, names=('a',)); (a,) = R._first_ngens(1)
>>> P = VariablePool(R)
>>> P.new_variable("my new variable")
a_2
>>> P.variables()
{a_0: None, a_1: None, a_2: 'my new variable'}
```

CHAPTER

FIVE

MUTABLE POSET

This module provides a class representing a finite partially ordered set (poset) for the purpose of being used as a data structure. Thus the posets introduced in this module are mutable, i.e., elements can be added and removed from a poset at any time.

To get in touch with Sage's "usual" posets, start with the page `Posets` in the reference manual.

5.1 Examples

5.1.1 First Steps

We start by creating an empty poset. This is simply done by

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P
poset()
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP()
>>> P
poset()
```

A poset should contain elements, thus let us add them with

```
sage: P.add(42)
sage: P.add(7)
sage: P.add(13)
sage: P.add(3)
```

```
>>> from sage.all import *
>>> P.add(Integer(42))
>>> P.add(Integer(7))
>>> P.add(Integer(13))
>>> P.add(Integer(3))
```

Let us look at the poset again:

```
sage: P
poset(3, 7, 13, 42)
```

```
sage: from sage.all import *
sage: P
poset(3, 7, 13, 42)
```

We see that they elements are sorted using $\leq$ which exists on the integers $\mathbf{Z}$. Since this is even a total order, we could have used a more efficient data structure. Alternatively, we can write

```
sage: MP([42, 7, 13, 3])
poset(3, 7, 13, 42)
```

```
sage: from sage.all import *
sage: MP([Integer(42), Integer(7), Integer(13), Integer(3)])
poset(3, 7, 13, 42)
```

to add several elements at once on construction.

5.1.2 A less boring Example

Let us continue with a less boring example. We define the class

```
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
```

```
sage: from sage.all import *
sage: class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
```

It is equipped with a $\leq$-operation such that $a \leq b$ if all entries of a are at most the corresponding entry of b. For example, we have

```
sage: a = T((1,1))
sage: b = T((2,1))
sage: c = T((1,2))
sage: a <= b, a <= c, b <= c
(True, True, False)
```

```
sage: from sage.all import *
sage: a = T((Integer(1),Integer(1)))
sage: b = T((Integer(2),Integer(1)))
sage: c = T((Integer(1),Integer(2)))
sage: a <= b, a <= c, b <= c
(True, True, False)
```

The last comparison gives `False`, since the comparison of the first component checks whether $2 \leq 1$.

Now, let us add such elements to a poset:

```
sage: Q = MP([T((1, 1)), T((3, 3)), T((4, 1)),
....:         T((3, 2)), T((2, 3)), T((2, 2))]); Q
poset((1, 1), (2, 2), (2, 3), (3, 2), (3, 3), (4, 1))
```

```
>>> from sage.all import *
>>> Q = MP([T((Integer(1), Integer(1))), T((Integer(3), Integer(3))), T((Integer(4),␣
↪Integer(1))),
...          T((Integer(3), Integer(2))), T((Integer(2), Integer(3))), T((Integer(2),␣
↪Integer(2)))]); Q
poset((1, 1), (2, 2), (2, 3), (3, 2), (3, 3), (4, 1))
```

In the representation above, the elements are sorted topologically, smallest first. This does not (directly) show more structural information. We can overcome this and display a "wiring layout" by typing:

```
sage: print(Q.repr_full(reverse=True))
poset((3, 3), (2, 3), (3, 2), (2, 2), (4, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors:   (3, 3), (4, 1)
+-- (3, 3)
|   +-- successors:   oo
|   +-- predecessors:   (2, 3), (3, 2)
+-- (2, 3)
|   +-- successors:   (3, 3)
|   +-- predecessors:   (2, 2)
+-- (3, 2)
|   +-- successors:   (3, 3)
|   +-- predecessors:   (2, 2)
+-- (2, 2)
|   +-- successors:   (2, 3), (3, 2)
|   +-- predecessors:   (1, 1)
+-- (4, 1)
|   +-- successors:   oo
|   +-- predecessors:   (1, 1)
+-- (1, 1)
|   +-- successors:   (2, 2), (4, 1)
|   +-- predecessors:   null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
```

```
>>> from sage.all import *
>>> print(Q.repr_full(reverse=True))
poset((3, 3), (2, 3), (3, 2), (2, 2), (4, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors:   (3, 3), (4, 1)
+-- (3, 3)
|   +-- successors:   oo
|   +-- predecessors:   (2, 3), (3, 2)
+-- (2, 3)
|   +-- successors:   (3, 3)
|   +-- predecessors:   (2, 2)
+-- (3, 2)
|   +-- successors:   (3, 3)
|   +-- predecessors:   (2, 2)
+-- (2, 2)
```

(continues on next page)

(continued from previous page)

```
|   +-- successors:   (2, 3), (3, 2)
|   +-- predecessors:   (1, 1)
+-- (4, 1)
|   +-- successors:   oo
|   +-- predecessors:   (1, 1)
+-- (1, 1)
|   +-- successors:   (2, 2), (4, 1)
|   +-- predecessors:   null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
```

Note that we use `reverse=True` to let the elements appear from largest (on the top) to smallest (on the bottom).

If you look at the output above, you'll see two additional elements, namely `oo` (∞) and `null` ($\emptyset$). So what are these strange animals? The answer is simple and maybe you can guess it already. The ∞-element is larger than every other element, therefore a successor of the maximal elements in the poset. Similarly, the $\emptyset$-element is smaller than any other element, therefore a predecessor of the poset's minimal elements. Both do not have to scare us; they are just there and sometimes useful.

AUTHORS:

- Daniel Krenn (2015)

ACKNOWLEDGEMENT:

- Daniel Krenn is supported by the Austrian Science Fund (FWF): P 24644-N26.

5.2 Classes and their Methods

class sage.data_structures.mutable_poset.**MutablePoset**(*data=None*, *key=None*, *merge=None*, *can_merge=None*)

Bases: `SageObject`

A data structure that models a mutable poset (partially ordered set).

INPUT:

- `data` – data from which to construct the poset. It can be any of the following:
  1. `None` (default), in which case an empty poset is created,
  2. a *`MutablePoset`*, which will be copied during creation,
  3. an iterable, whose elements will be in the poset.
- `key` – a function which maps elements to keys. If `None` (default), this is the identity, i.e., keys are equal to their elements.

  Two elements with the same keys are considered as equal; so only one of these two elements can be in the poset.

  This `key` is not used for sorting (in contrast to sorting-functions, e.g. `sorted`).
- `merge` – a function which merges its second argument (an element) to its first (again an element) and returns the result (as an element). If the return value is `None`, the element is removed from the poset.

  This hook is called by *`merge()`*. Moreover it is used during *`add()`* when an element (more precisely its key) is already in this poset.

`merge` is `None` (default) is equivalent to `merge` returning its first argument. Note that it is not allowed that the key of the returning element differs from the key of the first input parameter. This means `merge` must not change the position of the element in the poset.

- `can_merge` – a function which checks whether its second argument can be merged to its first

  This hook is called by *merge()*. Moreover it is used during *add()* when an element (more precisely its key) is already in this poset.

  `can_merge` is `None` (default) is equivalent to `can_merge` returning `True` in all cases.

OUTPUT: a mutable poset

You can find a short introduction and examples *here*.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
```

We illustrate the different input formats

1. No input:

```
sage: A = MP(); A
poset()
```

```
>>> from sage.all import *
>>> A = MP(); A
poset()
```

2. A *MutablePoset*:

```
sage: B = MP(A); B
poset()
sage: B.add(42)
sage: C = MP(B); C
poset(42)
```

```
>>> from sage.all import *
>>> B = MP(A); B
poset()
>>> B.add(Integer(42))
>>> C = MP(B); C
poset(42)
```

3. An iterable:

```
sage: C = MP([5, 3, 11]); C
poset(3, 5, 11)
```

```
>>> from sage.all import *
>>> C = MP([Integer(5), Integer(3), Integer(11)]); C
poset(3, 5, 11)
```

See also

MutablePosetShell.

add(*element*)

Add the given object as element to the poset.

INPUT:

- `element` – an object (hashable and supporting comparison with the operator <=)

OUTPUT: nothing

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2))])
sage: print(P.repr_full(reverse=True))
poset((4, 4), (1, 3), (1, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 1)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2)
+-- (1, 2)
|   +-- successors:   (1, 3)
|   +-- predecessors: (1, 1)
+-- (2, 1)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 2), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
sage: P.add(T((2, 2)))
sage: reprP = P.repr_full(reverse=True); print(reprP)
poset((4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 2)
+-- (1, 3)
```

(continues on next page)

(continued from previous page)

```
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2)
+-- (2, 2)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2), (2, 1)
+-- (1, 2)
|   +-- successors:   (1, 3), (2, 2)
|   +-- predecessors: (1, 1)
+-- (2, 1)
|   +-- successors:   (2, 2)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 2), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2)))])
>>> print(P.repr_full(reverse=True))
poset((4, 4), (1, 3), (1, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 1)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2)
+-- (1, 2)
|   +-- successors:   (1, 3)
|   +-- predecessors: (1, 1)
+-- (2, 1)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 2), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
>>> P.add(T((Integer(2), Integer(2))))
>>> reprP = P.repr_full(reverse=True); print(reprP)
poset((4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1))
```

(continues on next page)

(continued from previous page)

```
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 2)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2)
+-- (2, 2)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2), (2, 1)
+-- (1, 2)
|   +-- successors:   (1, 3), (2, 2)
|   +-- predecessors: (1, 1)
+-- (2, 1)
|   +-- successors:   (2, 2)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 2), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
```

When adding an element which is already in the poset, nothing happens:

```
sage: e = T((2, 2))
sage: P.add(e)
sage: P.repr_full(reverse=True) == reprP
True
```

```
>>> from sage.all import *
>>> e = T((Integer(2), Integer(2)))
>>> P.add(e)
>>> P.repr_full(reverse=True) == reprP
True
```

We can influence the behavior when an element with existing key is to be inserted in the poset. For example, we can perform an addition on some argument of the elements:

```
sage: def add(left, right):
....:     return (left[0], ''.join(sorted(left[1] + right[1])))
sage: A = MP(key=lambda k: k[0], merge=add)
sage: A.add((3, 'a'))
sage: A
poset((3, 'a'))
sage: A.add((3, 'b'))
sage: A
poset((3, 'ab'))
```

```
>>> from sage.all import *
>>> def add(left, right):
...     return (left[Integer(0)], ''.join(sorted(left[Integer(1)] +␣
↪right[Integer(1)])))
>>> A = MP(key=lambda k: k[Integer(0)], merge=add)
>>> A.add((Integer(3), 'a'))
>>> A
poset((3, 'a'))
>>> A.add((Integer(3), 'b'))
>>> A
poset((3, 'ab'))
```

We can also deal with cancellations. If the return value of our hook-function is `None`, then the element is removed out of the poset:

```
sage: def add_None(left, right):
....:     s = left[1] + right[1]
....:     if s == 0:
....:         return None
....:     return (left[0], s)
sage: B = MP(key=lambda k: k[0],
....:        merge=add_None)
sage: B.add((7, 42))
sage: B.add((7, -42))
sage: B
poset()
```

```
>>> from sage.all import *
>>> def add_None(left, right):
...     s = left[Integer(1)] + right[Integer(1)]
...     if s == Integer(0):
...         return None
...     return (left[Integer(0)], s)
>>> B = MP(key=lambda k: k[Integer(0)],
...        merge=add_None)
>>> B.add((Integer(7), Integer(42)))
>>> B.add((Integer(7), -Integer(42)))
>>> B
poset()
```

See also

discard(), *pop()*, *remove()*.

clear()

Remove all elements from this poset.

OUTPUT: nothing

See also

discard(), *pop()*, *remove()*.

contains(*key*)

Test whether `key` is encapsulated by one of the poset's elements.

INPUT:

- `key` – an object

OUTPUT: boolean

See also

shells(), *elements()*, *keys()*.

copy(*mapping=None*)

Create a shallow copy.

INPUT:

- `mapping` – a function which is applied on each of the elements

OUTPUT: a poset with the same content as `self`

See also

map(), *mapped()*.

difference(**other*)

Return a new poset where all elements of this poset, which are contained in one of the other given posets, are removed.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

Note

The key of an element is used for comparison. Thus elements with the same key are considered as equal.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.difference(Q)
poset(3, 7)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.difference(Q)
poset(3, 7)
```

See also

union(), *union_update()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

difference_update(**other*)

Remove all elements of another poset from this poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT: nothing

Note

The key of an element is used for comparison. Thus elements with the same key are considered as equal.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.difference_update(Q)
sage: P
poset(3, 7)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.difference_update(Q)
>>> P
poset(3, 7)
```

See also

union(), *union_update()*, *difference()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

discard(*key*, *raise_key_error=False*)

Remove the given object from the poset.

INPUT:

- `key` – the key of an object
- `raise_key_error` – boolean (default: `False`); switch raising `KeyError` on and off

OUTPUT: nothing

If the element is not a member and `raise_key_error` is set (not default), raise a `KeyError`.

Note

As with Python's `set`, the methods *remove()* and *discard()* only differ in their behavior when an element is not contained in the poset: *remove()* raises a `KeyError` whereas *discard()* does not raise any exception.

This default behavior can be overridden with the `raise_key_error` parameter.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: P.discard(T((1, 2)))
sage: P.remove(T((1, 2)))
Traceback (most recent call last):
...
KeyError: 'Key (1, 2) is not contained in this poset.'
sage: P.discard(T((1, 2)))
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> P.discard(T((Integer(1), Integer(2))))
>>> P.remove(T((Integer(1), Integer(2))))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
KeyError: 'Key (1, 2) is not contained in this poset.'
>>> P.discard(T((Integer(1), Integer(2))))
```

See also

add(), *clear()*, *remove()*, *pop()*.

element(*key*)

Return the element corresponding to `key`.

INPUT:

- `key` – the key of an object

OUTPUT: an object

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(42)
sage: e = P.element(42); e
42
sage: type(e)
<class 'sage.rings.integer.Integer'>
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP()
>>> P.add(Integer(42))
>>> e = P.element(Integer(42)); e
42
>>> type(e)
<class 'sage.rings.integer.Integer'>
```

See also

shell(), *get_key()*.

elements(***kwargs*)

Return an iterator over all elements.

INPUT:

- `kwargs` – arguments are passed to *shells()*

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7])
sage: [(v, type(v)) for v in sorted(P.elements())]
```

(continues on next page)

(continued from previous page)

```
[(3, <class 'sage.rings.integer.Integer'>),
 (7, <class 'sage.rings.integer.Integer'>),
 (42, <class 'sage.rings.integer.Integer'>)]
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)])
>>> [(v, type(v)) for v in sorted(P.elements())]
[(3, <class 'sage.rings.integer.Integer'>),
 (7, <class 'sage.rings.integer.Integer'>),
 (42, <class 'sage.rings.integer.Integer'>)]
```

Note that

```
sage: it = iter(P)
sage: sorted(it)
[3, 7, 42]
```

```
>>> from sage.all import *
>>> it = iter(P)
>>> sorted(it)
[3, 7, 42]
```

returns all elements as well.

See also

shells(), *shells_topological()*, *elements_topological()*, *keys()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

elements_topological(***kwargs*)

Return an iterator over all elements in topological order.

INPUT:

- `kwargs` – arguments are passed to *shells_topological()*

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: [(v, type(v)) for v in P.elements_topological(key=repr)]
[((1, 1), <class '__main__.T'>),
 ((1, 2), <class '__main__.T'>),
 ((1, 3), <class '__main__.T'>),
 ((2, 1), <class '__main__.T'>),
 ((2, 2), <class '__main__.T'>),
 ((4, 4), <class '__main__.T'>)]
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> [(v, type(v)) for v in P.elements_topological(key=repr)]
[((1, 1), <class '__main__.T'>),
 ((1, 2), <class '__main__.T'>),
 ((1, 3), <class '__main__.T'>),
 ((2, 1), <class '__main__.T'>),
 ((2, 2), <class '__main__.T'>),
 ((4, 4), <class '__main__.T'>)]
```

See also

shells(), *shells_topological()*, *elements()*, *keys()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

get_key(*element*)

Return the key corresponding to the given element.

INPUT:

- `element` – an object

OUTPUT: an object (the key of `element`)

See also

element(), *shell()*.

intersection(**other*)

Return the intersection of the given posets as a new poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

Note

The key of an element is used for comparison. Thus elements with the same key are considered as equal.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
```

(continues on next page)

(continued from previous page)

```
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.intersection(Q)
poset(42)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.intersection(Q)
poset(42)
```

See also

union(), *union_update()*, *difference()*, *difference_update()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

intersection_update(**other*)

Update this poset with the intersection of itself and another poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT: nothing

Note

The key of an element is used for comparison. Thus elements with the same key are considered as equal; `A.intersection_update(B)` and `B.intersection_update(A)` might result in different posets.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.intersection_update(Q)
sage: P
poset(42)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
```

(continues on next page)

(continued from previous page)

```
poset(4, 8, 42)
>>> P.intersection_update(Q)
>>> P
poset(42)
```

See also

union(), *union_update()*, *difference()*, *difference_update()*, *intersection()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

is_disjoint(*other*)

Return whether another poset is disjoint to this poset.

INPUT:

- `other` – a poset or an iterable; in the latter case the iterated objects are seen as elements of a poset

OUTPUT: nothing

Note

If this poset uses a `key`-function, then all comparisons are performed on the keys of the elements (and not on the elements themselves).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.is_disjoint(Q)
False
sage: P.is_disjoint(Q.difference(P))
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.is_disjoint(Q)
False
>>> P.is_disjoint(Q.difference(P))
True
```

See also

is_subset(), *is_superset()*, *union()*, *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*.

is_subset(*other*)

Return whether another poset contains this poset, i.e., whether this poset is a subset of the other poset.

INPUT:

- `other` – a poset or an iterable; in the latter case the iterated objects are seen as elements of a poset

OUTPUT: nothing

Note

If this poset uses a `key`-function, then all comparisons are performed on the keys of the elements (and not on the elements themselves).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.is_subset(Q)
False
sage: Q.is_subset(P)
False
sage: P.is_subset(P)
True
sage: P.is_subset(P.union(Q))
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.is_subset(Q)
False
>>> Q.is_subset(P)
False
>>> P.is_subset(P)
True
>>> P.is_subset(P.union(Q))
True
```

See also

is_disjoint(), *is_superset()*, *union()*, *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*.

is_superset(*other*)

Return whether this poset contains another poset, i.e., whether this poset is a superset of the other poset.

INPUT:

- `other` – a poset or an iterable; in the latter case the iterated objects are seen as elements of a poset

OUTPUT: nothing

Note

If this poset uses a `key`-function, then all comparisons are performed on the keys of the elements (and not on the elements themselves).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.is_superset(Q)
False
sage: Q.is_superset(P)
False
sage: P.is_superset(P)
True
sage: P.union(Q).is_superset(P)
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.is_superset(Q)
False
>>> Q.is_superset(P)
False
>>> P.is_superset(P)
True
>>> P.union(Q).is_superset(P)
True
```

See also

is_disjoint(), *is_subset()*, *union()*, *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*.

isdisjoint(*other*)

Alias of *is_disjoint()*.

issubset(*other*)

Alias of *is_subset()*.

issuperset(*other*)

Alias of *is_superset()*.

keys(***kwargs*)

Return an iterator over all keys of the elements.

INPUT:

- `kwargs` – arguments are passed to *shells()*

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7], key=lambda c: -c)
sage: [(v, type(v)) for v in sorted(P.keys())]
[(-42, <class 'sage.rings.integer.Integer'>),
 (-7, <class 'sage.rings.integer.Integer'>),
 (-3, <class 'sage.rings.integer.Integer'>)]

sage: [(v, type(v)) for v in sorted(P.elements())]
[(3, <class 'sage.rings.integer.Integer'>),
 (7, <class 'sage.rings.integer.Integer'>),
 (42, <class 'sage.rings.integer.Integer'>)]

sage: [(v, type(v)) for v in sorted(P.shells(),
....:                               key=lambda c: c.element)]
[(3, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 (7, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 (42, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>)]
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)], key=lambda c: -c)
>>> [(v, type(v)) for v in sorted(P.keys())]
[(-42, <class 'sage.rings.integer.Integer'>),
 (-7, <class 'sage.rings.integer.Integer'>),
 (-3, <class 'sage.rings.integer.Integer'>)]

>>> [(v, type(v)) for v in sorted(P.elements())]
[(3, <class 'sage.rings.integer.Integer'>),
 (7, <class 'sage.rings.integer.Integer'>),
 (42, <class 'sage.rings.integer.Integer'>)]

>>> [(v, type(v)) for v in sorted(P.shells(),
```

(continues on next page)

(continued from previous page)

```
...                                 key=lambda c: c.element)]
[(3, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 (7, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 (42, <class 'sage.data_structures.mutable_poset.MutablePosetShell'>)]
```

See also

shells(), *shells_topological()*, *elements()*, *elements_topological()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

keys_topological(***kwargs*)

Return an iterator over all keys of the elements in topological order.

INPUT:

- `kwargs` – arguments are passed to *shells_topological()*

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([(1, 1), (2, 1), (4, 4)],
....:        key=lambda c: c[0])
sage: [(v, type(v)) for v in P.keys_topological(key=repr)]
[(1, <class 'sage.rings.integer.Integer'>),
 (2, <class 'sage.rings.integer.Integer'>),
 (4, <class 'sage.rings.integer.Integer'>)]
sage: [(v, type(v)) for v in P.elements_topological(key=repr)]
[((1, 1), <... 'tuple'>),
 ((2, 1), <... 'tuple'>),
 ((4, 4), <... 'tuple'>)]
sage: [(v, type(v)) for v in P.shells_topological(key=repr)]
[((1, 1), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 ((2, 1), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
 ((4, 4), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>)]
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([(Integer(1), Integer(1)), (Integer(2), Integer(1)), (Integer(4),␣
↪Integer(4))],
...        key=lambda c: c[Integer(0)])
>>> [(v, type(v)) for v in P.keys_topological(key=repr)]
[(1, <class 'sage.rings.integer.Integer'>),
 (2, <class 'sage.rings.integer.Integer'>),
 (4, <class 'sage.rings.integer.Integer'>)]
>>> [(v, type(v)) for v in P.elements_topological(key=repr)]
[((1, 1), <... 'tuple'>),
 ((2, 1), <... 'tuple'>),
 ((4, 4), <... 'tuple'>)]
>>> [(v, type(v)) for v in P.shells_topological(key=repr)]
[((1, 1), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
```

(continues on next page)

(continued from previous page)

```
((2, 1), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>),
((4, 4), <class 'sage.data_structures.mutable_poset.MutablePosetShell'>)]
```

See also

shells(), *shells_topological()*, *elements()*, *elements_topological()*, *keys()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

map(*function*, *topological=False*, *reverse=False*)

Apply the given `function` to each element of this poset.

INPUT:

- `function` – a function mapping an existing element to a new element
- `topological` – boolean (default: `False`); if set, then the mapping is done in topological order, otherwise unordered
- `reverse` – is passed on to topological ordering

OUTPUT: nothing

Note

Since this method works inplace, it is not allowed that `function` alters the key of an element.

Note

If `function` returns `None`, then the element is removed.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))],
....:        key=lambda e: e[:2])
sage: P.map(lambda e: e + (sum(e),))
sage: P
poset((1, 2, 3), (1, 3, 4), (2, 1, 3), (2, 2, 4), (4, 4, 8))
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(3))), T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
```

(continues on next page)

(continued from previous page)

```
↪T((Integer(2), Integer(2)))],
...        key=lambda e: e[:Integer(2)])
>>> P.map(lambda e: e + (sum(e),))
>>> P
poset((1, 2, 3), (1, 3, 4), (2, 1, 3), (2, 2, 4), (4, 4, 8))
```

See also

copy(), *mapped()*.

mapped (*function*)

Return a poset where on each element the given `function` was applied.

INPUT:

- `function` – a function mapping an existing element to a new element
- `topological` – boolean (default: `False`); if set, then the mapping is done in topological order, otherwise unordered
- `reverse` – is passed on to topological ordering

OUTPUT: a *MutablePoset*

Note

`function` is not allowed to change the order of the keys, but changing the keys themselves is allowed (in contrast to *map()*).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: P.mapped(lambda e: str(e))
poset('(1, 2)', '(1, 3)', '(2, 1)', '(2, 2)', '(4, 4)')
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(3))), T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> P.mapped(lambda e: str(e))
poset('(1, 2)', '(1, 3)', '(2, 1)', '(2, 2)', '(4, 4)')
```

See also

copy(), *map()*.

maximal_elements()

Return an iterator over the maximal elements of this poset.

OUTPUT: an iterator

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((1, 2)), T((2, 2))])
sage: sorted(P.maximal_elements())
[(1, 3), (2, 2)]
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(1), Integer(2))), T((Integer(2), Integer(2)))])
>>> sorted(P.maximal_elements())
[(1, 3), (2, 2)]
```

See also

minimal_elements()

merge(*key=None*, *reverse=False*)

Merge the given element with its successors/predecessors.

INPUT:

- `key` – the key specifying an element or `None` (default), in which case this method is called on each element in this poset
- `reverse` – boolean (default: `False`); specifies which direction to go first: `False` searches towards `'oo'` and `True` searches towards `'null'`. When `key=None`, then this also specifies which elements are merged first.

OUTPUT: nothing

This method tests all (not necessarily direct) successors and predecessors of the given element whether they can be merged with the element itself. This is done by the `can_merge`-function of *MutablePoset*. If this merge is possible, then it is performed by calling *MutablePoset*'s `merge`-function and the corresponding successor/predecessor is removed from the poset.

Note

`can_merge` is applied in the sense of the condition of depth first iteration, i.e., once `can_merge` fails, the successors/predecessors are no longer tested.

Note

The motivation for such a merge behavior comes from asymptotic expansions: $O(n^3)$ merges with, for example, $3n^2$ or $O(n)$ to $O(n^3)$ (as n tends to ∞; see Wikipedia article Big_O_notation).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: key = lambda t: T(t[0:2])
sage: def add(left, right):
....:     return (left[0], left[1],
....:             ''.join(sorted(left[2] + right[2])))
sage: def can_add(left, right):
....:     return key(left) >= key(right)
sage: P = MP([(1, 1, 'a'), (1, 3, 'b'), (2, 1, 'c'),
....:         (4, 4, 'd'), (1, 2, 'e'), (2, 2, 'f')],
....:        key=key, merge=add, can_merge=can_add)
sage: Q = copy(P)
sage: Q.merge(T((1, 3)))
sage: print(Q.repr_full(reverse=True))
poset((4, 4, 'd'), (1, 3, 'abe'), (2, 2, 'f'), (2, 1, 'c'))
+-- oo
|   +-- no successors
|   +-- predecessors:   (4, 4, 'd')
+-- (4, 4, 'd')
|   +-- successors:   oo
|   +-- predecessors:   (1, 3, 'abe'), (2, 2, 'f')
+-- (1, 3, 'abe')
|   +-- successors:   (4, 4, 'd')
|   +-- predecessors:   null
+-- (2, 2, 'f')
|   +-- successors:   (4, 4, 'd')
|   +-- predecessors:   (2, 1, 'c')
+-- (2, 1, 'c')
|   +-- successors:   (2, 2, 'f')
|   +-- predecessors:   null
+-- null
|   +-- successors:   (1, 3, 'abe'), (2, 1, 'c')
|   +-- no predecessors
sage: for k in sorted(P.keys()):
....:     Q = copy(P)
....:     Q.merge(k)
....:     print('merging %s: %s' % (k, Q))
```

(continues on next page)

(continued from previous page)

```
merging (1, 1): poset((1, 1, 'a'), (1, 2, 'e'), (1, 3, 'b'),
                      (2, 1, 'c'), (2, 2, 'f'), (4, 4, 'd'))
merging (1, 2): poset((1, 2, 'ae'), (1, 3, 'b'), (2, 1, 'c'),
                      (2, 2, 'f'), (4, 4, 'd'))
merging (1, 3): poset((1, 3, 'abe'), (2, 1, 'c'), (2, 2, 'f'),
                      (4, 4, 'd'))
merging (2, 1): poset((1, 2, 'e'), (1, 3, 'b'), (2, 1, 'ac'),
                      (2, 2, 'f'), (4, 4, 'd'))
merging (2, 2): poset((1, 3, 'b'), (2, 2, 'acef'), (4, 4, 'd'))
merging (4, 4): poset((4, 4, 'abcdef'))

sage: Q = copy(P)
sage: Q.merge(); Q
poset((4, 4, 'abcdef'))
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> key = lambda t: T(t[Integer(0):Integer(2)])
>>> def add(left, right):
...     return (left[Integer(0)], left[Integer(1)],
...             ''.join(sorted(left[Integer(2)] + right[Integer(2)])))
>>> def can_add(left, right):
...     return key(left) >= key(right)
>>> P = MP([(Integer(1), Integer(1), 'a'), (Integer(1), Integer(3), 'b'),␣
↪(Integer(2), Integer(1), 'c'),
...          (Integer(4), Integer(4), 'd'), (Integer(1), Integer(2), 'e'),␣
↪(Integer(2), Integer(2), 'f')],
...        key=key, merge=add, can_merge=can_add)
>>> Q = copy(P)
>>> Q.merge(T((Integer(1), Integer(3))))
>>> print(Q.repr_full(reverse=True))
poset((4, 4, 'd'), (1, 3, 'abe'), (2, 2, 'f'), (2, 1, 'c'))
+-- oo
|   +-- no successors
|   +-- predecessors:   (4, 4, 'd')
+-- (4, 4, 'd')
|   +-- successors:   oo
|   +-- predecessors:   (1, 3, 'abe'), (2, 2, 'f')
+-- (1, 3, 'abe')
|   +-- successors:   (4, 4, 'd')
|   +-- predecessors:   null
+-- (2, 2, 'f')
|   +-- successors:   (4, 4, 'd')
|   +-- predecessors:   (2, 1, 'c')
+-- (2, 1, 'c')
|   +-- successors:   (2, 2, 'f')
|   +-- predecessors:   null
+-- null
|   +-- successors:   (1, 3, 'abe'), (2, 1, 'c')
```

(continues on next page)

(continued from previous page)

```
|   +-- no predecessors
>>> for k in sorted(P.keys()):
...     Q = copy(P)
...     Q.merge(k)
...     print('merging %s: %s' % (k, Q))
merging (1, 1): poset((1, 1, 'a'), (1, 2, 'e'), (1, 3, 'b'),
                      (2, 1, 'c'), (2, 2, 'f'), (4, 4, 'd'))
merging (1, 2): poset((1, 2, 'ae'), (1, 3, 'b'), (2, 1, 'c'),
                      (2, 2, 'f'), (4, 4, 'd'))
merging (1, 3): poset((1, 3, 'abe'), (2, 1, 'c'), (2, 2, 'f'),
                      (4, 4, 'd'))
merging (2, 1): poset((1, 2, 'e'), (1, 3, 'b'), (2, 1, 'ac'),
                      (2, 2, 'f'), (4, 4, 'd'))
merging (2, 2): poset((1, 3, 'b'), (2, 2, 'acef'), (4, 4, 'd'))
merging (4, 4): poset((4, 4, 'abcdef'))

>>> Q = copy(P)
>>> Q.merge(); Q
poset((4, 4, 'abcdef'))
```

See also

MutablePosetShell.merge()

minimal_elements()

Return an iterator over the minimal elements of this poset.

OUTPUT: an iterator

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: sorted(P.minimal_elements())
[(1, 2), (2, 1)]
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(3))), T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> sorted(P.minimal_elements())
[(1, 2), (2, 1)]
```

See also

maximal_elements()

property null

The shell $\emptyset$ whose element is smaller than any other element.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: z = P.null; z
null
sage: z.is_null()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP()
>>> z = P.null; z
null
>>> z.is_null()
True
```

See also

oo(), *MutablePosetShell.is_null()*, *MutablePosetShell.is_special()*.

property oo

The shell ∞ whose element is larger than any other element.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: oo = P.oo; oo
oo
sage: oo.is_oo()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP()
>>> oo = P.oo; oo
oo
>>> oo.is_oo()
True
```

See also

null(), *MutablePosetShell.is_oo()*, *MutablePosetShell.is_special()*.

pop(***kwargs*)

Remove and return an arbitrary poset element.

INPUT:

- `kwargs` – arguments are passed to *shells_topological()*

OUTPUT: an object

Note

The special elements `'null'` and `'oo'` cannot be popped.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(3)
sage: P
poset(3)
sage: P.pop()
3
sage: P
poset()
sage: P.pop()
Traceback (most recent call last):
...
KeyError: 'pop from an empty poset'
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP()
>>> P.add(Integer(3))
>>> P
poset(3)
>>> P.pop()
3
>>> P
poset()
>>> P.pop()
Traceback (most recent call last):
...
KeyError: 'pop from an empty poset'
```

See also

add(), *clear()*, *discard()*, *remove()*.

remove(*key*, *raise_key_error=True*)

Remove the given object from the poset.

INPUT:

- `key` – the key of an object

- raise_key_error – boolean (default: True); switch raising KeyError on and off

OUTPUT: nothing

If the element is not a member and raise_key_error is set (default), raise a KeyError.

Note

As with Python's set, the methods *remove()* and *discard()* only differ in their behavior when an element is not contained in the poset: *remove()* raises a KeyError whereas *discard()* does not raise any exception.

This default behavior can be overridden with the raise_key_error parameter.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: print(P.repr_full(reverse=True))
poset((4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 2)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2)
+-- (2, 2)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2), (2, 1)
+-- (1, 2)
|   +-- successors:   (1, 3), (2, 2)
|   +-- predecessors: (1, 1)
+-- (2, 1)
|   +-- successors:   (2, 2)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 2), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
sage: P.remove(T((1, 2)))
sage: print(P.repr_full(reverse=True))
poset((4, 4), (1, 3), (2, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
```

(continues on next page)

(continued from previous page)

```
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 2)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 1)
+-- (2, 2)
|   +-- successors:   (4, 4)
|   +-- predecessors: (2, 1)
+-- (2, 1)
|   +-- successors:   (2, 2)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 3), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> print(P.repr_full(reverse=True))
poset((4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 2)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2)
+-- (2, 2)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 2), (2, 1)
+-- (1, 2)
|   +-- successors:   (1, 3), (2, 2)
|   +-- predecessors: (1, 1)
+-- (2, 1)
|   +-- successors:   (2, 2)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 2), (2, 1)
|   +-- predecessors: null
+-- null
```

(continues on next page)

(continued from previous page)

```
|   +-- successors:   (1, 1)
|   +-- no predecessors
>>> P.remove(T((Integer(1), Integer(2))))
>>> print(P.repr_full(reverse=True))
poset((4, 4), (1, 3), (2, 2), (2, 1), (1, 1))
+-- oo
|   +-- no successors
|   +-- predecessors: (4, 4)
+-- (4, 4)
|   +-- successors:   oo
|   +-- predecessors: (1, 3), (2, 2)
+-- (1, 3)
|   +-- successors:   (4, 4)
|   +-- predecessors: (1, 1)
+-- (2, 2)
|   +-- successors:   (4, 4)
|   +-- predecessors: (2, 1)
+-- (2, 1)
|   +-- successors:   (2, 2)
|   +-- predecessors: (1, 1)
+-- (1, 1)
|   +-- successors:   (1, 3), (2, 1)
|   +-- predecessors: null
+-- null
|   +-- successors:   (1, 1)
|   +-- no predecessors
```

See also

add(), *clear()*, *discard()*, *pop()*.

repr(*include_special=False*, *reverse=False*)

Return a representation of the poset.

INPUT:

- `include_special` – boolean (default: `False`); whether to include the special elements `'null'` and `'oo'` or not
- `reverse` – boolean (default: `False`); if set, then largest elements are displayed first

OUTPUT: string

See also

repr_full()

repr_full(*reverse=False*)

Return a representation with ordering details of the poset.

INPUT:

- `reverse` – boolean (default: `False`); if set, then largest elements are displayed first

OUTPUT: string

See also

repr()

shell(*key*)

Return the shell of the element corresponding to `key`.

INPUT:

- `key` – the key of an object

OUTPUT: an instance of *MutablePosetShell*

Note

Each element is contained/encapsulated in a shell inside the poset.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(42)
sage: e = P.shell(42); e
42
sage: type(e)
<class 'sage.data_structures.mutable_poset.MutablePosetShell'>
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP()
>>> P.add(Integer(42))
>>> e = P.shell(Integer(42)); e
42
>>> type(e)
<class 'sage.data_structures.mutable_poset.MutablePosetShell'>
```

See also

element(), *get_key()*.

shells(*include_special=False*)

Return an iterator over all shells.

INPUT:

- `include_special` – boolean (default: `False`); if set, then including shells containing a smallest element ($\emptyset$) and a largest element (∞)

OUTPUT: an iterator

Note

Each element is contained/encapsulated in a shell inside the poset.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: tuple(P.shells())
()
sage: tuple(P.shells(include_special=True))
(null, oo)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP()
>>> tuple(P.shells())
()
>>> tuple(P.shells(include_special=True))
(null, oo)
```

See also

shells_topological(), *elements()*, *elements_topological()*, *keys()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

shells_topological(*include_special=False*, *reverse=False*, *key=None*)

Return an iterator over all shells in topological order.

INPUT:

- `include_special` – boolean (default: `False`); if set, then including shells containing a smallest element ($\emptyset$) and a largest element (∞).
- `reverse` – boolean (default: `False`); if set, reverses the order, i.e., `False` gives smallest elements first, `True` gives largest first.
- `key` – (default: `None`) a function used for sorting the direct successors of a shell (used in case of a tie). If this is `None`, no sorting occurs.

OUTPUT: an iterator

Note

Each element is contained/encapsulated in a shell inside the poset.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
```

(continues on next page)

(continued from previous page)

```
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: list(P.shells_topological(key=repr))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4)]
sage: list(P.shells_topological(reverse=True, key=repr))
[(4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1)]
sage: list(P.shells_topological(include_special=True, key=repr))
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4), oo]
sage: list(P.shells_topological(
....:     include_special=True, reverse=True, key=repr))
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1), null]
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> list(P.shells_topological(key=repr))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4)]
>>> list(P.shells_topological(reverse=True, key=repr))
[(4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1)]
>>> list(P.shells_topological(include_special=True, key=repr))
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4), oo]
>>> list(P.shells_topological(
...     include_special=True, reverse=True, key=repr))
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1), null]
```

See also

shells(), *elements()*, *elements_topological()*, *keys()*, *keys_topological()*, *MutablePosetShell.iter_depth_first()*, *MutablePosetShell.iter_topological()*.

symmetric_difference(*other*)

Return the symmetric difference of two posets as a new poset.

INPUT:

- `other` – a poset

Note

The key of an element is used for comparison. Thus elements with the same key are considered as equal.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.symmetric_difference(Q)
poset(3, 4, 7, 8)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.symmetric_difference(Q)
poset(3, 4, 7, 8)
```

See also

union(), *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

symmetric_difference_update(*other*)

Update this poset with the symmetric difference of itself and another poset.

INPUT:

- `other` – a poset

OUTPUT: nothing

Note

The key of an element is used for comparison. Thus elements with the same key are considered as equal; `A.symmetric_difference_update(B)` and `B.symmetric_difference_update(A)` might result in different posets.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.symmetric_difference_update(Q)
sage: P
poset(3, 4, 7, 8)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
```

(continues on next page)

(continued from previous page)

```
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.symmetric_difference_update(Q)
>>> P
poset(3, 4, 7, 8)
```

See also

union(), *union_update()*, *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

union(**other*)

Return the union of the given posets as a new poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

Note

The key of an element is used for comparison. Thus elements with the same key are considered as equal.

Due to keys and a `merge` function (see *MutablePoset*) this operation might not be commutative.

Todo

Use the already existing information in the other poset to speed up this function. (At the moment each element of the other poset is inserted one by one and without using this information.)

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.union(Q)
poset(3, 4, 7, 8, 42)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
```

(continues on next page)

(continued from previous page)

```
>>> P.union(Q)
poset(3, 4, 7, 8, 42)
```

See also

union_update(), *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

union_update(**other*)

Update this poset with the union of itself and another poset.

INPUT:

- `other` – a poset or an iterable. In the latter case the iterated objects are seen as elements of a poset. It is possible to specify more than one `other` as variadic arguments (arbitrary argument lists).

OUTPUT: nothing

Note

The key of an element is used for comparison. Thus elements with the same key are considered as equal; `A.union_update(B)` and `B.union_update(A)` might result in different posets.

Todo

Use the already existing information in the other poset to speed up this function. (At the moment each element of the other poset is inserted one by one and without using this information.)

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP([3, 42, 7]); P
poset(3, 7, 42)
sage: Q = MP([4, 8, 42]); Q
poset(4, 8, 42)
sage: P.union_update(Q)
sage: P
poset(3, 4, 7, 8, 42)
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP([Integer(3), Integer(42), Integer(7)]); P
poset(3, 7, 42)
>>> Q = MP([Integer(4), Integer(8), Integer(42)]); Q
poset(4, 8, 42)
>>> P.union_update(Q)
>>> P
poset(3, 4, 7, 8, 42)
```

See also

union(), *difference()*, *difference_update()*, *intersection()*, *intersection_update()*, *symmetric_difference()*, *symmetric_difference_update()*, *is_disjoint()*, *is_subset()*, *is_superset()*.

update(**other*)

Alias of *union_update()*.

class sage.data_structures.mutable_poset.**MutablePosetShell**(*poset*, *element*)

Bases: SageObject

A shell for an element of a *mutable poset*.

INPUT:

- `poset` – the poset to which this shell belongs
- `element` – the element which should be contained/encapsulated in this shell

OUTPUT: a shell for the given element

Note

If the *element()* of a shell is `None`, then this element is considered as "special" (see *is_special()*). There are two special elements, namely

- a `'null'` (an element smaller than each other element; it has no predecessors) and
- an `'oo'` (an element larger than each other element; it has no successors).

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: P = MP()
sage: P.add(66)
sage: P
poset(66)
sage: s = P.shell(66)
sage: type(s)
<class 'sage.data_structures.mutable_poset.MutablePosetShell'>
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> P = MP()
>>> P.add(Integer(66))
>>> P
poset(66)
>>> s = P.shell(Integer(66))
>>> type(s)
<class 'sage.data_structures.mutable_poset.MutablePosetShell'>
```

See also

MutablePoset

property element

The element contained in this shell.

See also

key(), *MutablePoset*.

eq(*other*)

Return whether this shell is equal to `other`.

INPUT:

- `other` – a shell

OUTPUT: boolean

Note

This method compares the keys of the elements contained in the (non-special) shells. In particular, elements/shells with the same key are considered as equal.

See also

le(), *MutablePoset*.

is_null()

Return whether this shell contains the null-element, i.e., the element smaller than any possible other element.

OUTPUT: boolean

See also

is_special(), *is_oo()*, *MutablePoset.null()*, *MutablePoset*.

is_oo()

Return whether this shell contains the infinity-element, i.e., the element larger than any possible other element.

OUTPUT: boolean

See also

is_null(), *is_special()*, *MutablePoset.oo()*, *MutablePoset*.

is_special()

Return whether this shell contains either the null-element, i.e., the element smaller than any possible other element or the infinity-element, i.e., the element larger than any possible other element.

OUTPUT: boolean

See also

is_null(), *is_oo()*, *MutablePoset*.

iter_depth_first(*reverse=False*, *key=None*, *condition=None*)

Iterate over all shells in depth first order.

INPUT:

- `reverse` – boolean (default: `False`); if set, reverses the order, i.e., `False` searches towards `'oo'` and `True` searches towards `'null'`
- `key` – (default: `None`) a function used for sorting the direct successors of a shell (used in case of a tie). If this is `None`, no sorting occurs.
- `condition` – (default: `None`) a function mapping a shell to `True` (include in iteration) or `False` (do not include). `None` is equivalent to a function returning always `True`. Note that the iteration does not go beyond a not included shell.

Note

The depth first search starts at this (`self`) shell. Thus only this shell and shells greater than (in case of `reverse=False`) this shell are visited.

ALGORITHM:

See Wikipedia article Depth-first_search.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: list(P.null.iter_depth_first(reverse=False, key=repr))
[null, (1, 1), (1, 2), (1, 3), (4, 4), oo, (2, 2), (2, 1)]
sage: list(P.oo.iter_depth_first(reverse=True, key=repr))
[oo, (4, 4), (1, 3), (1, 2), (1, 1), null, (2, 2), (2, 1)]
sage: list(P.null.iter_depth_first(
....:     condition=lambda s: s.element[0] == 1))
[null, (1, 1), (1, 2), (1, 3)]
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> list(P.null.iter_depth_first(reverse=False, key=repr))
```

(continues on next page)

(continued from previous page)

```
[null, (1, 1), (1, 2), (1, 3), (4, 4), oo, (2, 2), (2, 1)]
>>> list(P.oo.iter_depth_first(reverse=True, key=repr))
[oo, (4, 4), (1, 3), (1, 2), (1, 1), null, (2, 2), (2, 1)]
>>> list(P.null.iter_depth_first(
...     condition=lambda s: s.element[Integer(0)] == Integer(1)))
[null, (1, 1), (1, 2), (1, 3)]
```

See also

iter_topological(), *MutablePoset*.

iter_topological(*reverse=False*, *key=None*, *condition=None*)

Iterate over all shells in topological order.

INPUT:

- `reverse` – boolean (default: `False`); if set, reverses the order, i.e., `False` searches towards `'oo'` and `True` searches towards `'null'`
- `key` – (default: `None`) a function used for sorting the direct predecessors of a shell (used in case of a tie). If this is `None`, no sorting occurs.
- `condition` – (default: `None`) a function mapping a shell to `True` (include in iteration) or `False` (do not include). `None` is equivalent to a function returning always `True`. Note that the iteration does not go beyond a not included shell.

OUTPUT: an iterator

Note

The topological search will only find shells smaller than (in case of `reverse=False`) or equal to this (`self`) shell. This is in contrast to *iter_depth_first()*.

ALGORITHM:

Here a simplified version of the algorithm found in [Tar1976] and [CLRS2001] is used. See also Wikipedia article Topological_sorting.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
```

(continues on next page)

(continued from previous page)

```
↪T((Integer(2), Integer(1))),
...          T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
```

```
sage: for e in P.shells_topological(include_special=True,
....:                               reverse=True, key=repr):
....:     print(e)
....:     print(list(e.iter_topological(reverse=True, key=repr)))
oo
[oo]
(4, 4)
[oo, (4, 4)]
(1, 3)
[oo, (4, 4), (1, 3)]
(2, 2)
[oo, (4, 4), (2, 2)]
(1, 2)
[oo, (4, 4), (1, 3), (2, 2), (1, 2)]
(2, 1)
[oo, (4, 4), (2, 2), (2, 1)]
(1, 1)
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1)]
null
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1), null]
```

```
>>> from sage.all import *
>>> for e in P.shells_topological(include_special=True,
...                               reverse=True, key=repr):
...     print(e)
...     print(list(e.iter_topological(reverse=True, key=repr)))
oo
[oo]
(4, 4)
[oo, (4, 4)]
(1, 3)
[oo, (4, 4), (1, 3)]
(2, 2)
[oo, (4, 4), (2, 2)]
(1, 2)
[oo, (4, 4), (1, 3), (2, 2), (1, 2)]
(2, 1)
[oo, (4, 4), (2, 2), (2, 1)]
(1, 1)
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1)]
null
[oo, (4, 4), (1, 3), (2, 2), (1, 2), (2, 1), (1, 1), null]
```

```
sage: for e in P.shells_topological(include_special=True,
....:                               reverse=True, key=repr):
....:     print(e)
....:     print(list(e.iter_topological(reverse=False, key=repr)))
```

(continues on next page)

(continued from previous page)

```
oo
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4), oo]
(4, 4)
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4)]
(1, 3)
[null, (1, 1), (1, 2), (1, 3)]
(2, 2)
[null, (1, 1), (1, 2), (2, 1), (2, 2)]
(1, 2)
[null, (1, 1), (1, 2)]
(2, 1)
[null, (1, 1), (2, 1)]
(1, 1)
[null, (1, 1)]
null
[null]
```

```
>>> from sage.all import *
>>> for e in P.shells_topological(include_special=True,
...                               reverse=True, key=repr):
...     print(e)
...     print(list(e.iter_topological(reverse=False, key=repr)))
oo
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4), oo]
(4, 4)
[null, (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (4, 4)]
(1, 3)
[null, (1, 1), (1, 2), (1, 3)]
(2, 2)
[null, (1, 1), (1, 2), (2, 1), (2, 2)]
(1, 2)
[null, (1, 1), (1, 2)]
(2, 1)
[null, (1, 1), (2, 1)]
(1, 1)
[null, (1, 1)]
null
[null]
```

```
sage: list(P.null.iter_topological(
....:     reverse=True, condition=lambda s: s.element[0] == 1,
....:     key=repr))
[(1, 3), (1, 2), (1, 1), null]
```

```
>>> from sage.all import *
>>> list(P.null.iter_topological(
...     reverse=True, condition=lambda s: s.element[Integer(0)] == Integer(1),
...     key=repr))
[(1, 3), (1, 2), (1, 1), null]
```

See also

iter_depth_first(), *MutablePoset.shells_topological()*, *MutablePoset.elements_topological()*, *MutablePoset.keys_topological()*, *MutablePoset*.

property key

The key of the element contained in this shell.

The key of an element is determined by the mutable poset (the parent) via the `key`-function (see construction of a *MutablePoset*).

See also

element(), *MutablePoset*.

le(*other*, *reverse=False*)

Return whether this shell is less than or equal to `other`.

INPUT:

- `other` – a shell
- `reverse` – boolean (default: `False`); if set, then return whether this shell is greater than or equal to `other`

OUTPUT: boolean

Note

The comparison of the shells is based on the comparison of the keys of the elements contained in the shells, except for special shells (see *MutablePosetShell*).

See also

eq(), *MutablePoset*.

lower_covers(*shell*, *reverse=False*)

Return the lower covers of the specified `shell`; the search is started at this (`self`) shell.

A lower cover of x is an element y of the poset such that $y < x$ and there is no element z of the poset so that $y < z < x$.

INPUT:

- `shell` – the shell for which to find the covering shells There is no restriction of `shell` being contained in the poset If `shell` is contained in the poset, then use the more efficient methods *predecessors()* and *successors()*.
- `reverse` – boolean (default: `False`); if set, then find the upper covers (see also *upper_covers()*) instead of the lower covers

OUTPUT: a set of *shells*

Note

Suppose `reverse` is `False`. This method starts at the calling shell (`self`) and searches towards `'oo'`. Thus, only shells which are (not necessarily direct) successors of this shell are considered.

If `reverse` is `True`, then the reverse direction is taken.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: e = P.shell(T((2, 2))); e
(2, 2)
sage: sorted(P.null.lower_covers(e),
....:        key=lambda c: repr(c.element))
[(1, 2), (2, 1)]
sage: set(_) == e.predecessors()
True
sage: sorted(P.oo.upper_covers(e),
....:        key=lambda c: repr(c.element))
[(4, 4)]
sage: set(_) == e.successors()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> e = P.shell(T((Integer(2), Integer(2)))); e
(2, 2)
>>> sorted(P.null.lower_covers(e),
...        key=lambda c: repr(c.element))
[(1, 2), (2, 1)]
>>> set(_) == e.predecessors()
True
>>> sorted(P.oo.upper_covers(e),
...        key=lambda c: repr(c.element))
[(4, 4)]
>>> set(_) == e.successors()
True
```

```
sage: Q = MP([T((3, 2))])
sage: f = next(Q.shells())
```

(continues on next page)

(continued from previous page)

```
sage: sorted(P.null.lower_covers(f),
....:        key=lambda c: repr(c.element))
[(2, 2)]
sage: sorted(P.oo.upper_covers(f),
....:        key=lambda c: repr(c.element))
[(4, 4)]
```

```
>>> from sage.all import *
>>> Q = MP([T((Integer(3), Integer(2)))])
>>> f = next(Q.shells())
>>> sorted(P.null.lower_covers(f),
...        key=lambda c: repr(c.element))
[(2, 2)]
>>> sorted(P.oo.upper_covers(f),
...        key=lambda c: repr(c.element))
[(4, 4)]
```

See also

upper_covers(), *predecessors()*, *successors()*, *MutablePoset*.

merge(*element*, *check=True*, *delete=True*)

Merge the given element with the element contained in this shell.

INPUT:

- `element` – an element (of the poset)
- `check` – boolean (default: `True`); if set, then the `can_merge`-function of *MutablePoset* determines whether the merge is possible. `can_merge` is `None` means that this check is always passed.
- `delete` – boolean (default: `True`); if set, then `element` is removed from the poset after the merge

OUTPUT: nothing

Note

This operation depends on the parameters `merge` and `can_merge` of the *MutablePoset* this shell is contained in. These parameters are defined when the poset is constructed.

Note

If the `merge` function returns `None`, then this shell is removed from the poset.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: def add(left, right):
....:     return (left[0], ''.join(sorted(left[1] + right[1])))
sage: def can_add(left, right):
```

(continues on next page)

(continued from previous page)

```
....:     return left[0] <= right[0]
sage: P = MP([(1, 'a'), (3, 'b'), (2, 'c'), (4, 'd')],
....:        key=lambda c: c[0], merge=add, can_merge=can_add)
sage: P
poset((1, 'a'), (2, 'c'), (3, 'b'), (4, 'd'))
sage: P.shell(2).merge((3, 'b'))
sage: P
poset((1, 'a'), (2, 'bc'), (4, 'd'))
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> def add(left, right):
...     return (left[Integer(0)], ''.join(sorted(left[Integer(1)] +␣
↪right[Integer(1)])))
>>> def can_add(left, right):
...     return left[Integer(0)] <= right[Integer(0)]
>>> P = MP([(Integer(1), 'a'), (Integer(3), 'b'), (Integer(2), 'c'),␣
↪(Integer(4), 'd')],
...        key=lambda c: c[Integer(0)], merge=add, can_merge=can_add)
>>> P
poset((1, 'a'), (2, 'c'), (3, 'b'), (4, 'd'))
>>> P.shell(Integer(2)).merge((Integer(3), 'b'))
>>> P
poset((1, 'a'), (2, 'bc'), (4, 'd'))
```

See also

MutablePoset.merge(), *MutablePoset*.

property poset

The poset to which this shell belongs.

See also

MutablePoset

predecessors(*reverse=False*)

Return the predecessors of this shell.

INPUT:

- `reverse` – boolean (default: `False`); if set, then return successors instead

OUTPUT: set

See also

successors(), *MutablePoset*.

successors(*reverse=False*)

Return the successors of this shell.

INPUT:

- `reverse` – boolean (default: `False`); if set, then return predecessors instead

OUTPUT: set

See also

predecessors(), *MutablePoset*.

upper_covers(*shell*, *reverse=False*)

Return the upper covers of the specified `shell`; the search is started at this (`self`) shell.

An upper cover of x is an element y of the poset such that $x < y$ and there is no element z of the poset so that $x < z < y$.

INPUT:

- `shell` – the shell for which to find the covering shells There is no restriction of `shell` being contained in the poset If `shell` is contained in the poset, then use the more efficient methods *predecessors()* and *successors()*.
- `reverse` – boolean (default: `False`); if set, then find the lower covers (see also *lower_covers()*) instead of the upper covers.

OUTPUT: a set of *shells*

Note

Suppose `reverse` is `False`. This method starts at the calling shell (`self`) and searches towards `'null'`. Thus, only shells which are (not necessarily direct) predecessors of this shell are considered.

If `reverse` is `True`, then the reverse direction is taken.

EXAMPLES:

```
sage: from sage.data_structures.mutable_poset import MutablePoset as MP
sage: class T(tuple):
....:     def __le__(left, right):
....:         return all(l <= r for l, r in zip(left, right))
sage: P = MP([T((1, 1)), T((1, 3)), T((2, 1)),
....:         T((4, 4)), T((1, 2)), T((2, 2))])
sage: e = P.shell(T((2, 2))); e
(2, 2)
sage: sorted(P.null.lower_covers(e),
....:        key=lambda c: repr(c.element))
[(1, 2), (2, 1)]
sage: set(_) == e.predecessors()
True
sage: sorted(P.oo.upper_covers(e),
....:        key=lambda c: repr(c.element))
[(4, 4)]
sage: set(_) == e.successors()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.mutable_poset import MutablePoset as MP
>>> class T(tuple):
...     def __le__(left, right):
...         return all(l <= r for l, r in zip(left, right))
>>> P = MP([T((Integer(1), Integer(1))), T((Integer(1), Integer(3))),␣
↪T((Integer(2), Integer(1))),
...         T((Integer(4), Integer(4))), T((Integer(1), Integer(2))),␣
↪T((Integer(2), Integer(2)))])
>>> e = P.shell(T((Integer(2), Integer(2)))); e
(2, 2)
>>> sorted(P.null.lower_covers(e),
...        key=lambda c: repr(c.element))
[(1, 2), (2, 1)]
>>> set(_) == e.predecessors()
True
>>> sorted(P.oo.upper_covers(e),
...        key=lambda c: repr(c.element))
[(4, 4)]
>>> set(_) == e.successors()
True
```

```
sage: Q = MP([T((3, 2))])
sage: f = next(Q.shells())
sage: sorted(P.null.lower_covers(f),
....:        key=lambda c: repr(c.element))
[(2, 2)]
sage: sorted(P.oo.upper_covers(f),
....:        key=lambda c: repr(c.element))
[(4, 4)]
```

```
>>> from sage.all import *
>>> Q = MP([T((Integer(3), Integer(2)))])
>>> f = next(Q.shells())
>>> sorted(P.null.lower_covers(f),
...        key=lambda c: repr(c.element))
[(2, 2)]
>>> sorted(P.oo.upper_covers(f),
...        key=lambda c: repr(c.element))
[(4, 4)]
```

See also

predecessors(), *successors()*, *MutablePoset*.

sage.data_structures.mutable_poset.**is_MutablePoset**(*P*)

Test whether P inherits from *MutablePoset*.

See also

MutablePoset

CHAPTER

SIX

PAIRING HEAP

This module proposes several implementations of the pairing heap data structure [FSST1986]. See the Wikipedia article Pairing_heap for more information on this min-heap data structure.

- *PairingHeap_of_n_integers*: a pairing heap data structure with fixed capacity n. Its items are integers in the range $[0, n-1]$. Values can be of any type equipped with a comparison method (<=).
- *PairingHeap_of_n_hashables*: a pairing heap data structure with fixed capacity n. Its items can be of any hashable type. Values can be of any type equipped with a comparison method (<=).
- `PairingHeap`: interface to a pairing heap data structure written in C++. The advantages of this data structure are that: its capacity is unbounded; items can be of any hashable type equipped with a hashing method that can be supported by `std::unordered_map`; values can be of any specified type equipped with a comparison method (<=). This data structure is for internal use and therefore cannot be accessed from a shell.

EXAMPLES:

Pairing heap of n integers in the range $[0, n-1]$:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(10); P
PairingHeap_of_n_integers: capacity 10, size 0
sage: P.push(1, 3)
sage: P.push(2, 2)
sage: P
PairingHeap_of_n_integers: capacity 10, size 2
sage: P.top()
(2, 2)
sage: P.decrease(1, 1)
sage: P.top()
(1, 1)
sage: P.pop()
sage: P.top()
(2, 2)

sage: P = PairingHeap_of_n_integers(10)
sage: P.push(1, (2, 'a'))
sage: P.push(2, (2, 'b'))
sage: P.top()
(1, (2, 'a'))
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(10)); P
```

(continues on next page)

(continued from previous page)

```
PairingHeap_of_n_integers: capacity 10, size 0
>>> P.push(Integer(1), Integer(3))
>>> P.push(Integer(2), Integer(2))
>>> P
PairingHeap_of_n_integers: capacity 10, size 2
>>> P.top()
(2, 2)
>>> P.decrease(Integer(1), Integer(1))
>>> P.top()
(1, 1)
>>> P.pop()
>>> P.top()
(2, 2)

>>> P = PairingHeap_of_n_integers(Integer(10))
>>> P.push(Integer(1), (Integer(2), 'a'))
>>> P.push(Integer(2), (Integer(2), 'b'))
>>> P.top()
(1, (2, 'a'))
```

Pairing heap of n hashables:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(10); P
PairingHeap_of_n_hashables: capacity 10, size 0
sage: P.push(1, 3)
sage: P.push('b', 2)
sage: P.push((1, 'abc'), 4)
sage: P.top()
('b', 2)
sage: P.decrease((1, 'abc'), 1)
sage: P.top()
((1, 'abc'), 1)
sage: P.pop()
sage: P.top()
('b', 2)

sage: # needs sage.graphs
sage: P = PairingHeap_of_n_hashables(10)
sage: P.push(('a', 1), (2, 'b'))
sage: P.push(2, (2, 'a'))
sage: g = Graph(2, immutable=True)
sage: P.push(g, (3, 'z'))
sage: P.top()
(2, (2, 'a'))
sage: P.decrease(g, (1, 'z'))
sage: P.top()
(Graph on 2 vertices, (1, 'z'))
sage: while P:
....:     print(P.top())
....:     P.pop()
(Graph on 2 vertices, (1, 'z'))
```

(continues on next page)

(continued from previous page)

```
(2, (2, 'a'))
(('a', 1), (2, 'b'))
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(10)); P
PairingHeap_of_n_hashables: capacity 10, size 0
>>> P.push(Integer(1), Integer(3))
>>> P.push('b', Integer(2))
>>> P.push((Integer(1), 'abc'), Integer(4))
>>> P.top()
('b', 2)
>>> P.decrease((Integer(1), 'abc'), Integer(1))
>>> P.top()
((1, 'abc'), 1)
>>> P.pop()
>>> P.top()
('b', 2)

>>> # needs sage.graphs
>>> P = PairingHeap_of_n_hashables(Integer(10))
>>> P.push(('a', Integer(1)), (Integer(2), 'b'))
>>> P.push(Integer(2), (Integer(2), 'a'))
>>> g = Graph(Integer(2), immutable=True)
>>> P.push(g, (Integer(3), 'z'))
>>> P.top()
(2, (2, 'a'))
>>> P.decrease(g, (Integer(1), 'z'))
>>> P.top()
(Graph on 2 vertices, (1, 'z'))
>>> while P:
...     print(P.top())
...     P.pop()
(Graph on 2 vertices, (1, 'z'))
(2, (2, 'a'))
(('a', 1), (2, 'b'))
```

class sage.data_structures.pairing_heap.**PairingHeap_class**

Bases: `object`

Common class and methods for *PairingHeap_of_n_integers* and *PairingHeap_of_n_hashables*.

capacity()

Return the maximum capacity of the heap.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.capacity()
5
sage: P.push(1, 2)
sage: P.capacity()
```

(continues on next page)

(continued from previous page)

```
5
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.capacity()
5
>>> P.push(Integer(1), Integer(2))
>>> P.capacity()
5
```

empty()

Check whether the heap is empty.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.empty()
True
sage: P.push(1, 2)
sage: P.empty()
False
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.empty()
True
>>> P.push(Integer(1), Integer(2))
>>> P.empty()
False
```

full()

Check whether the heap is full.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(2)
sage: P.full()
False
sage: P.push(0, 2)
sage: P.push(1, 3)
sage: P.full()
True
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(2))
>>> P.full()
False
```

(continues on next page)

(continued from previous page)

```
>>> P.push(Integer(0), Integer(2))
>>> P.push(Integer(1), Integer(3))
>>> P.full()
True
```

pop()

Remove the top item from the heap.

If the heap is already empty, we do nothing.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5); P
PairingHeap_of_n_integers: capacity 5, size 0
sage: P.push(1, 2); P
PairingHeap_of_n_integers: capacity 5, size 1
sage: P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 0
sage: P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 0
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5)); P
PairingHeap_of_n_integers: capacity 5, size 0
>>> P.push(Integer(1), Integer(2)); P
PairingHeap_of_n_integers: capacity 5, size 1
>>> P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 0
>>> P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 0
```

size()

Return the number of items in the heap.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.size()
0
sage: P.push(1, 2)
sage: P.size()
1
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.size()
0
>>> P.push(Integer(1), Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> P.size()
1
```

One may also use Python's `__len__`:

```
sage: len(P)
1
```

```
>>> from sage.all import *
>>> len(P)
1
```

top()

Return the top pair (item, value) of the heap.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.push(1, 2)
sage: P.top()
(1, 2)
sage: P.push(3, 1)
sage: P.top()
(3, 1)

sage: P = PairingHeap_of_n_integers(3)
sage: P.top()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.push(Integer(1), Integer(2))
>>> P.top()
(1, 2)
>>> P.push(Integer(3), Integer(1))
>>> P.top()
(3, 1)

>>> P = PairingHeap_of_n_integers(Integer(3))
>>> P.top()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

top_value()

Return the value of the top item of the heap.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.push(1, 2)
sage: P.top()
(1, 2)
sage: P.top_value()
2

sage: P = PairingHeap_of_n_integers(3)
sage: P.top_value()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.push(Integer(1), Integer(2))
>>> P.top()
(1, 2)
>>> P.top_value()
2

>>> P = PairingHeap_of_n_integers(Integer(3))
>>> P.top_value()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

class sage.data_structures.pairing_heap.**PairingHeap_of_n_hashables**

Bases: *PairingHeap_class*

Pairing Heap for n hashable items.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(5); P
PairingHeap_of_n_hashables: capacity 5, size 0
sage: P.push(1, 3)
sage: P.push('abc', 2); P
PairingHeap_of_n_hashables: capacity 5, size 2
sage: P.top()
('abc', 2)
sage: P.decrease(1, 1)
sage: P.top()
(1, 1)
sage: P.pop()
sage: P.top()
('abc', 2)

sage: P = PairingHeap_of_n_hashables(5)
sage: P.push(1, (2, 3))
```

(continues on next page)

(continued from previous page)

```
sage: P.push('a', (2, 2))
sage: P.push('b', (3, 3))
sage: P.push('c', (2, 1))
sage: P.top()
('c', (2, 1))
sage: P.push(Graph(2, immutable=True), (1, 7))
sage: P.top()
(Graph on 2 vertices, (1, 7))
sage: P.decrease('b', (1, 5))
sage: P.top()
('b', (1, 5))
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(5)); P
PairingHeap_of_n_hashables: capacity 5, size 0
>>> P.push(Integer(1), Integer(3))
>>> P.push('abc', Integer(2)); P
PairingHeap_of_n_hashables: capacity 5, size 2
>>> P.top()
('abc', 2)
>>> P.decrease(Integer(1), Integer(1))
>>> P.top()
(1, 1)
>>> P.pop()
>>> P.top()
('abc', 2)

>>> P = PairingHeap_of_n_hashables(Integer(5))
>>> P.push(Integer(1), (Integer(2), Integer(3)))
>>> P.push('a', (Integer(2), Integer(2)))
>>> P.push('b', (Integer(3), Integer(3)))
>>> P.push('c', (Integer(2), Integer(1)))
>>> P.top()
('c', (2, 1))
>>> P.push(Graph(Integer(2), immutable=True), (Integer(1), Integer(7)))
>>> P.top()
(Graph on 2 vertices, (1, 7))
>>> P.decrease('b', (Integer(1), Integer(5)))
>>> P.top()
('b', (1, 5))
```

contains(*item*)

Check whether the specified item is in the heap.

INPUT:

- `item` – the item to consider

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(5)
```

(continues on next page)

(continued from previous page)

```
sage: 3 in P
False
sage: P.push(3, 33)
sage: 3 in P
True
sage: 100 in P
False
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(5))
>>> Integer(3) in P
False
>>> P.push(Integer(3), Integer(33))
>>> Integer(3) in P
True
>>> Integer(100) in P
False
```

decrease(*item*, *new_value*)

Decrease the value of specified item.

This method is more permissive than it should as it can also be used to push an item in the heap.

INPUT:

- `item` – the item to consider
- `new_value` – the new value for `item`

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(5)
sage: 3 in P
False
sage: P.decrease(3, 33)
sage: 3 in P
True
sage: P.top()
(3, 33)
sage: P.push(1, 10)
sage: P.top()
(1, 10)
sage: P.decrease(3, 7)
sage: P.top()
(3, 7)
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(5))
>>> Integer(3) in P
False
>>> P.decrease(Integer(3), Integer(33))
```

(continues on next page)

(continued from previous page)

```
>>> Integer(3) in P
True
>>> P.top()
(3, 33)
>>> P.push(Integer(1), Integer(10))
>>> P.top()
(1, 10)
>>> P.decrease(Integer(3), Integer(7))
>>> P.top()
(3, 7)
```

pop()

Remove the top item from the heap.

If the heap is already empty, we do nothing.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(5); len(P)
0
sage: P.push(1, 2); len(P)
1
sage: P.push(2, 3); len(P)
2
sage: P.pop(); len(P)
1
sage: P.pop(); len(P)
0
sage: P.pop(); len(P)
0
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(5)); len(P)
0
>>> P.push(Integer(1), Integer(2)); len(P)
1
>>> P.push(Integer(2), Integer(3)); len(P)
2
>>> P.pop(); len(P)
1
>>> P.pop(); len(P)
0
>>> P.pop(); len(P)
0
```

push(*item*, *value*)

Insert an item into the heap with specified value (priority).

INPUT:

- `item` – a hashable object; the item to add
- `value` – the value associated with `item`

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(5)
sage: P.push(1, 2)
sage: P.top()
(1, 2)
sage: P.push(3, 1)
sage: P.top()
(3, 1)
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(5))
>>> P.push(Integer(1), Integer(2))
>>> P.top()
(1, 2)
>>> P.push(Integer(3), Integer(1))
>>> P.top()
(3, 1)
```

top()

Return the top pair (item, value) of the heap.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(5)
sage: P.push(1, 2)
sage: P.top()
(1, 2)
sage: P.push(3, 1)
sage: P.top()
(3, 1)

sage: P = PairingHeap_of_n_hashables(3)
sage: P.top()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(5))
>>> P.push(Integer(1), Integer(2))
>>> P.top()
(1, 2)
>>> P.push(Integer(3), Integer(1))
>>> P.top()
(3, 1)

>>> P = PairingHeap_of_n_hashables(Integer(3))
>>> P.top()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

top_item()

Return the top item of the heap.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(5)
sage: P.push(1, 2)
sage: P.top()
(1, 2)
sage: P.top_item()
1

sage: P = PairingHeap_of_n_hashables(3)
sage: P.top_item()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(5))
>>> P.push(Integer(1), Integer(2))
>>> P.top()
(1, 2)
>>> P.top_item()
1

>>> P = PairingHeap_of_n_hashables(Integer(3))
>>> P.top_item()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

value(*item*)

Return the value associated with the item.

INPUT:

- `item` – the item to consider

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
sage: P = PairingHeap_of_n_hashables(5)
sage: P.push(3, 33)
sage: P.push(1, 10)
sage: P.value(3)
33
sage: P.value(7)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: 7 is not in the heap
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_hashables
>>> P = PairingHeap_of_n_hashables(Integer(5))
>>> P.push(Integer(3), Integer(33))
>>> P.push(Integer(1), Integer(10))
>>> P.value(Integer(3))
33
>>> P.value(Integer(7))
Traceback (most recent call last):
...
ValueError: 7 is not in the heap
```

class sage.data_structures.pairing_heap.**PairingHeap_of_n_integers**

Bases: *PairingHeap_class*

Pairing Heap for items in range $[0, n-1]$.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5); P
PairingHeap_of_n_integers: capacity 5, size 0
sage: P.push(1, 3)
sage: P.push(2, 2)
sage: P
PairingHeap_of_n_integers: capacity 5, size 2
sage: P.top()
(2, 2)
sage: P.decrease(1, 1)
sage: P.top()
(1, 1)
sage: P.pop()
sage: P.top()
(2, 2)
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5)); P
PairingHeap_of_n_integers: capacity 5, size 0
>>> P.push(Integer(1), Integer(3))
>>> P.push(Integer(2), Integer(2))
>>> P
PairingHeap_of_n_integers: capacity 5, size 2
>>> P.top()
(2, 2)
>>> P.decrease(Integer(1), Integer(1))
>>> P.top()
(1, 1)
```

(continues on next page)

(continued from previous page)

```
>>> P.pop()
>>> P.top()
(2, 2)
```

contains(*item*)

Check whether the specified item is in the heap.

INPUT:

- `item` – nonnegative integer; the item to consider

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: 3 in P
False
sage: P.push(3, 33)
sage: 3 in P
True
sage: 100 in P
False
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> Integer(3) in P
False
>>> P.push(Integer(3), Integer(33))
>>> Integer(3) in P
True
>>> Integer(100) in P
False
```

decrease(*item*, *new_value*)

Decrease the value of specified item.

This method is more permissive than it should as it can also be used to push an item in the heap.

INPUT:

- `item` – nonnegative integer; the item to consider
- `new_value` – the new value for `item`

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: 3 in P
False
sage: P.decrease(3, 33)
sage: 3 in P
True
sage: P.top()
(3, 33)
```

(continues on next page)

(continued from previous page)

```
sage: P.push(1, 10)
sage: P.top()
(1, 10)
sage: P.decrease(3, 7)
sage: P.top()
(3, 7)
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> Integer(3) in P
False
>>> P.decrease(Integer(3), Integer(33))
>>> Integer(3) in P
True
>>> P.top()
(3, 33)
>>> P.push(Integer(1), Integer(10))
>>> P.top()
(1, 10)
>>> P.decrease(Integer(3), Integer(7))
>>> P.top()
(3, 7)
```

pop()

Remove the top item from the heap.

If the heap is already empty, we do nothing.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5); P
PairingHeap_of_n_integers: capacity 5, size 0
sage: P.push(1, 2); P
PairingHeap_of_n_integers: capacity 5, size 1
sage: P.push(2, 3); P
PairingHeap_of_n_integers: capacity 5, size 2
sage: P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 1
sage: P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 0
sage: P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 0
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5)); P
PairingHeap_of_n_integers: capacity 5, size 0
>>> P.push(Integer(1), Integer(2)); P
PairingHeap_of_n_integers: capacity 5, size 1
>>> P.push(Integer(2), Integer(3)); P
```

(continues on next page)

(continued from previous page)

```
PairingHeap_of_n_integers: capacity 5, size 2
>>> P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 1
>>> P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 0
>>> P.pop(); P
PairingHeap_of_n_integers: capacity 5, size 0
```

push(*item*, *value*)

Insert an item into the heap with specified value (priority).

INPUT:

- `item` – nonnegative integer; the item to consider
- `value` – the value associated with `item`

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.push(1, 2)
sage: P.top()
(1, 2)
sage: P.push(3, 1)
sage: P.top()
(3, 1)
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.push(Integer(1), Integer(2))
>>> P.top()
(1, 2)
>>> P.push(Integer(3), Integer(1))
>>> P.top()
(3, 1)
```

top()

Return the top pair (item, value) of the heap.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.push(1, 2)
sage: P.top()
(1, 2)
sage: P.push(3, 1)
sage: P.top()
(3, 1)

sage: P = PairingHeap_of_n_integers(3)
sage: P.top()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.push(Integer(1), Integer(2))
>>> P.top()
(1, 2)
>>> P.push(Integer(3), Integer(1))
>>> P.top()
(3, 1)

>>> P = PairingHeap_of_n_integers(Integer(3))
>>> P.top()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

top_item()

Return the top item of the heap.

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.push(1, 2)
sage: P.top()
(1, 2)
sage: P.top_item()
1

sage: P = PairingHeap_of_n_integers(3)
sage: P.top_item()
Traceback (most recent call last):
...
ValueError: trying to access the top of an empty heap
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.push(Integer(1), Integer(2))
>>> P.top()
(1, 2)
>>> P.top_item()
1

>>> P = PairingHeap_of_n_integers(Integer(3))
>>> P.top_item()
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: trying to access the top of an empty heap
```

value(*item*)

Return the value associated with the item.

INPUT:

- `item` – nonnegative integer; the item to consider

EXAMPLES:

```
sage: from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
sage: P = PairingHeap_of_n_integers(5)
sage: P.push(3, 33)
sage: P.push(1, 10)
sage: P.value(3)
33
sage: P.value(7)
Traceback (most recent call last):
...
ValueError: 7 is not in the heap
```

```
>>> from sage.all import *
>>> from sage.data_structures.pairing_heap import PairingHeap_of_n_integers
>>> P = PairingHeap_of_n_integers(Integer(5))
>>> P.push(Integer(3), Integer(33))
>>> P.push(Integer(1), Integer(10))
>>> P.value(Integer(3))
33
>>> P.value(Integer(7))
Traceback (most recent call last):
...
ValueError: 7 is not in the heap
```

CHAPTER

SEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

d

m

INDEX

F

G

I

K

L

M

N

O

P

R

S

T

U

V