Sage Reference Manual: Discrete dynamics

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

INTERVAL EXCHANGE TRANSFORMATIONS AND LINEAR INVOLUTIONS

1.1 Class factories for Interval exchange transformations.

This library is designed for the usage and manipulation of interval exchange transformations and linear involutions. It defines specialized types of permutation (constructed using `iet.Permutation()`) some associated graph (constructed using `iet.RauzyGraph()`) and some maps of intervals (constructed using `iet.IntervalExchangeTransformation()`).

EXAMPLES:

Creation of an interval exchange transformation:

```
sage: T = iet.IntervalExchangeTransformation(('a b','b a'),(sqrt(2),1))
sage: print T
Interval exchange transformation of [0, sqrt(2) + 1[ with permutation
a b
b a
```

It can also be initialized using permutation (group theoretic ones):

```
sage: p = Permutation([3,2,1])
sage: T = iet.IntervalExchangeTransformation(p, [1/3,2/3,1])
sage: print T
Interval exchange transformation of [0, 2[ with permutation
1 2 3
3 2 1
```

For the manipulation of permutations of iet, there are special types provided by this module. All of them can be constructed using the constructor iet.Permutation. For the creation of labelled permutations of interval exchange transformation:

```
sage: p1 = iet.Permutation('a b c', 'c b a')
sage: print p1
a b c
c b a
```

They can be used for initialization of an iet:

```
sage: p = iet.Permutation('a b','b a')
sage: T = iet.IntervalExchangeTransformation(p, [1,sqrt(2)])
sage: print T
Interval exchange transformation of [0, sqrt(2) + 1[ with permutation
a b
b a
```

You can also, create labelled permutations of linear involutions:

```
sage: p = iet.GeneralizedPermutation('a a b', 'b c c')
sage: print p
a a b
b c c
```

Sometimes it's more easy to deal with reduced permutations:

```
sage: p = iet.Permutation('a b c', 'c b a', reduced = True)
sage: print p
a b c
c b a
```

Permutations with flips:

```
sage: p1 = iet.Permutation('a b c', 'c b a', flips = ['a','c'])
sage: print p1
-a  b -c
-c  b -a
```

Creation of Rauzy diagrams:

```
sage: r = iet.RauzyDiagram('a b c', 'c b a')
```

Reduced Rauzy diagrams are constructed using the same arguments than for permutations:

```
sage: r = iet.RauzyDiagram('a b b','c c a')
sage: r_red = iet.RauzyDiagram('a b b','c c a',reduced=True)
sage: r.cardinality()
12
sage: r_red.cardinality()
4
```

By defaut, Rauzy diagram are generated by induction on the right. You can use several options to enlarge (or restrict) the diagram (try help(iet.RauzyDiagram) for more precisions):

```
sage: r1 = iet.RauzyDiagram('a b c','c b a',right_induction=True)
sage: r2 = iet.RauzyDiagram('a b c','c b a',left_right_inversion=True)
```

You can consider self similar iet using path in Rauzy diagrams and eigenvectors of the corresponding matrix:

```
sage: p = iet.Permutation("a b c d", "d c b a")
sage: d = p.rauzy_diagram()
sage: g = d.path(p, 't', 't', 'b', 't', 'b', 'b', 't', 'b')
sage: g
Path of length 8 in a Rauzy diagram
sage: g.is_loop()
True
sage: g.is_full()
True
```

```
sage: m = g.matrix()
sage: v = m.eigenvectors_right()[-1][1][0]
sage: T1 = iet.IntervalExchangeTransformation(p, v)
sage: T2 = T1.rauzy_move(iterations=8)
sage: T1.normalize(1) == T2.normalize(1)
True
```

REFERENCES:

AUTHORS:

- Vincent Delecroix (2009-09-29): initial version

sage.dynamics.interval_exchanges.constructors.**GeneralizedPermutation**(**args*,
**kargs)

> Returns a permutation of an interval exchange transformation.

> Those permutations are the combinatoric part of linear involutions and were introduced by Danthony-Nogueira [DN90]. The full combinatoric study and precise links with strata of quadratic differentials was achieved few years later by Boissy-Lanneau [BL08].

> INPUT:

>> •`intervals` - strings, list, tuples

>> •`reduced` - boolean (defaut: False) specifies reduction. False means labelled permutation and True means reduced permutation.

>> •`flips` - iterable (default: None) the letters which correspond to flipped intervals.

> OUTPUT:

> generalized permutation – the output type depends on the data.

> EXAMPLES:

> Creation of labelled generalized permutations:
> ```
> sage: iet.GeneralizedPermutation('a b b','c c a')
> a b b
> c c a
> sage: iet.GeneralizedPermutation('a a','b b c c')
> a a
> b b c c
> sage: iet.GeneralizedPermutation([[0,1,2,3,1],[4,2,5,3,5,4,0]])
> 0 1 2 3 1
> 4 2 5 3 5 4 0
> ```

> Creation of reduced generalized permutations:
> ```
> sage: iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
> a b b
> c c a
> sage: iet.GeneralizedPermutation('a a b b', 'c c d d', reduced = True)
> a a b b
> c c d d
> ```

> Creation of flipped generalized permutations:
> ```
> sage: iet.GeneralizedPermutation('a b c a', 'd c d b', flips = ['a','b'])
> -a -b  c -a
>  d  c  d -b
> ```

TESTS:
```
sage: iet.GeneralizedPermutation('a a b b', 'c c d d', reduced = 'may')
Traceback (most recent call last):
...
TypeError: reduced must be of type boolean
sage: iet.GeneralizedPermutation('a b c a', 'd c d b', flips = ['e','b'])
Traceback (most recent call last):
...
TypeError: The flip list is not valid
sage: iet.GeneralizedPermutation('a b c a', 'd c c b', flips = ['a','b'])
Traceback (most recent call last):
...
ValueError: Letters must reappear twice
```

sage.dynamics.interval_exchanges.constructors.**IET** (*permutation=None*, *lengths=None*)
Constructs an Interval exchange transformation.

An interval exchange transformation (or iet) is a map from an interval to itself. It is defined on the interval except at a finite number of points (the singularities) and is a translation on each connected component of the complement of the singularities. Moreover it is a bijection on its image (or it is injective).

An interval exchange transformation is encoded by two datas. A permutation (that corresponds to the way we echange the intervals) and a vector of positive reals (that corresponds to the lengths of the complement of the singularities).

INPUT:

- •permutation - a permutation

- •lengths - a list or a dictionnary of lengths

OUTPUT:

interval exchange transformation – an map of an interval

EXAMPLES:

Two initialization methods, the first using a iet.Permutation:
```
sage: p = iet.Permutation('a b c','c b a')
sage: t = iet.IntervalExchangeTransformation(p, {'a':1,'b':0.4523,'c':2.8})
```

The second is more direct:
```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),{'a':1,'b':4})
```

It's also possible to initialize the lengths only with a list:
```
sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,0.4,2])
```

The two fundamental operations are Rauzy move and normalization:
```
sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,0.4,2])
sage: s = t.rauzy_move()
sage: s_n = s.normalize(t.length())
sage: s_n.length() == t.length()
True
```

A not too simple example of a self similar interval exchange transformation:
```
sage: p = iet.Permutation('a b c d','d c b a')
sage: d = p.rauzy_diagram()
sage: g = d.path(p, 't', 't', 'b', 't', 'b', 'b', 't', 'b')
```

```
sage: m = g.matrix()
sage: v = m.eigenvectors_right()[-1][1][0]
sage: t = iet.IntervalExchangeTransformation(p,v)
sage: s = t.rauzy_move(iterations=8)
sage: s.normalize() == t.normalize()
True
```

TESTS:
```
sage: iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,2])
Traceback (most recent call last):
...
ValueError: bad number of lengths
sage: iet.IntervalExchangeTransformation(('a b c','c b a'),[0.1,'rho',2])
Traceback (most recent call last):
...
TypeError: unable to convert x (='rho') into a real number
sage: iet.IntervalExchangeTransformation(('a b c','c b a'),[0.1,-2,2])
Traceback (most recent call last):
...
ValueError: lengths must be positive
```

`sage.dynamics.interval_exchanges.constructors.`**`IntervalExchangeTransformation`**(*permutation=None,*
lengths=None)

Constructs an Interval exchange transformation.

An interval exchange transformation (or iet) is a map from an interval to itself. It is defined on the interval except at a finite number of points (the singularities) and is a translation on each connected component of the complement of the singularities. Moreover it is a bijection on its image (or it is injective).

An interval exchange transformation is encoded by two datas. A permutation (that corresponds to the way we echange the intervals) and a vector of positive reals (that corresponds to the lengths of the complement of the singularities).

INPUT:

- `permutation` - a permutation

- `lengths` - a list or a dictionnary of lengths

OUTPUT:

interval exchange transformation – an map of an interval

EXAMPLES:

Two initialization methods, the first using a iet.Permutation:
```
sage: p = iet.Permutation('a b c','c b a')
sage: t = iet.IntervalExchangeTransformation(p, {'a':1,'b':0.4523,'c':2.8})
```

The second is more direct:
```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),{'a':1,'b':4})
```

It's also possible to initialize the lengths only with a list:
```
sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,0.4,2])
```

The two fundamental operations are Rauzy move and normalization:
```
sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,0.4,2])
sage: s = t.rauzy_move()
```

```
sage: s_n = s.normalize(t.length())
sage: s_n.length() == t.length()
True
```

A not too simple example of a self similar interval exchange transformation:
```
sage: p = iet.Permutation('a b c d','d c b a')
sage: d = p.rauzy_diagram()
sage: g = d.path(p, 't', 't', 'b', 't', 'b', 'b', 't', 'b')
sage: m = g.matrix()
sage: v = m.eigenvectors_right()[-1][1][0]
sage: t = iet.IntervalExchangeTransformation(p,v)
sage: s = t.rauzy_move(iterations=8)
sage: s.normalize() == t.normalize()
True
```

TESTS:
```
sage: iet.IntervalExchangeTransformation(('a b c','c b a'),[0.123,2])
Traceback (most recent call last):
...
ValueError: bad number of lengths
sage: iet.IntervalExchangeTransformation(('a b c','c b a'),[0.1,'rho',2])
Traceback (most recent call last):
...
TypeError: unable to convert x (='rho') into a real number
sage: iet.IntervalExchangeTransformation(('a b c','c b a'),[0.1,-2,2])
Traceback (most recent call last):
...
ValueError: lengths must be positive
```

sage.dynamics.interval_exchanges.constructors.**Permutation**(*args*, **kargs*)
> Returns a permutation of an interval exchange transformation.

> Those permutations are the combinatoric part of an interval exchange transformation (IET). The combinatorial study of those objects starts with Gerard Rauzy [R79] and William Veech [V78].

> The combinatoric part of interval exchange transformation can be taken independently from its dynamical origin. It has an important link with strata of Abelian differential (see `strata`)

> INPUT:

> - `intervals` - string, two strings, list, tuples that can be converted to two lists

> - `reduced` - boolean (default: False) specifies reduction. False means labelled permutation and True means reduced permutation.

> - `flips` - iterable (default: None) the letters which correspond to flipped intervals.

> OUTPUT:

> permutation – the output type depends of the data.

> EXAMPLES:

> Creation of labelled permutations
```
sage: iet.Permutation('a b c d','d c b a')
a b c d
d c b a
sage: iet.Permutation([[0,1,2,3],[2,1,3,0]])
0 1 2 3
2 1 3 0
```

```
sage: iet.Permutation([0, 'A', 'B', 1], ['B', 0, 1, 'A'])
0 A B 1
B 0 1 A
```

Creation of reduced permutations:

```
sage: iet.Permutation('a b c', 'c b a', reduced = True)
a b c
c b a
sage: iet.Permutation([0, 1, 2, 3], [1, 3, 0, 2])
0 1 2 3
1 3 0 2
```

Creation of flipped permutations:

```
sage: iet.Permutation('a b c', 'c b a', flips=['a','b'])
-a -b  c
 c -b -a
sage: iet.Permutation('a b c', 'c b a', flips=['a'], reduced=True)
-a  b  c
 c  b -a
```

TESTS:

```
sage: p = iet.Permutation('a b c','c b a')
sage: iet.Permutation(p) == p
True
sage: iet.Permutation(p, reduced=True) == p.reduced()
True

sage: p = iet.Permutation('a','a',flips='a',reduced=True)
sage: iet.Permutation(p) == p
True

sage: p = iet.Permutation('a b c','c b a',flips='a')
sage: iet.Permutation(p) == p
True
sage: iet.Permutation(p, reduced=True) == p.reduced()
True

sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: iet.Permutation(p) == p
True
```

TESTS:

```
sage: iet.Permutation('a b c','c b a',reduced='badly')
Traceback (most recent call last):
...
TypeError: reduced must be of type boolean
sage: iet.Permutation('a','a',flips='b',reduced=True)
Traceback (most recent call last):
...
ValueError: flips contains not valid letters
sage: iet.Permutation('a b c','c a a',reduced=True)
Traceback (most recent call last):
...
ValueError: letters must appear once in each interval
```

sage.dynamics.interval_exchanges.constructors.**Permutations_iterator**(*nintervals=None*,
irreducible=True,
reduced=False,
alphabet=None)

Returns an iterator over permutations.

This iterator allows you to iterate over permutations with given constraints. If you want to iterate over permutations coming from a given stratum you have to use the module `strata` and generate Rauzy diagrams from connected components.

INPUT:

- `nintervals` - non negative integer

- `irreducible` - boolean (default: True)

- `reduced` - boolean (default: False)

- `alphabet` - alphabet (default: None)

OUTPUT:

iterator – an iterator over permutations

EXAMPLES:

Generates all reduced permutations with given number of intervals:

```
sage: P = iet.Permutations_iterator(nintervals=2,alphabet="ab",reduced=True)
sage: for p in P: print p, "\n* *"
a b
b a
* *
sage: P = iet.Permutations_iterator(nintervals=3,alphabet="abc",reduced=True)
sage: for p in P: print p, "\n* * *"
a b c
b c a
* * *
a b c
c a b
* * *
a b c
c b a
* * *
```

TESTS:

```
sage: P = iet.Permutations_iterator(nintervals=None, alphabet=None)
Traceback (most recent call last):
...
ValueError: You must specify an alphabet or a length
sage: P = iet.Permutations_iterator(nintervals=None, alphabet=ZZ)
Traceback (most recent call last):
...
ValueError: You must specify a length with infinite alphabet
```

sage.dynamics.interval_exchanges.constructors.**RauzyDiagram**(**args*, ***kargs*)

Return an object coding a Rauzy diagram.

The Rauzy diagram is an oriented graph with labelled edges. The set of vertices corresponds to the permutations obtained by different operations (mainly the .rauzy_move() operations that corresponds to an induction of interval exchange transformation). The edges correspond to the action of the different operations considered.

It first appeard in the original article of Rauzy [R79].

INPUT:

- `intervals` - lists, or strings, or tuples

- `reduced` - boolean (default: False) to precise reduction

- `flips` - list (default: []) for flipped permutations

- `right_induction` - boolean (default: True) consideration of left induction in the diagram

- `left_induction` - boolean (default: False) consideration of right induction in the diagram

- `left_right_inversion` - boolean (default: False) consideration of inversion

- `top_bottom_inversion` - boolean (default: False) consideration of reversion

- `symmetric` - boolean (default: False) consideration of the symmetric operation

OUTPUT:

Rauzy diagram – the Rauzy diagram that corresponds to your request

EXAMPLES:

Standard Rauzy diagrams:

```
sage: iet.RauzyDiagram('a b c d', 'd b c a')
Rauzy diagram with 12 permutations
sage: iet.RauzyDiagram('a b c d', 'd b c a', reduced = True)
Rauzy diagram with 6 permutations
```

Extended Rauzy diagrams:

```
sage: iet.RauzyDiagram('a b c d', 'd b c a', symmetric=True)
Rauzy diagram with 144 permutations
```

Using Rauzy diagrams and path in Rauzy diagrams:

```
sage: r = iet.RauzyDiagram('a b c', 'c b a')
sage: print r
Rauzy diagram with 3 permutations
sage: p = iet.Permutation('a b c','c b a')
sage: p in r
True
sage: g0 = r.path(p, 'top', 'bottom','top')
sage: g1 = r.path(p, 'bottom', 'top', 'bottom')
sage: print g0.is_loop(), g1.is_loop()
True True
sage: print g0.is_full(), g1.is_full()
False False
sage: g = g0 + g1
sage: g
Path of length 6 in a Rauzy diagram
sage: print g.is_loop(), g.is_full()
True True
sage: m = g.matrix()
sage: print m
[1 1 1]
[2 4 1]
```

```
        [2 3 2]
sage: s = g.orbit_substitution()
sage: s
WordMorphism: a->acbbc, b->acbbcbbc, c->acbc
sage: s.incidence_matrix() == m
True
```

We can then create the corresponding interval exchange transformation and comparing the orbit of 0 to the fixed point of the orbit substitution:

```
sage: v = m.eigenvectors_right()[-1][1][0]
sage: T = iet.IntervalExchangeTransformation(p, v).normalize()
sage: print T
Interval exchange transformation of [0, 1[ with permutation
a b c
c b a
sage: w1 = []
sage: x = 0
sage: for i in range(20):
....:    w1.append(T.in_which_interval(x))
....:    x = T(x)
sage: w1 = Word(w1)
sage: w1
word: acbbcacbcacbbcbbcacb
sage: w2 = s.fixed_point('a')
sage: w2[:20]
word: acbbcacbcacbbcbbcacb
sage: w2[:20] == w1
True
```

1.2 Labelled permutations

A labelled (generalized) permutation is better suited to study the dynamic of a translation surface than a reduced one (see the module `sage.dynamics.interval_exchanges.reduced`). The latter is more adapted to the study of strata. This kind of permutation was introduced by Yoccoz [Yoc05] (see also [MMY03]).

In fact, there is a geometric counterpart of labelled permutations. They correspond to translation surfaces with marked outgoing separatrices (i.e. we fix a label for each of them).

Remarks that Rauzy diagram of reduced objects are significantly smaller than the one for labelled object (for the permutation a b d b e / e d c a c the labelled Rauzy diagram contains 8760 permutations, and the reduced only 73). But, as it is in geometrical way, the labelled Rauzy diagram is a covering of the reduced Rauzy diagram.

AUTHORS:

- Vincent Delecroix (2009-09-29) : initial version

TESTS:

```
sage: from sage.dynamics.interval_exchanges.labelled import LabelledPermutationIET
sage: LabelledPermutationIET([['a', 'b', 'c'], ['c', 'b', 'a']])
a b c
c b a
sage: LabelledPermutationIET([[1,2,3,4],[4,1,2,3]])
1 2 3 4
4 1 2 3
sage: from sage.dynamics.interval_exchanges.labelled import LabelledPermutationLI
```

```
sage: LabelledPermutationLI([[1,1],[2,2,3,3,4,4]])
1 1
2 2 3 3 4 4
sage: LabelledPermutationLI([['a','a','b','b','c','c'],['d','d']])
a a b b c c
d d
sage: from sage.dynamics.interval_exchanges.labelled import FlippedLabelledPermutationIET
sage: FlippedLabelledPermutationIET([[1,2,3],[3,2,1]],flips=[1,2])
-1 -2  3
 3 -2 -1
sage: FlippedLabelledPermutationIET([['a','b','c'],['b','c','a']],flips='b')
 a -b  c
-b  c  a
sage: from sage.dynamics.interval_exchanges.labelled import FlippedLabelledPermutationLI
sage: FlippedLabelledPermutationLI([[1,1],[2,2,3,3,4,4]], flips=[1,4])
-1 -1
 2  2  3  3 -4 -4
sage: FlippedLabelledPermutationLI([['a','a','b','b'],['c','c']],flips='ac')
-a -a  b  b
-c -c
sage: from sage.dynamics.interval_exchanges.labelled import LabelledRauzyDiagram
sage: p = LabelledPermutationIET([[1,2,3],[3,2,1]])
sage: d1 = LabelledRauzyDiagram(p)
sage: p = LabelledPermutationIET([['a','b'],['b','a']])
sage: d = p.rauzy_diagram()
sage: g1 = d.path(p, 'top', 'bottom')
sage: g1.matrix()
[1 1]
[1 2]
sage: g2 = d.path(p, 'bottom', 'top')
sage: g2.matrix()
[2 1]
[1 1]
sage: p = LabelledPermutationIET([['a','b','c','d'],['d','c','b','a']])
sage: d = p.rauzy_diagram()
sage: g = d.path(p, 't', 't', 'b', 't', 'b', 'b', 't', 'b')
sage: g
Path of length 8 in a Rauzy diagram
sage: g.is_loop()
True
sage: g.is_full()
True
sage: s1 = g.orbit_substitution()
sage: s1
WordMorphism: a->adbd, b->adbdbd, c->adccd, d->adcd
sage: s2 = g.interval_substitution()
sage: s2
WordMorphism: a->abcd, b->bab, c->cdc, d->dcbababcd
sage: s1.incidence_matrix() == s2.incidence_matrix().transpose()
True
```

REFERENCES:

class sage.dynamics.interval_exchanges.labelled.**FlippedLabelledPermutation**(*intervals=None,*
al-
pha-
bet=None,
flips=None)

Bases: `sage.dynamics.interval_exchanges.labelled.LabelledPermutation`

General template for labelled objects

> **Warning:** Internal class! Do not use directly!

list (*flips=False*)

Returns a list associated to the permutation.

INPUT:

> •`flips` - boolean (default: `False`)

OUTPUT:

list – two lists of labels

EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('0 0 1 2 2 1', '3 3', flips='1')
sage: p.list(flips=True)
[[('0', 1), ('0', 1), ('1', -1), ('2', 1), ('2', 1), ('1', -1)], [('3', 1), ('3', 1)]]
sage: p.list(flips=False)
[['0', '0', '1', '2', '2', '1'], ['3', '3']]
```

The list can be used to reconstruct the permutation
```
sage: p = iet.Permutation('a b c','c b a',flips='ab')
sage: p == iet.Permutation(p.list(), flips=p.flips())
True

sage: p = iet.GeneralizedPermutation('a b b c','c d d a',flips='ad')
sage: p == iet.GeneralizedPermutation(p.list(),flips=p.flips())
True
```

class sage.dynamics.interval_exchanges.labelled.**FlippedLabelledPermutationIET** (*intervals=None, alphabet=None, flips=None*)

Bases: `sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutation`, `sage.dynamics.interval_exchanges.template.FlippedPermutationIET`, `sage.dynamics.interval_exchanges.labelled.LabelledPermutationIET`

Flipped labelled permutation from iet.

EXAMPLES:

Reducibility testing (does not depends of flips):
```
sage: p = iet.Permutation('a b c', 'c b a',flips='a')
sage: p.is_irreducible()
True
sage: q = iet.Permutation('a b c d', 'b a d c', flips='bc')
sage: q.is_irreducible()
False
```

Rauzy movability and Rauzy move:
```
sage: p = iet.Permutation('a b c', 'c b a',flips='a')
sage: print p
-a  b  c
 c  b -a
```

```
sage: print p.rauzy_move(1)
-c -a  b
-c  b -a
sage: print p.rauzy_move(0)
-a  b  c
 c -a  b
```

Rauzy diagrams:
```
sage: d = iet.RauzyDiagram('a b c d','d a b c',flips='a')
```

AUTHORS:

- Vincent Delecroix (2009-09-29): initial version

rauzy_diagram(***kargs*)
    Returns the Rauzy diagram associated to this permutation.

    For more information, try help(iet.RauzyDiagram)

    OUTPUT:

    RauzyDiagram – the Rauzy diagram of `self`

    EXAMPLES:
```
sage: p = iet.Permutation('a b c', 'c b a',flips='a')
sage: p.rauzy_diagram()
Rauzy diagram with 3 permutations
```

rauzy_move(*winner=None*, *side=None*)
    Returns the Rauzy move.

    INPUT:

    - `winner` - 'top' (or 't' or 0) or 'bottom' (or 'b' or 1)

    - `side` - (default: 'right') 'right' (or 'r') or 'left' (or 'l')

    OUTPUT:

    permutation – the Rauzy move of `self`

    EXAMPLES:
```
sage: p = iet.Permutation('a b','b a',flips='a')
sage: p.rauzy_move('top')
-a  b
 b -a
sage: p.rauzy_move('bottom')
-b -a
-b -a

sage: p = iet.Permutation('a b c','c b a',flips='b')
sage: p.rauzy_move('top')
 a -b  c
 c  a -b
sage: p.rauzy_move('bottom')
 a  c -b
 c -b  a
```

reduced()
    The associated reduced permutation.

permutation – the associated reduced permutation

EXAMPLES:
```
sage: p = iet.Permutation('a b c','c b a',flips='a')
sage: q = iet.Permutation('a b c','c b a',flips='a',reduced=True)
sage: p.reduced() == q
True
```

class sage.dynamics.interval_exchanges.labelled.**FlippedLabelledPermutationLI**(*intervals=None*, *alphabet=None*, *flips=None*)

   Bases: `sage.dynamics.interval_exchanges.labelled.FlippedLabelledPermutation`, `sage.dynamics.interval_exchanges.template.FlippedPermutationLI`, `sage.dynamics.interval_exchanges.labelled.LabelledPermutationLI`

   Flipped labelled quadratic (or generalized) permutation.

   EXAMPLES:

   Reducibility testing:
```
sage: p = iet.GeneralizedPermutation('a b b', 'c c a', flips='a')
sage: p.is_irreducible()
True
```

   Reducibility testing with associated decomposition:
```
sage: p = iet.GeneralizedPermutation('a b c a', 'b d d c', flips='ab')
sage: p.is_irreducible()
False
sage: test, decomp = p.is_irreducible(return_decomposition = True)
sage: print test
False
sage: print decomp
(['a'], ['c', 'a'], [], ['c'])
```

   Rauzy movability and Rauzy move:
```
sage: p = iet.GeneralizedPermutation('a a b b c c', 'd d', flips='d')
sage: p.has_rauzy_move(0)
False
sage: p.has_rauzy_move(1)
True
sage: p = iet.GeneralizedPermutation('a a b','b c c',flips='c')
sage: p.has_rauzy_move(0)
True
sage: p.has_rauzy_move(1)
True
```

   left_rauzy_move(*winner*)
      Perform a Rauzy move on the left.

      INPUT:

         •`winner` - either 'top' or 'bottom' ('t' or 'b' for short)

      OUTPUT:

– a permutation

EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a b','b c c')
sage: p.left_rauzy_move(0)
a a b b
c c
sage: p.left_rauzy_move(1)
a a b
b c c

sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.left_rauzy_move(0)
a b b
c c a
sage: p.left_rauzy_move(1)
b b
c c a a
```

rauzy_diagram(***kargs*)

Returns the associated Rauzy diagram.

For more information, try help(RauzyDiagram)

OUTPUT :

– a RauzyDiagram

EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a b b a', 'c d c d')
sage: d = p.rauzy_diagram()
```

reduced()

The associated reduced permutation.

OUTPUT:

permutation – the associated reduced permutation

EXAMPLE:
```
sage: p = iet.GeneralizedPermutation('a a','b b c c',flips='a')
sage: q = iet.GeneralizedPermutation('a a','b b c c',flips='a',reduced=True)
sage: p.reduced() == q
True
```

right_rauzy_move(*winner*)

Perform a Rauzy move on the right (the standard one).

INPUT:

  • `winner` - either 'top' or 'bottom' ('t' or 'b' for short)

OUTPUT:

permutation – the Rauzy move of `self`

EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a b','b c c',flips='c')
sage: p.right_rauzy_move(0)
 a  a  b
-c  b -c
```

```
sage: p.right_rauzy_move(1)
 a   a
-b -c -b -c

sage: p = iet.GeneralizedPermutation('a b b','c c a',flips='ab')
sage: p.right_rauzy_move(0)
 a -b  a -b
 c   c
sage: p.right_rauzy_move(1)
 b -a  b
 c   c -a
```

class sage.dynamics.interval_exchanges.labelled.**FlippedLabelledRauzyDiagram**(*p*, *right_induction=True*, *left_induction=False*, *left_right_inversion=Fal*, *top_bottom_inversion=F*, *symmetric=False*)

    Bases: sage.dynamics.interval_exchanges.template.FlippedRauzyDiagram, sage.dynamics.interval_exchanges.labelled.LabelledRauzyDiagram

    Rauzy diagram of flipped labelled permutations

class sage.dynamics.interval_exchanges.labelled.**LabelledPermutation**(*intervals=None*, *alphabet=None*)

    Bases: sage.structure.sage_object.SageObject

    General template for labelled objects.

> **Warning:** Internal class! Do not use directly!

    erase_letter(*letter*)
        Return the permutation with the specified letter removed.

        OUTPUT:

        permutation – the resulting permutation

        EXAMPLES:
```
sage: p = iet.Permutation('a b c d','c d b a')
sage: p.erase_letter('a')
b c d
c d b
sage: p.erase_letter('b')
a c d
c d a
sage: p.erase_letter('c')
a b d
d b a
sage: p.erase_letter('d')
a b c
c b a

sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.erase_letter('a')
```

```
b b
c c
```

Beware, there is no validity check for permutation from linear involutions:
```
sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.erase_letter('b')
a
c c a
```

length(*interval=None*)

Returns a 2-uple of lengths.

p.length() is identical to (p.length_top(), p.length_bottom()) If an interval is specified, it returns the length of the specified interval.

INPUT:

- • interval - None, 'top' or 'bottom'

OUTPUT:

tuple – a 2-uple of integers

EXAMPLES:
```
sage: iet.Permutation('a b c','c b a').length()
(3, 3)
sage: iet.GeneralizedPermutation('a a','b b c c').length()
(2, 4)
sage: iet.GeneralizedPermutation('a a b b','c c').length()
(4, 2)
```

length_bottom()

Returns the number of intervals in the bottom segment.

OUTPUT:

integer – number of intervals

EXAMPLES:
```
sage: iet.Permutation('a b','b a').length_bottom()
2
sage: iet.GeneralizedPermutation('a a','b b c c').length_bottom()
4
sage: iet.GeneralizedPermutation('a a b b','c c').length_bottom()
2
```

length_top()

Returns the number of intervals in the top segment.

OUTPUT:

integer – number of intervals

EXAMPLES:
```
sage: iet.Permutation('a b c','c b a').length_top()
3
sage: iet.GeneralizedPermutation('a a','b b c c').length_top()
2
sage: iet.GeneralizedPermutation('a a b b','c c').length_top()
4
```

list()
> Returns a list of two lists corresponding to the intervals.
>
> OUTPUT:
>
> list – two lists of labels
>
> EXAMPLES:
>
> The list of an permutation from iet:
> ```
> sage: p1 = iet.Permutation('1 2 3', '3 1 2')
> sage: p1.list()
> [['1', '2', '3'], ['3', '1', '2']]
> sage: p1.alphabet("abc")
> sage: p1.list()
> [['a', 'b', 'c'], ['c', 'a', 'b']]
> ```
>
> Recovering the permutation from this list (and the alphabet):
> ```
> sage: q1 = iet.Permutation(p1.list(),alphabet=p1.alphabet())
> sage: p1 == q1
> True
> ```
>
> The list of a quadratic permutation:
> ```
> sage: p2 = iet.GeneralizedPermutation('g o o', 'd d g')
> sage: p2.list()
> [['g', 'o', 'o'], ['d', 'd', 'g']]
> ```
>
> Recovering the permutation:
> ```
> sage: q2 = iet.GeneralizedPermutation(p2.list(),alphabet=p2.alphabet())
> sage: p2 == q2
> True
> ```

rauzy_move_loser(*winner=None*, *side=None*)
> Returns the loser of a Rauzy move
>
> INPUT:
>
> > • `winner` - either 'top' or 'bottom' ('t' or 'b' for short)
> >
> > • `side` - either 'left' or 'right' ('l' or 'r' for short)
>
> OUTPUT:
>
> – a label
>
> EXAMPLES:
> ```
> sage: p = iet.Permutation('a b c d','b d a c')
> sage: p.rauzy_move_loser('top','right')
> 'c'
> sage: p.rauzy_move_loser('bottom','right')
> 'd'
> sage: p.rauzy_move_loser('top','left')
> 'b'
> sage: p.rauzy_move_loser('bottom','left')
> 'a'
> ```

rauzy_move_matrix(*winner=None*, *side='right'*)
> Returns the Rauzy move matrix.

This matrix corresponds to the action of a Rauzy move on the vector of lengths. By convention (to get a positive matrix), the matrix is defined as the inverse transformation on the length vector.

OUTPUT:

matrix – a square matrix of positive integers

EXAMPLES:
```
sage: p = iet.Permutation('a b','b a')
sage: p.rauzy_move_matrix('t')
[1 0]
[1 1]
sage: p.rauzy_move_matrix('b')
[1 1]
[0 1]

sage: p = iet.Permutation('a b c d','b d a c')
sage: q = p.left_right_inverse()
sage: m0 = p.rauzy_move_matrix(winner='top',side='right')
sage: n0 = q.rauzy_move_matrix(winner='top',side='left')
sage: m0 == n0
True
sage: m1 = p.rauzy_move_matrix(winner='bottom',side='right')
sage: n1 = q.rauzy_move_matrix(winner='bottom',side='left')
sage: m1 == n1
True
```

rauzy_move_winner (*winner=None*, *side=None*)
    Returns the winner of a Rauzy move.

    INPUT:

        • `winner` - either 'top' or 'bottom' ('t' or 'b' for short)

        • `side` - either 'left' or 'right' ('l' or 'r' for short)

    OUTPUT:

    – a label

    EXAMPLES:
```
sage: p = iet.Permutation('a b c d','b d a c')
sage: p.rauzy_move_winner('top','right')
'd'
sage: p.rauzy_move_winner('bottom','right')
'c'
sage: p.rauzy_move_winner('top','left')
'a'
sage: p.rauzy_move_winner('bottom','left')
'b'

sage: p = iet.GeneralizedPermutation('a b b c','d c a e d e')
sage: p.rauzy_move_winner('top','right')
'c'
sage: p.rauzy_move_winner('bottom','right')
'e'
sage: p.rauzy_move_winner('top','left')
'a'
sage: p.rauzy_move_winner('bottom','left')
'd'
```

class `sage.dynamics.interval_exchanges.labelled.`**`LabelledPermutationIET`**(*intervals=None*, *alphabet=None*)

    Bases: `sage.dynamics.interval_exchanges.labelled.LabelledPermutation`, `sage.dynamics.interval_exchanges.template.PermutationIET`

    Labelled permutation for iet

    EXAMPLES:

    Reducibility testing:

```
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.is_irreducible()
True
```

```
sage: q = iet.Permutation('a b c d', 'b a d c')
sage: q.is_irreducible()
False
```

    Rauzy movability and Rauzy move:

```
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.has_rauzy_move('top')
True
sage: print p.rauzy_move('bottom')
a c b
c b a
sage: p.has_rauzy_move('top')
True
sage: print p.rauzy_move('top')
a b c
c a b
```

    Rauzy diagram:

```
sage: p = iet.Permutation('a b c', 'c b a')
sage: d = p.rauzy_diagram()
sage: p in d
True
```

    `has_rauzy_move`(*winner=None*, *side=None*)

        Returns `True` if you can perform a Rauzy move.

        INPUT:

            •`winner` - the winner interval ('top' or 'bottom')

            •`side` - (default: 'right') the side ('left' or 'right')

        OUTPUT:

        bool – `True` if self has a Rauzy move

        EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: p.has_rauzy_move()
True
```

```
sage: p = iet.Permutation('a b c','b a c')
sage: p.has_rauzy_move()
False
```

`is_identity``()`

 Returns True if self is the identity.

 OUTPUT:

 bool – True if self corresponds to the identity

 EXAMPLES:
```
sage: iet.Permutation("a b","a b").is_identity()
True
sage: iet.Permutation("a b","b a").is_identity()
False
```

`rauzy_diagram`(***args*)

 Returns the associated Rauzy diagram.

 For more information try help(iet.RauzyDiagram).

 OUTPUT:

 Rauzy diagram – the Rauzy diagram of the permutation

 EXAMPLES:
```
sage: p = iet.Permutation('a b c', 'c b a')
sage: d = p.rauzy_diagram()
```

`rauzy_move`(*winner=None*, *side=None*, *iteration=1*)

 Returns the Rauzy move.

 INPUT:

 • `winner` - the winner interval ('top' or 'bottom')

 • `side` - (default: 'right') the side ('left' or 'right')

 OUTPUT:

 permutation – the Rauzy move of the permutation

 EXAMPLES:
```
sage: p = iet.Permutation('a b','b a')
sage: p.rauzy_move('t','right')
a b
b a
sage: p.rauzy_move('b','right')
a b
b a

sage: p = iet.Permutation('a b c','c b a')
sage: p.rauzy_move('t','right')
a b c
c a b
sage: p.rauzy_move('b','right')
a c b
c b a

sage: p = iet.Permutation('a b','b a')
sage: p.rauzy_move('t','left')
a b
b a
sage: p.rauzy_move('b','left')
```

```
a b
b a

sage: p = iet.Permutation('a b c','c b a')
sage: p.rauzy_move('t','left')
a b c
b c a
sage: p.rauzy_move('b','left')
b a c
c b a
```

rauzy_move_interval_substitution(*winner=None*, *side=None*)
    Returns the interval substitution associated.

    INPUT:

        •`winner` - the winner interval ('top' or 'bottom')

        •`side` - (default: 'right') the side ('left' or 'right')

    OUTPUT:

    WordMorphism – a substitution on the alphabet of the permutation

    EXAMPLES:
    ```
    sage: p = iet.Permutation('a b','b a')
    sage: p.rauzy_move_interval_substitution('top','right')
    WordMorphism: a->a, b->ba
    sage: p.rauzy_move_interval_substitution('bottom','right')
    WordMorphism: a->ab, b->b
    sage: p.rauzy_move_interval_substitution('top','left')
    WordMorphism: a->ba, b->b
    sage: p.rauzy_move_interval_substitution('bottom','left')
    WordMorphism: a->a, b->ab
    ```

rauzy_move_orbit_substitution(*winner=None*, *side=None*)
    Return the action of the rauzy_move on the orbit.

    INPUT:

        •`i` - integer

        •`winner` - the winner interval ('top' or 'bottom')

        •`side` - (default: 'right') the side ('right' or 'left')

    OUTPUT:

    WordMorphism – a substitution on the alphabet of self

    EXAMPLES:
    ```
    sage: p = iet.Permutation('a b','b a')
    sage: p.rauzy_move_orbit_substitution('top','right')
    WordMorphism: a->ab, b->b
    sage: p.rauzy_move_orbit_substitution('bottom','right')
    WordMorphism: a->a, b->ab
    sage: p.rauzy_move_orbit_substitution('top','left')
    WordMorphism: a->a, b->ba
    sage: p.rauzy_move_orbit_substitution('bottom','left')
    WordMorphism: a->ba, b->b
    ```

reduced()
> Returns the associated reduced abelian permutation.
>
> OUTPUT:
>
> a reduced permutation – the underlying reduced permutation
>
> EXAMPLES:
> ```
> sage: p = iet.Permutation("a b c d","d c a b")
> sage: q = iet.Permutation("a b c d","d c a b",reduced=True)
> sage: p.reduced() == q
> True
> ```

class sage.dynamics.interval_exchanges.labelled.**LabelledPermutationLI** (*intervals=None*,
alpha-
bet=None)

> Bases: sage.dynamics.interval_exchanges.labelled.LabelledPermutation,
> sage.dynamics.interval_exchanges.template.PermutationLI
>
> Labelled quadratic (or generalized) permutation
>
> EXAMPLES:
>
> Reducibility testing:
> ```
> sage: p = iet.GeneralizedPermutation('a b b', 'c c a')
> sage: p.is_irreducible()
> True
> ```
>
> Reducibility testing with associated decomposition:
> ```
> sage: p = iet.GeneralizedPermutation('a b c a', 'b d d c')
> sage: p.is_irreducible()
> False
> sage: test, decomposition = p.is_irreducible(return_decomposition = True)
> sage: print test
> False
> sage: print decomposition
> (['a'], ['c', 'a'], [], ['c'])
> ```
>
> Rauzy movability and Rauzy move:
> ```
> sage: p = iet.GeneralizedPermutation('a a b b c c', 'd d')
> sage: p.has_rauzy_move(0)
> False
> sage: p.has_rauzy_move(1)
> True
> sage: q = p.rauzy_move(1)
> sage: print q
> a a b b c
> c d d
> sage: q.has_rauzy_move(0)
> True
> sage: q.has_rauzy_move(1)
> True
> ```
>
> Rauzy diagrams:
> ```
> sage: p = iet.GeneralizedPermutation('0 0 1 1','2 2')
> sage: r = p.rauzy_diagram()
> sage: p in r
> True
> ```

has_right_rauzy_move(*winner*)
Test of Rauzy movability with a specified winner

A quadratic (or generalized) permutation is rauzy_movable type depending on the possible length of the last interval. It is dependent of the length equation.

INPUT:

   •`winner` - 'top' (or 't' or 0) or 'bottom' (or 'b' or 1)

OUTPUT:

bool – `True` if self has a Rauzy move

EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a','b b')
sage: p.has_right_rauzy_move('top')
False
sage: p.has_right_rauzy_move('bottom')
False

sage: p = iet.GeneralizedPermutation('a a b','b c c')
sage: p.has_right_rauzy_move('top')
True
sage: p.has_right_rauzy_move('bottom')
True

sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: p.has_right_rauzy_move('top')
True
sage: p.has_right_rauzy_move('bottom')
False

sage: p = iet.GeneralizedPermutation('a a b b','c c')
sage: p.has_right_rauzy_move('top')
False
sage: p.has_right_rauzy_move('bottom')
True
```

left_rauzy_move(*winner*)
Perform a Rauzy move on the left.

INPUT:

   •`winner` - 'top' or 'bottom'

OUTPUT:

permutation – the Rauzy move of self

EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a b','b c c')
sage: p.left_rauzy_move(0)
a a b b
c c
sage: p.left_rauzy_move(1)
a a b
b c c

sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.left_rauzy_move(0)
a b b
```

```
c c a
sage: p.left_rauzy_move(1)
b b
c c a a
```

TESTS:
```
sage: p = iet.GeneralizedPermutation('a a b','b c c')
sage: q = p.top_bottom_inverse()
sage: q = q.left_rauzy_move(0)
sage: q = q.top_bottom_inverse()
sage: q == p.left_rauzy_move(1)
True
sage: q = p.top_bottom_inverse()
sage: q = q.left_rauzy_move(1)
sage: q = q.top_bottom_inverse()
sage: q == p.left_rauzy_move(0)
True
sage: q = p.left_right_inverse()
sage: q = q.right_rauzy_move(0)
sage: q = q.left_right_inverse()
sage: q == p.left_rauzy_move(0)
True
sage: q = p.left_right_inverse()
sage: q = q.right_rauzy_move(1)
sage: q = q.left_right_inverse()
sage: q == p.left_rauzy_move(1)
True
```

rauzy_diagram(***kargs*)
    Returns the associated RauzyDiagram.

    OUTPUT:

    Rauzy diagram – the Rauzy diagram of the permutation

    EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a b c b', 'c d d a')
sage: d = p.rauzy_diagram()
sage: p in d
True
```

    For more information, try help(iet.RauzyDiagram)

reduced()
    Returns the associated reduced quadratic permutations.

    OUTPUT:

    permutation – the underlying reduced permutation

    EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: q = p.reduced()
sage: q
a a
b b c c
sage: p.rauzy_move(0).reduced() == q.rauzy_move(0)
True
```

right_rauzy_move(*winner*)

>   Perform a Rauzy move on the right (the standard one).

>   INPUT:

>>   •`winner` - 'top' (or 't' or 0) or 'bottom' (or 'b' or 1)

>   OUTPUT:

>   boolean – `True` if self has a Rauzy move

>   EXAMPLES:

```
sage: p = iet.GeneralizedPermutation('a a b','b c c')
sage: p.right_rauzy_move(0)
a a b
b c c
sage: p.right_rauzy_move(1)
a a
b b c c

sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.right_rauzy_move(0)
a a b b
c c
sage: p.right_rauzy_move(1)
a b b
c c a
```

>   TESTS:

```
sage: p = iet.GeneralizedPermutation('a a b','b c c')
sage: q = p.top_bottom_inverse()
sage: q = q.right_rauzy_move(0)
sage: q = q.top_bottom_inverse()
sage: q == p.right_rauzy_move(1)
True
sage: q = p.top_bottom_inverse()
sage: q = q.right_rauzy_move(1)
sage: q = q.top_bottom_inverse()
sage: q == p.right_rauzy_move(0)
True
sage: p = p.left_right_inverse()
sage: q = q.left_rauzy_move(0)
sage: q = q.left_right_inverse()
sage: q == p.right_rauzy_move(0)
True
sage: q = p.left_right_inverse()
sage: q = q.left_rauzy_move(1)
sage: q = q.left_right_inverse()
sage: q == p.right_rauzy_move(1)
True
```

sage.dynamics.interval_exchanges.labelled.**LabelledPermutationsIET_iterator**(*nintervals=None*, *irreducible=True*, *alphabet=None*)

>   Returns an iterator over labelled permutations.

INPUT:

- `nintervals` - integer or `None`

- `irreducible` - boolean (default: `True`)

- `alphabet` - something that should be converted to an alphabet of at least nintervals letters

OUTPUT:

iterator – an iterator over permutations

TESTS:
```
sage: for p in iet.Permutations_iterator(2, alphabet="ab"):
....:     print p, "\n****"   #indirect doctest
a b
b a
****
b a
a b
****
sage: for p in iet.Permutations_iterator(3, alphabet="abc"):
....:     print p, "\n*****"   #indirect doctest
a b c
b c a
*****
a b c
c a b
*****
a b c
c b a
*****
a c b
b a c
*****
a c b
b c a
*****
a c b
c b a
*****
b a c
a c b
*****
b a c
c a b
*****
b a c
c b a
*****
b c a
a b c
*****
b c a
a c b
*****
b c a
c a b
*****
c a b
a b c
```

```
*****
c a b
b a c
*****
c a b
b c a
*****
c b a
a b c
*****
c b a
a c b
*****
c b a
b a c
*****
```

class sage.dynamics.interval_exchanges.labelled.**LabelledRauzyDiagram**(*p*, *right_induction=True*, *left_induction=False*, *left_right_inversion=False*, *top_bottom_inversion=False*, *symmetric=False*)

Bases: sage.dynamics.interval_exchanges.template.RauzyDiagram

Template for Rauzy diagrams of labelled permutations.

> **Warning:** DO NOT USE

class Path(*parent*, **data*)

Bases: sage.dynamics.interval_exchanges.template.RauzyDiagram.Path

Path in Labelled Rauzy diagram.

dual_substitution()

Returns the substitution of intervals obtained.

OUTPUT:

WordMorphism – the word morphism corresponding to the interval

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: p0 = r.path(p,0)
sage: s0 = p0.interval_substitution()
sage: s0
WordMorphism: a->a, b->ba
sage: p1 = r.path(p,1)
sage: s1 = p1.interval_substitution()
sage: s1
WordMorphism: a->ab, b->b
sage: (p0 + p1).interval_substitution() == s1 * s0
True
sage: (p1 + p0).interval_substitution() == s0 * s1
True
```

interval_substitution()
>   Returns the substitution of intervals obtained.
>
>   OUTPUT:
>
>   WordMorphism – the word morphism corresponding to the interval
>
>   EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: p0 = r.path(p,0)
sage: s0 = p0.interval_substitution()
sage: s0
WordMorphism: a->a, b->ba
sage: p1 = r.path(p,1)
sage: s1 = p1.interval_substitution()
sage: s1
WordMorphism: a->ab, b->b
sage: (p0 + p1).interval_substitution() == s1 * s0
True
sage: (p1 + p0).interval_substitution() == s0 * s1
True
```

is_full()
>   Tests the fullness.
>
>   A path is full if all intervals win at least one time.
>
>   OUTPUT:
>
>   boolean – `True` if the path is full and `False` else
>
>   EXAMPLE:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g0 = r.path(p,'t','b','t')
sage: g1 = r.path(p,'b','t','b')
sage: g0.is_full()
False
sage: g1.is_full()
False
sage: (g0 + g1).is_full()
True
sage: (g1 + g0).is_full()
True
```

matrix()
>   Returns the matrix associated to a path.
>
>   The matrix associated to a Rauzy induction, is the linear application that allows to recover the lengths of `self` from the lengths of the induced.
>
>   OUTPUT:
>
>   matrix – a square matrix of integers
>
>   EXAMPLES:

```
sage: p = iet.Permutation('a1 a2','a2 a1')
sage: d = p.rauzy_diagram()
sage: g = d.path(p,'top')
sage: g.matrix()
[1 0]
```

```
[1 1]
sage: g = d.path(p,'bottom')
sage: g.matrix()
[1 1]
[0 1]
sage: p = iet.Permutation('a b c','c b a')
sage: d = p.rauzy_diagram()
sage: g = d.path(p)
sage: g.matrix() == identity_matrix(3)
True
sage: g = d.path(p,'top')
sage: g.matrix()
[1 0 0]
[0 1 0]
[1 0 1]
sage: g = d.path(p,'bottom')
sage: g.matrix()
[1 0 1]
[0 1 0]
[0 0 1]
```

orbit_substitution()
   Returns the substitution on the orbit of the left extremity.

   OUTPUT:

   WordMorhpism – the word morphism corresponding to the orbit

   EXAMPLES:
```
sage: p = iet.Permutation('a b','b a')
sage: d = p.rauzy_diagram()
sage: g0 = d.path(p,'top')
sage: s0 = g0.orbit_substitution()
sage: s0
WordMorphism: a->ab, b->b
sage: g1 = d.path(p,'bottom')
sage: s1 = g1.orbit_substitution()
sage: s1
WordMorphism: a->a, b->ab
sage: (g0 + g1).orbit_substitution() == s0 * s1
True
sage: (g1 + g0).orbit_substitution() == s1 * s0
True
```

substitution()
   Returns the substitution on the orbit of the left extremity.

   OUTPUT:

   WordMorhpism – the word morphism corresponding to the orbit

   EXAMPLES:
```
sage: p = iet.Permutation('a b','b a')
sage: d = p.rauzy_diagram()
sage: g0 = d.path(p,'top')
sage: s0 = g0.orbit_substitution()
sage: s0
WordMorphism: a->ab, b->b
sage: g1 = d.path(p,'bottom')
sage: s1 = g1.orbit_substitution()
```

```
sage: s1
WordMorphism: a->a, b->ab
sage: (g0 + g1).orbit_substitution() == s0 * s1
True
sage: (g1 + g0).orbit_substitution() == s1 * s0
True
```

LabelledRauzyDiagram.**edge_to_interval_substitution**(*p=None*, *edge_type=None*)
    Returns the interval substitution associated to an edge

    OUTPUT:

    WordMorphism – the WordMorphism corresponding to the edge

    EXAMPLE:
```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: r.edge_to_interval_substitution(None,None)
WordMorphism: a->a, b->b, c->c
sage: r.edge_to_interval_substitution(p,0)
WordMorphism: a->a, b->b, c->ca
sage: r.edge_to_interval_substitution(p,1)
WordMorphism: a->ac, b->b, c->c
```

LabelledRauzyDiagram.**edge_to_orbit_substitution**(*p=None*, *edge_type=None*)
    Returns the interval substitution associated to an edge

    OUTPUT:

    WordMorphism – the word morphism corresponding to the edge

    EXAMPLE:
```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: r.edge_to_orbit_substitution(None,None)
WordMorphism: a->a, b->b, c->c
sage: r.edge_to_orbit_substitution(p,0)
WordMorphism: a->ac, b->b, c->c
sage: r.edge_to_orbit_substitution(p,1)
WordMorphism: a->a, b->b, c->ac
```

LabelledRauzyDiagram.**full_loop_iterator**(*start=None*, *max_length=1*)
    Returns an iterator over all full path starting at start.

    INPUT:

        •`start` - the start point

        •`max_length` - a limit on the length of the paths

    OUTPUT:

    iterator – iterator over full loops

    EXAMPLE:
```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: for g in r.full_loop_iterator(p,2):
....:     print g.matrix(), "\n*****"
[1 1]
[1 2]
```

```
     *****
     [2 1]
     [1 1]
     *****
```

LabelledRauzyDiagram.**full_nloop_iterator**(*start=None*, *length=1*)
Returns an iterator over all full loops of given length.

INPUT:

- `start` - the initial permutation

- `length` - the length to consider

OUTPUT:

iterator – an iterator over the full loops of given length

EXAMPLE:
```
sage: p = iet.Permutation('a b','b a')
sage: d = p.rauzy_diagram()
sage: for g in d.full_nloop_iterator(p,2):
....:     print g.matrix(), "\n*****"
[1 1]
[1 2]
*****
[2 1]
[1 1]
*****
```

1.3 Reduced permutations

A reduced (generalized) permutation is better suited to study strata of Abelian (or quadratic) holomorphic forms on Riemann surfaces. The Rauzy diagram is an invariant of such a component. Corentin Boissy proved the identification of Rauzy diagrams with connected components of stratas. But the geometry of the diagram and the relation with the strata is not yet totally understood.

AUTHORS:

- Vincent Delecroix (2000-09-29): initial version

TESTS:

```
sage: from sage.dynamics.interval_exchanges.reduced import ReducedPermutationIET
sage: ReducedPermutationIET([['a','b'],['b','a']])
a b
b a
sage: ReducedPermutationIET([[1,2,3],[3,1,2]])
1 2 3
3 1 2
sage: from sage.dynamics.interval_exchanges.reduced import ReducedPermutationLI
sage: ReducedPermutationLI([[1,1],[2,2,3,3,4,4]])
1 1
2 2 3 3 4 4
sage: ReducedPermutationLI([['a','a','b','b','c','c'],['d','d']])
a a b b c c
d d
sage: from sage.dynamics.interval_exchanges.reduced import FlippedReducedPermutationIET
```

```
sage: FlippedReducedPermutationIET([[1,2,3],[3,2,1]],flips=[1,2])
-1 -2  3
 3 -2 -1
sage: FlippedReducedPermutationIET([['a','b','c'],['b','c','a']],flips='b')
 a -b  c
-b  c  a
sage: from sage.dynamics.interval_exchanges.reduced import FlippedReducedPermutationLI
sage: FlippedReducedPermutationLI([[1,1],[2,2,3,3,4,4]], flips=[1,4])
-1 -1
 2  2  3  3 -4 -4
sage: FlippedReducedPermutationLI([['a','a','b','b'],['c','c']],flips='ac')
-a -a  b  b
-c -c
sage: from sage.dynamics.interval_exchanges.reduced import ReducedRauzyDiagram
sage: p = ReducedPermutationIET([[1,2,3],[3,2,1]])
sage: d = ReducedRauzyDiagram(p)
```

class sage.dynamics.interval_exchanges.reduced.**FlippedReducedPermutation**(*intervals=None*, *flips=None*, *alphabet=None*)

> Bases: `sage.dynamics.interval_exchanges.reduced.ReducedPermutation`
>
> Flipped Reduced Permutation.
>
> > **Warning:** Internal class! Do not use directly!
>
> INPUT:
>
> > •`intervals` - a list of two lists
> >
> > •`flips` - the flipped letters
> >
> > •`alphabet` - an alphabet
>
> **right_rauzy_move**(*winner*)
> > Performs a Rauzy move on the right.
> >
> > EXAMPLE:
> > ```
> > sage: p = iet.Permutation('a b c','c b a',reduced=True,flips='c')
> > sage: p.right_rauzy_move('top')
> > -a   b -c
> > -a  -c   b
> > ```

class sage.dynamics.interval_exchanges.reduced.**FlippedReducedPermutationIET**(*intervals=None*, *flips=None*, *alphabet=None*)

> Bases: `sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutation`,
> `sage.dynamics.interval_exchanges.template.FlippedPermutationIET`,
> `sage.dynamics.interval_exchanges.reduced.ReducedPermutationIET`
>
> Flipped Reduced Permutation from iet
>
> EXAMPLES

```
sage: p = iet.Permutation('a b c', 'c b a', flips=['a'], reduced=True)
sage: p.rauzy_move(1)
-a -b  c
-a  c -b
```

TESTS:
```
sage: p = iet.Permutation('a b','b a',flips=['a'])
sage: p == loads(dumps(p))
True
```

list (*flips=False*)

Returns a list representation of self.

INPUT:

- **flips** - boolean (default: False) if True the output contains 2-uple of (label, flip)

EXAMPLES:

:: sage: p = iet.Permutation('a b','b a',reduced=True,flips='b') sage: p.list(flips=True) [[('a', 1), ('b', -1)], [('b', -1), ('a', 1)]] sage: p.list(flips=False) [['a', 'b'], ['b', 'a']] sage: p.alphabet([0,1]) sage: p.list(flips=True) [[(0, 1), (1, -1)], [(1, -1), (0, 1)]] sage: p.list(flips=False) [[0, 1], [1, 0]]

One can recover the initial permutation from this list:
```
sage: p = iet.Permutation('a b','b a',reduced=True,flips='a')
sage: iet.Permutation(p.list(), flips=p.flips(), reduced=True) == p
True
```

rauzy_diagram (***kargs*)

Returns the associated Rauzy diagram.

EXAMPLES:
```
sage: p = iet.Permutation('a b','b a',reduced=True,flips='a')
sage: r = p.rauzy_diagram()
sage: p in r
True
```

class sage.dynamics.interval_exchanges.reduced.**FlippedReducedPermutationLI** (*intervals=None*, *flips=None*, *alphabet=None*)

Bases: sage.dynamics.interval_exchanges.reduced.FlippedReducedPermutation, sage.dynamics.interval_exchanges.template.FlippedPermutationLI, sage.dynamics.interval_exchanges.reduced.ReducedPermutationLI

Flipped Reduced Permutation from li

EXAMPLES:

Creation using the GeneralizedPermutation function:
```
sage: p = iet.GeneralizedPermutation('a a b', 'b c c', reduced=True, flips='a')
```

list (*flips=False*)

Returns a list representation of self.

INPUT:

- flips - boolean (default: False) return the list with flips

---

EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a','b b',reduced=True,flips='a')
sage: p.list(flips=True)
[[('a', -1), ('a', -1)], [('b', 1), ('b', 1)]]
sage: p.list(flips=False)
[['a', 'a'], ['b', 'b']]

sage: p = iet.GeneralizedPermutation('a a b','b c c',reduced=True,flips='abc')
sage: p.list(flips=True)
[[('a', -1), ('a', -1), ('b', -1)], [('b', -1), ('c', -1), ('c', -1)]]
sage: p.list(flips=False)
[['a', 'a', 'b'], ['b', 'c', 'c']]
```

one can rebuild the permutation from the list:
```
sage: p = iet.GeneralizedPermutation('a a b','b c c',flips='a',reduced=True)
sage: iet.GeneralizedPermutation(p.list(),flips=p.flips(),reduced=True) == p
True
```

rauzy_diagram(***kargs*)
    Returns the associated Rauzy diagram.

    For more explanation and a list of arguments try help(iet.RauzyDiagram)

    EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a b','c c b',reduced=True)
sage: r = p.rauzy_diagram()
sage: p in r
True
```

class sage.dynamics.interval_exchanges.reduced.**FlippedReducedRauzyDiagram**(*p,*
                                                                            right_induction=True,
                                                                            left_induction=False,
                                                                            left_right_inversion=False,
                                                                            top_bottom_inversion=False,
                                                                            sym-
                                                                            met-
                                                                            ric=False)
    Bases: sage.dynamics.interval_exchanges.template.FlippedRauzyDiagram,
    sage.dynamics.interval_exchanges.reduced.ReducedRauzyDiagram

    Rauzy diagram of flipped reduced permutations.

class sage.dynamics.interval_exchanges.reduced.**ReducedPermutation**(*intervals=None,*
                                                                    alpha-
                                                                    bet=None)
    Bases: sage.structure.sage_object.SageObject

    Template for reduced objects.

    > **Warning:** Internal class! Do not use directly!

    INPUT:

    • `intervals` - a list of two list of labels

    • `alphabet` - (default: None) any object that can be used to initialize an Alphabet or None. In this latter
      case, the letter of the intervals are used to generate one.

erase_letter(*letter*)
    Erases a letter.

    INPUT:

        •`letter` - a letter which is a label of an interval of self

    EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.erase_letter('a')
b c
c b

sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.erase_letter('a')
b b
c c
```

left_rauzy_move(*winner*)
    Performs a Rauzy move on the left.

    EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: p.left_rauzy_move(0)
a b c
b c a
sage: p.right_rauzy_move(1)
a b c
b c a

sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
sage: p.left_rauzy_move(0)
a a b
b c c
```

length(*interval=None*)
    Returns the 2-uple of lengths.

    p.length() is identical to (p.length_top(), p.length_bottom()) If an interval is specified, it returns the length of the specified interval.

    INPUT:

        •`interval` - None, 'top' (or 't' or 0) or 'bottom' (or 'b' or 1)

    OUTPUT:

    integer or 2-uple of integers – the corresponding lengths

    EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.length()
(3, 3)
sage: p = iet.GeneralizedPermutation('a a b','c d c b d')
sage: p.length()
(3, 5)
```

length_bottom()
    Returns the number of intervals in the bottom segment.

    OUTPUT:

integer – the length of the bottom segment

EXAMPLES:
```
sage: p = iet.Permutation('a b c','c b a')
sage: p.length_bottom()
3
sage: p = iet.GeneralizedPermutation('a a b','c d c b d')
sage: p.length_bottom()
5
```

length_top()
   Returns the number of intervals in the top segment.

   OUTPUT:

   integer – the length of the top segment

   EXAMPLES:
```
sage: p = iet.Permutation('a b c','c b a')
sage: p.length_top()
3
sage: p = iet.GeneralizedPermutation('a a b','c d c b d')
sage: p.length_top()
3
sage: p = iet.GeneralizedPermutation('a b c b d c d', 'e a e')
sage: p.length_top()
7
```

right_rauzy_move(*winner*)
   Performs a Rauzy move on the right.

   EXAMPLES:
```
sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: p.right_rauzy_move(0)
a b c
c a b
sage: p.right_rauzy_move(1)
a b c
b c a

sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
sage: p.right_rauzy_move(0)
a b b
c c a
```

class sage.dynamics.interval_exchanges.reduced.**ReducedPermutationIET**(*intervals=None*, *alphabet=None*)

   Bases: sage.dynamics.interval_exchanges.reduced.ReducedPermutation, sage.dynamics.interval_exchanges.template.PermutationIET

   Reduced permutation from iet

   Permutation from iet without numerotation of intervals. For initialization, you should use GeneralizedPermutation which is the class factory for all permutation types.

   EXAMPLES:

   Equality testing (no equality of letters but just of ordering):

```
sage: p = iet.Permutation('a b c', 'c b a', reduced = True)
sage: q = iet.Permutation('p q r', 'r q p', reduced = True)
sage: p == q
True
```

Reducibility testing:
```
sage: p = iet.Permutation('a b c', 'c b a', reduced = True)
sage: p.is_irreducible()
True
```

```
sage: q = iet.Permutation('a b c d', 'b a d c', reduced = True)
sage: q.is_irreducible()
False
```

Rauzy movability and Rauzy move:
```
sage: p = iet.Permutation('a b c', 'c b a', reduced = True)
sage: p.has_rauzy_move(1)
True
sage: print p.rauzy_move(1)
a b c
b c a
```

Rauzy diagrams:
```
sage: p = iet.Permutation('a b c d', 'd a b c')
sage: p_red = iet.Permutation('a b c d', 'd a b c', reduced = True)
sage: d = p.rauzy_diagram()
sage: d_red = p_red.rauzy_diagram()
sage: p.rauzy_move(0) in d
True
sage: print d.cardinality(), d_red.cardinality()
12 6
```

has_rauzy_move (*winner*, *side='right'*)

    Tests if the permutation is rauzy_movable on the left.

    EXAMPLES:
```
sage: p = iet.Permutation('a b c','a c b',reduced=True)
sage: p.has_rauzy_move(0,'right')
True
sage: p.has_rauzy_move(0,'left')
False
sage: p.has_rauzy_move(1,'right')
True
sage: p.has_rauzy_move(1,'left')
False
```

```
sage: p = iet.Permutation('a b c d','c a b d',reduced=True)
sage: p.has_rauzy_move(0,'right')
False
sage: p.has_rauzy_move(0,'left')
True
sage: p.has_rauzy_move(1,'right')
False
sage: p.has_rauzy_move(1,'left')
True
```

is_identity()
> Returns True if self is the identity.
>
> EXAMPLES:
> ```
> sage: iet.Permutation("a b","a b",reduced=True).is_identity()
> True
> sage: iet.Permutation("a b","b a",reduced=True).is_identity()
> False
> ```

list()
> Returns a list of two list that represents the permutation.
>
> EXAMPLES:
> ```
> sage: p = iet.GeneralizedPermutation('a b','b a',reduced=True)
> sage: p.list() == [p[0], p[1]]
> True
> sage: p.list() == [['a', 'b'], ['b', 'a']]
> True
>
> sage: p = iet.GeneralizedPermutation('a b c', 'b c a',reduced=True)
> sage: iet.GeneralizedPermutation(p.list(),reduced=True) == p
> True
> ```

rauzy_diagram(***kargs*)
> Returns the associated Rauzy diagram.
>
> OUTPUT:
>
> A Rauzy diagram
>
> EXAMPLES:
> ```
> sage: p = iet.Permutation('a b c d', 'd a b c',reduced=True)
> sage: d = p.rauzy_diagram()
> sage: p.rauzy_move(0) in d
> True
> sage: p.rauzy_move(1) in d
> True
> ```
>
> For more information, try help RauzyDiagram

rauzy_move_relabel(*winner*, *side='right'*)
> Returns the relabelization obtained from this move.
>
> EXAMPLE:
> ```
> sage: p = iet.Permutation('a b c d','d c b a')
> sage: q = p.reduced()
> sage: p_t = p.rauzy_move('t')
> sage: q_t = q.rauzy_move('t')
> sage: s_t = q.rauzy_move_relabel('t')
> sage: s_t
> WordMorphism: a->a, b->b, c->c, d->d
> sage: map(s_t, p_t[0]) == map(Word, q_t[0])
> True
> sage: map(s_t, p_t[1]) == map(Word, q_t[1])
> True
> sage: p_b = p.rauzy_move('b')
> sage: q_b = q.rauzy_move('b')
> sage: s_b = q.rauzy_move_relabel('b')
> sage: s_b
> ```

```
WordMorphism: a->a, b->d, c->b, d->c
sage: map(s_b, q_b[0]) == map(Word, p_b[0])
True
sage: map(s_b, q_b[1]) == map(Word, p_b[1])
True
```

class sage.dynamics.interval_exchanges.reduced.**ReducedPermutationLI**(*intervals=None*, *alphabet=None*)

    Bases: sage.dynamics.interval_exchanges.reduced.ReducedPermutation, sage.dynamics.interval_exchanges.template.PermutationLI

    Reduced quadratic (or generalized) permutation.

    EXAMPLES:

    Reducibility testing:
```
sage: p = iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
sage: p.is_irreducible()
True

sage: p = iet.GeneralizedPermutation('a b c a', 'b d d c', reduced = True)
sage: p.is_irreducible()
False
sage: test, decomposition = p.is_irreducible(return_decomposition = True)
sage: test
False
sage: decomposition
(['a'], ['c', 'a'], [], ['c'])
```

    Rauzy movavability and Rauzy move:
```
sage: p = iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
sage: p.has_rauzy_move(0)
True
sage: p.rauzy_move(0)
a a b b
c c
sage: p.rauzy_move(0).has_rauzy_move(0)
False
sage: p.rauzy_move(1)
a b b
c c a
```

    Rauzy diagrams:
```
sage: p_red = iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
sage: d_red = p_red.rauzy_diagram()
sage: d_red.cardinality()
4
```

    list()
        The permutations as a list of two lists.

        EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a b b', 'c c a', reduced = True)
sage: list(p)
[['a', 'b', 'b'], ['c', 'c', 'a']]
```

rauzy_diagram(***kargs*)

> Returns the associated Rauzy diagram.
>
> The Rauzy diagram of a permutation corresponds to all permutations that we could obtain from this one by Rauzy move. The set obtained is a labelled Graph. The label of vertices being 0 or 1 depending on the type.
>
> OUTPUT:
>
> Rauzy diagram – the graph of permutations obtained by rauzy induction
>
> EXAMPLES:
> ```
> sage: p = iet.Permutation('a b c d', 'd a b c')
> sage: d = p.rauzy_diagram()
> ```

sage.dynamics.interval_exchanges.reduced.**ReducedPermutationsIET_iterator**(*nintervals=None*, *irreducible=True*, *alphabet=None*)

> Returns an iterator over reduced permutations
>
> INPUT:
>
> - `nintervals` - integer or None
>
> - `irreducible` - boolean
>
> - `alphabet` - something that should be converted to an alphabet of at least nintervals letters
>
> TESTS:
> ```
> sage: for p in iet.Permutations_iterator(3,reduced=True,alphabet="abc"):
> ....:     print p  #indirect doctest
> a b c
> b c a
> a b c
> c a b
> a b c
> c b a
> ```

class sage.dynamics.interval_exchanges.reduced.**ReducedRauzyDiagram**(*p*, *right_induction=True*, *left_induction=False*, *left_right_inversion=False*, *top_bottom_inversion=False*, *symmetric=False*)

> Bases: `sage.dynamics.interval_exchanges.template.RauzyDiagram`
>
> Rauzy diagram of reduced permutations

sage.dynamics.interval_exchanges.reduced.**alphabetized_atwin**(*twin*, *alphabet*)

> Alphabetization of a twin of iet.
>
> TESTS:
> ```
> sage: from sage.dynamics.interval_exchanges.reduced import alphabetized_atwin
> ```

```
sage: twin = [[0,1],[0,1]]
sage: alphabet = Alphabet("ab")
sage: alphabetized_atwin(twin, alphabet)
[['a', 'b'], ['a', 'b']]

sage: twin = [[1,0],[1,0]]
sage: alphabet = Alphabet([0,1])
sage: alphabetized_atwin(twin, alphabet)
[[0, 1], [1, 0]]

sage: twin = [[1,2,3,0],[3,0,1,2]]
sage: alphabet = Alphabet("abcd")
sage: alphabetized_atwin(twin,alphabet)
[['a', 'b', 'c', 'd'], ['d', 'a', 'b', 'c']]
```

sage.dynamics.interval_exchanges.reduced.**alphabetized_qtwin**(*twin*, *alphabet*)
   Alphabetization of a qtwin.

   TESTS:
```
sage: from sage.dynamics.interval_exchanges.reduced import alphabetized_qtwin

sage: twin = [[(1,0),(1,1)],[(0,0),(0,1)]]
sage: alphabet = Alphabet("ab")
sage: print alphabetized_qtwin(twin,alphabet)
[['a', 'b'], ['a', 'b']]

sage: twin = [[(1,1), (1,0)],[(0,1), (0,0)]]
sage: alphabet=Alphabet("AB")
sage: alphabetized_qtwin(twin,alphabet)
[['A', 'B'], ['B', 'A']]
sage: alphabet=Alphabet("BA")
sage: alphabetized_qtwin(twin,alphabet)
[['B', 'A'], ['A', 'B']]

sage: twin = [[(0,1),(0,0)],[(1,1),(1,0)]]
sage: alphabet=Alphabet("ab")
sage: print alphabetized_qtwin(twin,alphabet)
[['a', 'a'], ['b', 'b']]

sage: twin = [[(0,2),(1,1),(0,0)],[(1,2),(0,1),(1,0)]]
sage: alphabet=Alphabet("abc")
sage: print alphabetized_qtwin(twin,alphabet)
[['a', 'b', 'a'], ['c', 'b', 'c']]
```

sage.dynamics.interval_exchanges.reduced.**labelize_flip**(*couple*)
   Returns a string from a 2-uple couple of the form (name, flip).

   TESTS:
```
sage: from sage.dynamics.interval_exchanges.reduced import labelize_flip
sage: labelize_flip((4,1))
' 4'
sage: labelize_flip(('a',-1))
'-a'
```

1.4 Permutations template

This file define high level operations on permutations (alphabet, the different rauzy induction, ...) shared by reduced and labeled permutations.

AUTHORS:

- Vincent Delecroix (2008-12-20): initial version

Todo

- construct as options different string representations for a permutation

    - the two intervals: str

    - the two intervals on one line: str_one_line

    - the separatrix diagram: str_separatrix_diagram

    - twin[0] and twin[1] for reduced permutation

    - nothing (useful for Rauzy diagram)

---

class sage.dynamics.interval_exchanges.template.**FlippedPermutation**
> Bases: `sage.dynamics.interval_exchanges.template.Permutation`

> Template for flipped generalized permutations.

> > **Warning:** Internal class! Do not use directly!

> AUTHORS:

> > •Vincent Delecroix (2008-12-20): initial version

> **str** (*sep='n'*)
> > String representation.

> > TESTS:
> > ```
> > sage: p = iet.GeneralizedPermutation('a a','b b',flips='a')
> > sage: print p.str()
> > -a -a
> >  b  b
> >  sage: print p.str('/')
> >  -a -a/ b  b
> > ```

class sage.dynamics.interval_exchanges.template.**FlippedPermutationIET**
> Bases: `sage.dynamics.interval_exchanges.template.FlippedPermutation`,
> `sage.dynamics.interval_exchanges.template.PermutationIET`

> Template for flipped Abelian permutations.

> > **Warning:** Internal class! Do not use directly!

> AUTHORS:

> > •Vincent Delecroix (2008-12-20): initial version

> **flips** ()
> > Returns the list of flips.

EXAMPLES:
```
sage: p = iet.Permutation('a b c','c b a',flips='ac')
sage: p.flips()
['a', 'c']
```

class sage.dynamics.interval_exchanges.template.**FlippedPermutationLI**
    Bases: sage.dynamics.interval_exchanges.template.FlippedPermutation,
    sage.dynamics.interval_exchanges.template.PermutationLI

    Template for flipped quadratic permutations.

> **Warning:** Internal class! Do not use directly!

    AUTHORS:

> •Vincent Delecroix (2008-12-20): initial version

    flips()
        Returns the list of flipped intervals.

        EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a','b b',flips='a')
sage: p.flips()
['a']
sage: p = iet.GeneralizedPermutation('a a','b b',flips='b',reduced=True)
sage: p.flips()
['b']
```

class sage.dynamics.interval_exchanges.template.**FlippedRauzyDiagram**(*p*,
                                                    right_induction=True,
                                                    left_induction=False,
                                                      left_right_inversion=False,
                                                      top_bottom_inversion=False,
                                                      symmetric=False)

    Bases: sage.dynamics.interval_exchanges.template.RauzyDiagram

    Template for flipped Rauzy diagrams.

    AUTHORS:

> •Vincent Delecroix (2009-09-29): initial version

    complete(*p*, *reducible=False*)
        Completion of the Rauzy diagram

        Add all successors of p for defined operations in edge_types. Could be used for generating non (strongly) connected Rauzy diagrams. Sometimes, for flipped permutations, the maximal connected graph in all permutations is not strongly connected. Finding such components needs to call most than once the .complete() method.

        INPUT:

> •p - a permutation

> •reducible - put or not reducible permutations

        EXAMPLES:
```
sage: p = iet.Permutation('a b c','c b a',flips='a')
sage: d = p.rauzy_diagram()
```

```
sage: d
Rauzy diagram with 3 permutations
sage: p = iet.Permutation('a b c','c b a',flips='b')
sage: d.complete(p)
sage: d
Rauzy diagram with 8 permutations
sage: p = iet.Permutation('a b c','c b a',flips='a')
sage: d.complete(p)
sage: d
Rauzy diagram with 8 permutations
```

class sage.dynamics.interval_exchanges.template.**Permutation**
    Bases: sage.structure.sage_object.SageObject

    Template for all permutations.

> **Warning:** Internal class! Do not use directly!

    This class implement generic algorithm (stratum, connected component, ...) and unfies all its children.

    alphabet (*data=None*)
        Manages the alphabet of self.

        If there is no argument, the method returns the alphabet used. If the argument could be converted to an alphabet, this alphabet will be used.

        INPUT:

            •data - None or something that could be converted to an alphabet

        OUTPUT:

        – either None or the current alphabet

        EXAMPLES:
```
sage: p = iet.Permutation('a b','a b')
sage: p.alphabet([0,1])
sage: p.alphabet() == Alphabet([0,1])
True
sage: p
0 1
0 1
sage: p.alphabet("cd")
sage: p.alphabet() == Alphabet(['c','d'])
True
sage: p
c d
c d
```

    has_rauzy_move (*winner='top'*, *side=None*)
        Tests the legality of a Rauzy move.

        INPUT:

            •winner - 'top' or 'bottom' corresponding to the interval

            •side - 'left' or 'right' (defaut)

        OUTPUT:

        – a boolean

EXAMPLES:

```
sage: p = iet.Permutation('a b','a b')
sage: p.has_rauzy_move('top','right')
False
sage: p.has_rauzy_move('bottom','right')
False
sage: p.has_rauzy_move('top','left')
False
sage: p.has_rauzy_move('bottom','left')
False

sage: p = iet.Permutation('a b c','b a c')
sage: p.has_rauzy_move('top','right')
False
sage: p.has_rauzy_move('bottom', 'right')
False
sage: p.has_rauzy_move('top','left')
True
sage: p.has_rauzy_move('bottom','left')
True

sage: p = iet.Permutation('a b','b a')
sage: p.has_rauzy_move('top','right')
True
sage: p.has_rauzy_move('bottom','right')
True
sage: p.has_rauzy_move('top','left')
True
sage: p.has_rauzy_move('bottom','left')
True
```

horizontal_inverse()

Returns the top-bottom inverse.

You can use also use the shorter .tb_inverse().

OUTPUT:

– a permutation

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: p.top_bottom_inverse()
b a
a b
sage: p = iet.Permutation('a b','b a',reduced=True)
sage: p.top_bottom_inverse() == p
True

sage: p = iet.Permutation('a b c d','c d a b')
sage: p.top_bottom_inverse()
c d a b
a b c d
```

TESTS:

```
sage: p = iet.Permutation('a b','a b')
sage: p == p.top_bottom_inverse()
True
sage: p is p.top_bottom_inverse()
```

```
    False
    sage: p = iet.GeneralizedPermutation('a a','b b',reduced=True)
    sage: p == p.top_bottom_inverse()
    True
    sage: p is p.top_bottom_inverse()
    False
```

left_right_inverse()
> Returns the left-right inverse.

> You can also use the shorter .lr_inverse()

> OUTPUT:

> – a permutation

> EXAMPLES:
```
    sage: p = iet.Permutation('a b c','c a b')
    sage: p.left_right_inverse()
    c b a
    b a c
    sage: p = iet.Permutation('a b c d','c d a b')
    sage: p.left_right_inverse()
    d c b a
    b a d c

    sage: p = iet.GeneralizedPermutation('a a','b b c c')
    sage: p.left_right_inverse()
    a a
    c c b b

    sage: p = iet.Permutation('a b c','c b a',reduced=True)
    sage: p.left_right_inverse() == p
    True
    sage: p = iet.Permutation('a b c','c a b',reduced=True)
    sage: q = p.left_right_inverse()
    sage: q == p
    False
    sage: q
    a b c
    b c a

    sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
    sage: p.left_right_inverse() == p
    True
    sage: p = iet.GeneralizedPermutation('a b b','c c a',reduced=True)
    sage: q = p.left_right_inverse()
    sage: q == p
    False
    sage: q
    a a b
    b c c
```

> TESTS:
```
    sage: p = iet.GeneralizedPermutation('a a','b b')
    sage: p.left_right_inverse()
    a a
    b b
    sage: p is p.left_right_inverse()
```

```
False
sage: p == p.left_right_inverse()
True
```

letters()

Returns the list of letters of the alphabet used for representation.

The letters used are not necessarily the whole alphabet (for example if the alphabet is infinite).

OUTPUT:

– a list of labels

EXAMPLES:

```
sage: p = iet.Permutation([1,2],[2,1])
sage: p.alphabet(Alphabet(name="NN"))
sage: p
0 1
1 0
sage: p.letters()
[0, 1]
```

lr_inverse()

Returns the left-right inverse.

You can also use the shorter .lr_inverse()

OUTPUT:

– a permutation

EXAMPLES:

```
sage: p = iet.Permutation('a b c','c a b')
sage: p.left_right_inverse()
c b a
b a c
sage: p = iet.Permutation('a b c d','c d a b')
sage: p.left_right_inverse()
d c b a
b a d c

sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: p.left_right_inverse()
a a
c c b b

sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: p.left_right_inverse() == p
True
sage: p = iet.Permutation('a b c','c a b',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
sage: q
a b c
b c a

sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
sage: p.left_right_inverse() == p
True
```

```
sage: p = iet.GeneralizedPermutation('a b b','c c a',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
sage: q
a a b
b c c
```

TESTS:
```
sage: p = iet.GeneralizedPermutation('a a','b b')
sage: p.left_right_inverse()
a a
b b
sage: p is p.left_right_inverse()
False
sage: p == p.left_right_inverse()
True
```

rauzy_move(*winner*, *side='right'*, *iteration=1*)
   Returns the permutation after a Rauzy move.

   INPUT:

      •`winner` - 'top' or 'bottom' interval

      •`side` - 'right' or 'left' (defaut: 'right') corresponding to the side on which the Rauzy move must be performed.

      •`iteration` - a non negative integer

   OUTPUT:

      •a permutation

   TESTS:
```
sage: p = iet.Permutation('a b','b a')
sage: p.rauzy_move(winner=0, side='right') == p
True
sage: p.rauzy_move(winner=1, side='right') == p
True
sage: p.rauzy_move(winner=0, side='left') == p
True
sage: p.rauzy_move(winner=1, side='left') == p
True

sage: p = iet.Permutation('a b c','c b a')
sage: p.rauzy_move(winner=0, side='right')
a b c
c a b
sage: p.rauzy_move(winner=1, side='right')
a c b
c b a
sage: p.rauzy_move(winner=0, side='left')
a b c
b c a
sage: p.rauzy_move(winner=1, side='left')
b a c
c b a
```

str(*sep='n'*)

A string representation of the generalized permutation.

INPUT:

- **sep** - (default: 'n') a separator for the two intervals

OUTPUT:

string – the string that represents the permutation

EXAMPLES:

For permutations of iet:

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.str()
'a b c\nc b a'
sage: p.str(sep=' | ')
'a b c | c b a'
```

..the permutation can be rebuilt from the standard string:

```
sage: p == iet.Permutation(p.str())
True
```

For permutations of li:

```
sage: p = iet.GeneralizedPermutation('a b b','c c a')
sage: p.str()
'a b b\nc c a'
sage: p.str(sep=' | ')
'a b b | c c a'
```

..the generalized permutation can be rebuilt from the standard string:

```
sage: p == iet.GeneralizedPermutation(p.str())
True
```

symmetric()

Returns the symmetric permutation.

The symmetric permutation is the composition of the top-bottom inversion and the left-right inversion (which are geometrically orientation reversing).

OUTPUT:

– a permutation

EXAMPLES:

```
sage: p = iet.Permutation("a b c","c b a")
sage: p.symmetric()
a b c
c b a
sage: q = iet.Permutation("a b c d","b d a c")
sage: q.symmetric()
c a d b
d c b a

sage: p = iet.Permutation('a b c d','c a d b')
sage: q = p.symmetric()
sage: q1 = p.tb_inverse().lr_inverse()
sage: q2 = p.lr_inverse().tb_inverse()
```

```
sage: q == q1
True
sage: q == q2
True
```

TESTS:
```
sage: p = iet.GeneralizedPermutation('a a b','b c c',reduced=True)
sage: q = p.symmetric()
sage: q1 = p.tb_inverse().lr_inverse()
sage: q2 = p.lr_inverse().tb_inverse()
sage: q == q1
True
sage: q == q2
True

sage: p = iet.GeneralizedPermutation('a a b','b c c',reduced=True,flips='a')
sage: q = p.symmetric()
sage: q1 = p.tb_inverse().lr_inverse()
sage: q2 = p.lr_inverse().tb_inverse()
sage: q == q1
True
sage: q == q2
True
```

tb_inverse()
> Returns the top-bottom inverse.

> You can use also use the shorter .tb_inverse().

> OUTPUT:

> – a permutation

> EXAMPLES:
> ```
> sage: p = iet.Permutation('a b','b a')
> sage: p.top_bottom_inverse()
> b a
> a b
> sage: p = iet.Permutation('a b','b a',reduced=True)
> sage: p.top_bottom_inverse() == p
> True
>
> sage: p = iet.Permutation('a b c d','c d a b')
> sage: p.top_bottom_inverse()
> c d a b
> a b c d
> ```

> TESTS:
> ```
> sage: p = iet.Permutation('a b','a b')
> sage: p == p.top_bottom_inverse()
> True
> sage: p is p.top_bottom_inverse()
> False
> sage: p = iet.GeneralizedPermutation('a a','b b',reduced=True)
> sage: p == p.top_bottom_inverse()
> True
> sage: p is p.top_bottom_inverse()
> False
> ```

top_bottom_inverse()
> Returns the top-bottom inverse.
>
> You can use also use the shorter .tb_inverse().
>
> OUTPUT:
>
> – a permutation
>
> EXAMPLES:
> ```
> sage: p = iet.Permutation('a b','b a')
> sage: p.top_bottom_inverse()
> b a
> a b
> sage: p = iet.Permutation('a b','b a',reduced=True)
> sage: p.top_bottom_inverse() == p
> True
>
> sage: p = iet.Permutation('a b c d','c d a b')
> sage: p.top_bottom_inverse()
> c d a b
> a b c d
> ```
>
> TESTS:
> ```
> sage: p = iet.Permutation('a b','a b')
> sage: p == p.top_bottom_inverse()
> True
> sage: p is p.top_bottom_inverse()
> False
> sage: p = iet.GeneralizedPermutation('a a','b b',reduced=True)
> sage: p == p.top_bottom_inverse()
> True
> sage: p is p.top_bottom_inverse()
> False
> ```

vertical_inverse()
> Returns the left-right inverse.
>
> You can also use the shorter .lr_inverse()
>
> OUTPUT:
>
> – a permutation
>
> EXAMPLES:
> ```
> sage: p = iet.Permutation('a b c','c a b')
> sage: p.left_right_inverse()
> c b a
> b a c
> sage: p = iet.Permutation('a b c d','c d a b')
> sage: p.left_right_inverse()
> d c b a
> b a d c
>
> sage: p = iet.GeneralizedPermutation('a a','b b c c')
> sage: p.left_right_inverse()
> a a
> c c b b
> ```

```
sage: p = iet.Permutation('a b c','c b a',reduced=True)
sage: p.left_right_inverse() == p
True
sage: p = iet.Permutation('a b c','c a b',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
sage: q
a b c
b c a

sage: p = iet.GeneralizedPermutation('a a','b b c c',reduced=True)
sage: p.left_right_inverse() == p
True
sage: p = iet.GeneralizedPermutation('a b b','c c a',reduced=True)
sage: q = p.left_right_inverse()
sage: q == p
False
sage: q
a a b
b c c
```

TESTS:
```
sage: p = iet.GeneralizedPermutation('a a','b b')
sage: p.left_right_inverse()
a a
b b
sage: p is p.left_right_inverse()
False
sage: p == p.left_right_inverse()
True
```

class sage.dynamics.interval_exchanges.template.**PermutationIET**
    Bases: `sage.dynamics.interval_exchanges.template.Permutation`

Template for permutation from Interval Exchange Transformation.

> **Warning:** Internal class! Do not use directly!

AUTHOR:

- Vincent Delecroix (2008-12-20): initial version

arf_invariant()
    Returns the Arf invariant of the suspension of self.

    OUTPUT:

    integer – 0 or 1

    EXAMPLES:

    Permutations from the odd and even component of H(2,2,2):
```
sage: a = range(10)
sage: b1 = [3,2,4,6,5,7,9,8,1,0]
sage: b0 = [6,5,4,3,2,7,9,8,1,0]
sage: p1 = iet.Permutation(a,b1)
sage: print p1.arf_invariant()
1
```

```
sage: p0 = iet.Permutation(a,b0)
sage: print p0.arf_invariant()
0
```

Permutations from the odd and even component of H(4,4):

```
sage: a = range(11)
sage: b1 = [3,2,5,4,6,8,7,10,9,1,0]
sage: b0 = [5,4,3,2,6,8,7,10,9,1,0]
sage: p1 = iet.Permutation(a,b1)
sage: print p1.arf_invariant()
1
sage: p0 = iet.Permutation(a,b0)
sage: print p0.arf_invariant()
0
```

REFERENCES:

[Jo80] D. Johnson, "Spin structures and quadratic forms on surfaces", J. London Math. Soc (2), 22, 1980, 365-373

[KoZo03] M. Kontsevich, A. Zorich "Connected components of the moduli spaces of Abelian differentials with prescribed singularities", Inventiones Mathematicae, 153, 2003, 631-678

attached_in_degree()
    Returns the degree of the singularity at the right of the interval.

    OUTPUT:

    – a positive integer

    EXAMPLES:

```
sage: p1 = iet.Permutation('a b c d e f g','d c g f e b a')
sage: p2 = iet.Permutation('a b c d e f g','e d c g f b a')
sage: p1.attached_in_degree()
1
sage: p2.attached_in_degree()
3
```

attached_out_degree()
    Returns the degree of the singularity at the left of the interval.

    OUTPUT:

    – a positive integer

    EXAMPLES:

```
sage: p1 = iet.Permutation('a b c d e f g','d c g f e b a')
sage: p2 = iet.Permutation('a b c d e f g','e d c g f b a')
sage: p1.attached_out_degree()
3
sage: p2.attached_out_degree()
1
```

attached_type()
    Return the singularity degree attached on the left and the right.

    OUTPUT:

    ([degre], angle_parity) – if the same singularity is attached on the left and right

([left_degree, right_degree], 0) – the degrees at the left and the right which are different singularitites

EXAMPLES:

With two intervals:
```
sage: p = iet.Permutation('a b','b a')
sage: p.attached_type()
([0], 1)
```

With three intervals:
```
sage: p = iet.Permutation('a b c','b c a')
sage: p.attached_type()
([0], 1)
```

```
sage: p = iet.Permutation('a b c','c a b')
sage: p.attached_type()
([0], 1)
```

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.attached_type()
([0, 0], 0)
```

With four intervals:
```
sage: p = iet.Permutation('1 2 3 4','4 3 2 1')
sage: p.attached_type()
([2], 0)
```

connected_component (*marked_separatrix='no'*)
    Returns a connected components of a stratum.

    EXAMPLES:

    Permutations from the stratum H(6):
```
sage: a = range(8)
sage: b_hyp = [7,6,5,4,3,2,1,0]
sage: b_odd = [3,2,5,4,7,6,1,0]
sage: b_even = [5,4,3,2,7,6,1,0]
sage: p_hyp = iet.Permutation(a, b_hyp)
sage: p_odd = iet.Permutation(a, b_odd)
sage: p_even = iet.Permutation(a, b_even)
sage: print p_hyp.connected_component()
H_hyp(6)
sage: print p_odd.connected_component()
H_odd(6)
sage: print p_even.connected_component()
H_even(6)
```

    Permutations from the stratum H(4,4):
```
sage: a = range(11)
sage: b_hyp = [10,9,8,7,6,5,4,3,2,1,0]
sage: b_odd = [3,2,5,4,6,8,7,10,9,1,0]
sage: b_even = [5,4,3,2,6,8,7,10,9,1,0]
sage: p_hyp = iet.Permutation(a,b_hyp)
sage: p_odd = iet.Permutation(a,b_odd)
sage: p_even = iet.Permutation(a,b_even)
sage: p_hyp.stratum() == AbelianStratum(4,4)
```

```
True
sage: print p_hyp.connected_component()
H_hyp(4, 4)
sage: p_odd.stratum() == AbelianStratum(4,4)
True
sage: print p_odd.connected_component()
H_odd(4, 4)
sage: p_even.stratum() == AbelianStratum(4,4)
True
sage: print p_even.connected_component()
H_even(4, 4)
```

As for stratum you can specify that you want to attach the singularity on the left of the interval using the option marked_separatrix:

```
sage: a = [1,2,3,4,5,6,7,8,9]
sage: b4_odd = [4,3,6,5,7,9,8,2,1]
sage: b4_even = [6,5,4,3,7,9,8,2,1]
sage: b2_odd = [4,3,5,7,6,9,8,2,1]
sage: b2_even = [7,6,5,4,3,9,8,2,1]
sage: p4_odd = iet.Permutation(a,b4_odd)
sage: p4_even = iet.Permutation(a,b4_even)
sage: p2_odd = iet.Permutation(a,b2_odd)
sage: p2_even = iet.Permutation(a,b2_even)
sage: p4_odd.connected_component(marked_separatrix='out')
H_odd^out(4, 2)
sage: p4_even.connected_component(marked_separatrix='out')
H_even^out(4, 2)
sage: p2_odd.connected_component(marked_separatrix='out')
H_odd^out(2, 4)
sage: p2_even.connected_component(marked_separatrix='out')
H_even^out(2, 4)
sage: p2_odd.connected_component() == p4_odd.connected_component()
True
sage: p2_odd.connected_component('out') == p4_odd.connected_component('out')
False
```

cylindric()

Returns a permutation in the Rauzy class such that

twin[0][-1] == 0 twin[1][-1] == 0

TESTS:

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.cylindric() == p
True
sage: p = iet.Permutation('a b c d','b d a c')
sage: q = p.cylindric()
sage: q[0][0] == q[1][-1]
True
sage: q[1][0] == q[1][0]
True
```

decompose()

Returns the decomposition of self.

OUTPUT:

– a list of permutations

EXAMPLES:
```
sage: p = iet.Permutation('a b c','c b a').decompose()[0]
sage: p
a b c
c b a

sage: p1,p2,p3 = iet.Permutation('a b c d e','b a c e d').decompose()
sage: p1
a b
b a
sage: p2
c
c
sage: p3
d e
e d
```

erase_marked_points()
> Returns a permutation equivalent to self but without marked points.

> EXAMPLES:
```
sage: a = iet.Permutation('a b1 b2 c d', 'd c b1 b2 a')
sage: a.erase_marked_points()
a b1 c d
d c b1 a
```

genus()
> Returns the genus corresponding to any suspension of the permutation.

> OUTPUT:

> – a positive integer

> EXAMPLES:
```
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.genus()
1

sage: p = iet.Permutation('a b c d','d c b a')
sage: p.genus()
2
```

> **REFERENCES:** Veech

intersection_matrix()
> Returns the intersection matrix.

> This $d * d$ antisymmetric matrix is given by the rule :

$$m_{ij} = \begin{cases} 1 & i < j \text{ and } \pi(i) > \pi(j) \\ -1 & i > j \text{ and } \pi(i) < \pi(j) \\ 0 & \text{else} \end{cases}$$

> OUTPUT:

> •a matrix

> EXAMPLES:

```
sage: p = iet.Permutation('a b c d','d c b a')
sage: p.intersection_matrix()
[ 0  1  1  1]
[-1  0  1  1]
[-1 -1  0  1]
[-1 -1 -1  0]

sage: p = iet.Permutation('1 2 3 4 5','5 3 2 4 1')
sage: p.intersection_matrix()
[ 0  1  1  1  1]
[-1  0  1  0  1]
[-1 -1  0  0  1]
[-1  0  0  0  1]
[-1 -1 -1 -1  0]
```

is_cylindric()
    Returns True if the permutation is Rauzy_1n.

    A permutation is cylindric if 1 and n are exchanged.

    EXAMPLES:
```
sage: iet.Permutation('1 2 3','3 2 1').is_cylindric()
True
sage: iet.Permutation('1 2 3','2 1 3').is_cylindric()
False
```

is_hyperelliptic()
    Returns True if the permutation is in the class of the symmetric permutations (with eventual marked points).

    This is equivalent to say that the suspension lives in an hyperelliptic stratum of Abelian differentials H_hyp(2g-2) or H_hyp(g-1, g-1) with some marked points.

    EXAMPLES:
```
sage: iet.Permutation('a b c d','d c b a').is_hyperelliptic()
True
sage: iet.Permutation('0 1 2 3 4 5','5 2 1 4 3 0').is_hyperelliptic()
False
```

    REFERENCES:

    Gerard Rauzy, "Echanges d'intervalles et transformations induites", Acta Arith. 34, no. 3, 203-212, 1980

    M. Kontsevich, A. Zorich "Connected components of the moduli space of Abelian differentials with prescripebd singularities" Invent. math. 153, 631-678 (2003)

is_irreducible(*return_decomposition=False*)
    Tests the irreducibility.

    An abelian permutation p = (p0,p1) is reducible if: set(p0[:i]) = set(p1[:i]) for an i < len(p0)

    OUTPUT:

    •a boolean

    EXAMPLES:
```
sage: p = iet.Permutation('a b c', 'c b a')
sage: p.is_irreducible()
True

sage: p = iet.Permutation('a b c', 'b a c')
```

```
sage: p.is_irreducible()
False
```

order_of_rauzy_action(*winner*, *side=None*)

Returns the order of the action of a Rauzy move.

INPUT:

- `winner` - string `'top'` or `'bottom'`

- `side` - string `'left'` or `'right'`

OUTPUT:

An integer corresponding to the order of the Rauzy action.

EXAMPLES:

```
sage: p = iet.Permutation('a b c d','d a c b')
sage: p.order_of_rauzy_action('top', 'right')
3
sage: p.order_of_rauzy_action('bottom', 'right')
2
sage: p.order_of_rauzy_action('top', 'left')
1
sage: p.order_of_rauzy_action('bottom', 'left')
3
```

separatrix_diagram(*side=False*)

Returns the separatrix diagram of the permutation.

INPUT:

- `side` - boolean

OUTPUT:

– a list of lists

EXAMPLES:

```
sage: iet.Permutation([0, 1], [1, 0]).separatrix_diagram()
[[(1, 0), (1, 0)]]

sage: iet.Permutation('a b c d','d c b a').separatrix_diagram()
[[('d', 'a'), 'b', 'c', ('d', 'a'), 'b', 'c']]
```

stratum(*marked_separatrix='no'*)

Returns the strata in which any suspension of this permutation lives.

OUTPUT:

- a stratum of Abelian differentials

EXAMPLES:

```
sage: p = iet.Permutation('a b c', 'c b a')
sage: print p.stratum()
H(0, 0)

sage: p = iet.Permutation('a b c d', 'd a b c')
sage: print p.stratum()
H(0, 0, 0)
```

```
sage: p = iet.Permutation(range(9), [8,5,2,7,4,1,6,3,0])
sage: print p.stratum()
H(1, 1, 1, 1)
```

You can specify that you want to attach the singularity on the left (or on the right) with the option marked_separatrix:

```
sage: a = 'a b c d e f g h i j'
sage: b3 = 'd c g f e j i h b a'
sage: b2 = 'd c e g f j i h b a'
sage: b1 = 'e d c g f h j i b a'
sage: p3 = iet.Permutation(a, b3)
sage: p3.stratum()
H(3, 2, 1)
sage: p3.stratum(marked_separatrix='out')
H^out(3, 2, 1)
sage: p2 = iet.Permutation(a, b2)
sage: p2.stratum()
H(3, 2, 1)
sage: p2.stratum(marked_separatrix='out')
H^out(2, 3, 1)
sage: p1 = iet.Permutation(a, b1)
sage: p1.stratum()
H(3, 2, 1)
sage: p1.stratum(marked_separatrix='out')
H^out(1, 3, 2)
```

AUTHORS:

- Vincent Delecroix (2008-12-20)

to_permutation()
    Returns the permutation as an element of the symetric group.

EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: p.to_permutation()
[3, 2, 1]

sage: p = Permutation([2,4,1,3])
sage: q = iet.Permutation(p)
sage: q.to_permutation() == p
True
```

class sage.dynamics.interval_exchanges.template.**PermutationLI**
    Bases: `sage.dynamics.interval_exchanges.template.Permutation`

Template for quadratic permutation.

Warning: Internal class! Do not use directly!

AUTHOR:

- Vincent Delecroix (2008-12-20): initial version

has_right_rauzy_move(*winner*)
    Test of Rauzy movability (with an eventual specified choice of winner)

A quadratic (or generalized) permutation is rauzy_movable type depending on the possible length of the last interval. It's dependent of the length equation.

INPUT:

•`winner` - the integer 'top' or 'bottom'

EXAMPLES:
```
sage: p = iet.GeneralizedPermutation('a a','b b')
sage: p.has_right_rauzy_move('top')
False
sage: p.has_right_rauzy_move('bottom')
False

sage: p = iet.GeneralizedPermutation('a a b','b c c')
sage: p.has_right_rauzy_move('top')
True
sage: p.has_right_rauzy_move('bottom')
True

sage: p = iet.GeneralizedPermutation('a a','b b c c')
sage: p.has_right_rauzy_move('top')
True
sage: p.has_right_rauzy_move('bottom')
False

sage: p = iet.GeneralizedPermutation('a a b b','c c')
sage: p.has_right_rauzy_move('top')
False
sage: p.has_right_rauzy_move('bottom')
True
```

is_irreducible(*return_decomposition=False*)
    Test of reducibility

    A quadratic (or generalized) permutation is *reducible* if there exists a decomposition

    $$A1uB1|...|B1uA2$$
    $$A1uB2|...|B2uA2$$

    where no corners is empty, or exactly one corner is empty and it is on the left, or two and they are both on the right or on the left. The definition is due to [BL08] where they prove that the property of being irreducible is stable under Rauzy induction.

    INPUT:

        •`return_decomposition` - boolean (default: False) - if True, and the permutation is reducible, returns also the blocs A1 u B1, B1 u A2, A1 u B2 and B2 u A2 of a decomposition as above.

    OUTPUT:

    If return_decomposition is True, returns a 2-uple (test,decomposition) where test is the preceding test and decomposition is a 4-uple (A11,A12,A21,A22) where:

    A11 = A1 u B1 A12 = B1 u A2 A21 = A1 u B2 A22 = B2 u A2

    EXAMPLES:
```
sage: GP = iet.GeneralizedPermutation

sage: GP('a a','b b').is_irreducible()
False
```

```
sage: GP('a a b','b c c').is_irreducible()
True
sage: GP('1 2 3 4 5 1','5 6 6 4 3 2').is_irreducible()
True
```

TESTS:

Test reducible permutations with no empty corner:
```
sage: GP('1 4 1 3','4 2 3 2').is_irreducible(True)
(False, (['1', '4'], ['1', '3'], ['4', '2'], ['3', '2']))
```

Test reducible permutations with one left corner empty:
```
sage: GP('1 2 2 3 1','4 4 3').is_irreducible(True)
(False, (['1'], ['3', '1'], [], ['3']))
sage: GP('4 4 3','1 2 2 3 1').is_irreducible(True)
(False, ([], ['3'], ['1'], ['3', '1']))
```

Test reducible permutations with two left corners empty:
```
sage: GP('1 1 2 3','4 2 4 3').is_irreducible(True)
(False, ([], ['3'], [], ['3']))
```

Test reducible permutations with two right corners empty:
```
sage: GP('1 2 2 3 3','1 4 4').is_irreducible(True)
(False, (['1'], [], ['1'], []))
sage: GP('1 2 2','1 3 3').is_irreducible(True)
(False, (['1'], [], ['1'], []))
sage: GP('1 2 3 3','2 1 4 4 5 5').is_irreducible(True)
(False, (['1', '2'], [], ['2', '1'], []))
```

AUTHORS:

- Vincent Delecroix (2008-12-20)

class sage.dynamics.interval_exchanges.template.**RauzyDiagram**(*p*,
right_induction=True,
left_induction=False,
left_right_inversion=False,
top_bottom_inversion=False,
symmetric=False)

Bases: `sage.structure.sage_object.SageObject`

Template for Rauzy diagrams.

AUTHORS:

- Vincent Delecroix (2008-12-20): initial version

class **Path**(*parent*, **data*)
Bases: `sage.structure.sage_object.SageObject`

Path in Rauzy diagram.

> A path in a Rauzy diagram corresponds to a subsimplex of the simplex of lengths. This corre-
> spondance is obtained via the Rauzy induction. To a idoc IET we can associate a unique path in
> a Rauzy diagram. This establishes a correspondance between infinite full path in Rauzy diagram
> and equivalence topologic class of IET.

append(*edge_type*)

    Append an edge to the path.

    EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p)
sage: g.append('top')
sage: g
Path of length 1 in a Rauzy diagram
sage: g.append('bottom')
sage: g
Path of length 2 in a Rauzy diagram
```

composition(*function*, *composition=None*)

    Compose an edges function on a path

    INPUT:

        •`path` - either a Path or a tuple describing a path

        •`function` - function must be of the form

        •`composition` - the composition function

    AUTHOR:

        •Vincent Delecroix (2009-09-29)

    EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: def f(i,t):
....:     if t is None: return []
....:     return [t]
sage: g = r.path(p)
sage: g.composition(f,list.__add__)
[]
sage: g = r.path(p,0,1)
sage: g.composition(f, list.__add__)
[0, 1]
```

edge_types()

    Returns the edge types of the path.

    EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p, 0, 1)
sage: g.edge_types()
[0, 1]
```

end()

    Returns the last vertex of the path.

    EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g1 = r.path(p, 't', 'b', 't')
sage: g1.end() == p
True
sage: g2 = r.path(p, 'b', 't', 'b')
sage: g2.end() == p
True
```

extend(*path*)

Extends self with another path.

EXAMPLES:

```
sage: p = iet.Permutation('a b c d','d c b a')
sage: r = p.rauzy_diagram()
sage: g1 = r.path(p,'t','t')
sage: g2 = r.path(p.rauzy_move('t',iteration=2),'b','b')
sage: g = r.path(p,'t','t','b','b')
sage: g == g1 + g2
True
sage: g = copy(g1)
sage: g.extend(g2)
sage: g == g1 + g2
True
```

is_loop()

Tests whether the path is a loop (start point = end point).

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: r.path(p).is_loop()
True
sage: r.path(p,0,1,0,0).is_loop()
True
```

losers()

Returns a list of the loosers on the path.

EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g0 = r.path(p,'t','b','t')
sage: g0.losers()
['a', 'c', 'b']
sage: g1 = r.path(p,'b','t','b')
sage: g1.losers()
['c', 'a', 'b']
```

pop()

Pops the queue of the path

OUTPUT:

a path corresponding to the last edge

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p,0,1,0)
sage: g0,g1,g2,g3 = g[0], g[1], g[2], g[3]
sage: g.pop() == r.path(g2,0)
True
sage: g == r.path(g0,0,1)
True
sage: g.pop() == r.path(g1,1)
True
sage: g == r.path(g0,0)
True
```

```
sage: g.pop() == r.path(g0,0)
True
sage: g == r.path(g0)
True
sage: g.pop() == r.path(g0)
True
```

right_composition (*function*, *composition=None*)

Compose an edges function on a path

INPUT:

- `function` - function must be of the form (indice,type) -> element. Moreover function(None,None) must be an identity element for initialization.
- `composition` - the composition function for the function. * if None (defaut None)

TEST:

```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: def f(i,t):
....:     if t is None: return []
....:     return [t]
sage: g = r.path(p)
sage: g.right_composition(f,list.__add__)
[]
sage: g = r.path(p,0,1)
sage: g.right_composition(f, list.__add__)
[1, 0]
```

start ()

Returns the first vertex of the path.

EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p, 't', 'b')
sage: g.start() == p
True
```

winners ()

Returns the winner list associated to the edge of the path.

EXAMPLES:

```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: r.path(p).winners()
[]
sage: r.path(p,0).winners()
['b']
sage: r.path(p,1).winners()
['a']
```

RauzyDiagram.**alphabet** (*data=None*)

TESTS:

```
sage: r = iet.RauzyDiagram('a b','b a')
sage: r.alphabet() == Alphabet(['a','b'])
True
sage: r = iet.RauzyDiagram([0,1],[1,0])
sage: r.alphabet() == Alphabet([0,1])
True
```

RauzyDiagram.**cardinality**()
    Returns the number of permutations in this Rauzy diagram.

    OUTPUT:

        •*integer* - the number of vertices in the diagram

    EXAMPLES:
```
sage: r = iet.RauzyDiagram('a b','b a')
sage: r.cardinality()
1
sage: r = iet.RauzyDiagram('a b c','c b a')
sage: r.cardinality()
3
sage: r = iet.RauzyDiagram('a b c d','d c b a')
sage: r.cardinality()
7
```

RauzyDiagram.**complete**(*p*)
    Completion of the Rauzy diagram.

    Add to the Rauzy diagram all permutations that are obtained by successive operations defined by
    edge_types(). The permutation must be of the same type and the same length as the one used for the
    creation.

    INPUT:

        •p - a permutation of Interval exchange transformation

    Rauzy diagram is the reunion of all permutations that could be obtained with successive rauzy moves. This
    function just use the functions __getitem__ and has_rauzy_move and rauzy_move which must be defined
    for child and their corresponding permutation types.

    TEST:
```
sage: r = iet.RauzyDiagram('a b c','c b a')    #indirect doctest
sage: r = iet.RauzyDiagram('a b c','c b a',left_induction=True) #indirect doctest
sage: r = iet.RauzyDiagram('a b c','c b a',symmetric=True)    #indirect doctest
sage: r = iet.RauzyDiagram('a b c','c b a',lr_inversion=True)    #indirect doctest
sage: r = iet.RauzyDiagram('a b c','c b a',tb_inversion=True)    #indirect doctest
```

RauzyDiagram.**edge_iterator**()
    Returns an iterator over the edges of the graph.

    EXAMPLES:
```
sage: p = iet.Permutation('a b','b a')
sage: r = p.rauzy_diagram()
sage: for e in r.edge_iterator():
....:  print e[0].str(sep='/'), '-->', e[1].str(sep='/')
a b/b a --> a b/b a
a b/b a --> a b/b a
```

RauzyDiagram.**edge_to_loser**(*p=None*, *edge_type=None*)
    Return the corresponding loser

    TEST:
```
sage: r = iet.RauzyDiagram('a b','b a')
sage: r.edge_to_loser(None,None)
[]
```

RauzyDiagram.**edge_to_matrix**(*p=None*, *edge_type=None*)
Return the corresponding matrix

INPUT:

•`p` - a permutation

•`edge_type` - 0 or 1 corresponding to the type of the edge

OUTPUT:

A matrix

EXAMPLES:
```
sage: p = iet.Permutation('a b c','c b a')
sage: d = p.rauzy_diagram()
sage: print d.edge_to_matrix(p,1)
[1 0 1]
[0 1 0]
[0 0 1]
```

RauzyDiagram.**edge_to_winner**(*p=None*, *edge_type=None*)
Return the corresponding winner

TEST:
```
sage: r = iet.RauzyDiagram('a b','b a')
sage: r.edge_to_winner(None,None)
[]
```

RauzyDiagram.**edge_types**()
Print information about edges.

EXAMPLES:
```
sage: r = iet.RauzyDiagram('a b', 'b a')
sage: r.edge_types()
0: rauzy_move(0, -1)
1: rauzy_move(1, -1)

sage: r = iet.RauzyDiagram('a b', 'b a', left_induction=True)
sage: r.edge_types()
0: rauzy_move(0, -1)
1: rauzy_move(1, -1)
2: rauzy_move(0, 0)
3: rauzy_move(1, 0)

sage: r = iet.RauzyDiagram('a b',' b a',symmetric=True)
sage: r.edge_types()
0: rauzy_move(0, -1)
1: rauzy_move(1, -1)
2: symmetric()
```

RauzyDiagram.**edge_types_index**(*data*)
Try to convert the data as an edge type.

INPUT:

•`data` - a string

OUTPUT:

integer

EXAMPLES:

For a standard Rauzy diagram (only right induction) the 0 index corresponds to the 'top' induction and the index 1 corresponds to the 'bottom' one:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: r.edge_types_index('top')
0
sage: r[p][0] == p.rauzy_move('top')
True
sage: r.edge_types_index('bottom')
1
sage: r[p][1] == p.rauzy_move('bottom')
True
```

The special operations (inversion and symmetry) always appears after the different Rauzy inductions:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram(symmetric=True)
sage: r.edge_types_index('symmetric')
2
sage: r[p][2] == p.symmetric()
True
```

This function always try to resolve conflictuous name. If it's impossible a ValueError is raised:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram(left_induction=True)
sage: r.edge_types_index('top')
Traceback (most recent call last):
...
ValueError: left and right inductions must be differentiated
sage: r.edge_types_index('top_right')
0
sage: r[p][0] == p.rauzy_move(0)
True
sage: r.edge_types_index('bottom_left')
3
sage: r[p][3] == p.rauzy_move('bottom', 'left')
True

sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram(left_right_inversion=True,top_bottom_inversion=True)
sage: r.edge_types_index('inversion')
Traceback (most recent call last):
...
ValueError: left-right and top-bottom inversions must be differentiated
sage: r.edge_types_index('lr_inverse')
2
sage: p.lr_inverse() == r[p][2]
True
sage: r.edge_types_index('tb_inverse')
3
sage: p.tb_inverse() == r[p][3]
True
```

Short names are accepted:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram(right_induction='top',top_bottom_inversion=True)
```

```
sage: r.edge_types_index('top_rauzy_move')
0
sage: r.edge_types_index('t')
0
sage: r.edge_types_index('tb')
1
sage: r.edge_types_index('inversion')
1
sage: r.edge_types_index('inverse')
1
sage: r.edge_types_index('i')
1
```

RauzyDiagram.**edges**(*labels=True*)

Returns a list of the edges.

EXAMPLES:
```
sage: r = iet.RauzyDiagram('a b','b a')
sage: len(r.edges())
2
```

RauzyDiagram.**graph**()

Returns the Rauzy diagram as a Graph object

The graph returned is more precisely a DiGraph (directed graph) with loops and multiedges allowed.

EXAMPLES:
```
sage: r = iet.RauzyDiagram('a b c','c b a')
sage: r
Rauzy diagram with 3 permutations
sage: r.graph()
Looped multi-digraph on 3 vertices
```

RauzyDiagram.**letters**()

Returns the letters used by the RauzyDiagram.

EXAMPLES:
```
sage: r = iet.RauzyDiagram('a b','b a')
sage: r.alphabet()
{'a', 'b'}
sage: r.letters()
['a', 'b']
sage: r.alphabet('ABCDEF')
sage: r.alphabet()
{'A', 'B', 'C', 'D', 'E', 'F'}
sage: r.letters()
['A', 'B']
```

RauzyDiagram.**path**(**data*)

Returns a path over this Rauzy diagram.

INPUT:

   •initial_vertex - the initial vertex (starting point of the path)

   •data - a sequence of edges

EXAMPLES:

```
sage: p = iet.Permutation('a b c','c b a')
sage: r = p.rauzy_diagram()
sage: g = r.path(p, 'top', 'bottom')
```

RauzyDiagram.**vertex_iterator**()

Returns an iterator over the vertices

EXAMPLES:
```
sage: r = iet.RauzyDiagram('a b','b a')
sage: for p in r.vertex_iterator(): print p
a b
b a

sage: r = iet.RauzyDiagram('a b c d','d c b a')
sage: from itertools import ifilter
sage: r_1n = ifilter(lambda x: x.is_cylindric(), r)
sage: for p in r_1n: print p
a b c d
d c b a
```

RauzyDiagram.**vertices**()

Returns a list of the vertices.

EXAMPLES:
```
sage: r = iet.RauzyDiagram('a b','b a')
sage: for p in r.vertices(): print p
a b
b a
```

sage.dynamics.interval_exchanges.template.**interval_conversion**(*interval=None*)

Converts the argument in 0 or 1.

INPUT:

•winner - 'top' (or 't' or 0) or bottom (or 'b' or 1)

OUTPUT:

integer – 0 or 1

TESTS:
```
sage: from sage.dynamics.interval_exchanges.template import interval_conversion
sage: interval_conversion('top')
0
sage: interval_conversion('t')
0
sage: interval_conversion(0)
0
sage: interval_conversion('bottom')
1
sage: interval_conversion('b')
1
sage: interval_conversion(1)
1
```

sage.dynamics.interval_exchanges.template.**labelize_flip**(*couple*)

Returns a string from a 2-uple couple of the form (name, flip).

TESTS:

```
sage: from sage.dynamics.interval_exchanges.template import labelize_flip
sage: labelize_flip((0,1))
' 0'
sage: labelize_flip((0,-1))
'-0'
```

sage.dynamics.interval_exchanges.template.**side_conversion**(*side=None*)

Converts the argument in 0 or -1.

INPUT:

   •side - either 'left' (or 'l' or 0) or 'right' (or 'r' or -1)

OUTPUT:

integer – 0 or -1

TESTS:

```
sage: from sage.dynamics.interval_exchanges.template import side_conversion
sage: side_conversion('left')
0
sage: side_conversion('l')
0
sage: side_conversion(0)
0
sage: side_conversion('right')
-1
sage: side_conversion('r')
-1
sage: side_conversion(1)
-1
sage: side_conversion(-1)
-1
```

sage.dynamics.interval_exchanges.template.**twin_list_iet**(*a=None*)

Returns the twin list of intervals.

The twin intervals is the correspondance between positions of labels in such way that a[interval][position] is a[1-interval][twin[interval][position]]

INPUT:

   •a - two lists of labels

OUTPUT:

list – a list of two lists of integers

TESTS:

```
sage: from sage.dynamics.interval_exchanges.template import twin_list_iet
sage: twin_list_iet([['a','b','c'],['a','b','c']])
[[0, 1, 2], [0, 1, 2]]
sage: twin_list_iet([['a','b','c'],['a','c','b']])
[[0, 2, 1], [0, 2, 1]]
sage: twin_list_iet([['a','b','c'],['b','a','c']])
[[1, 0, 2], [1, 0, 2]]
sage: twin_list_iet([['a','b','c'],['b','c','a']])
[[2, 0, 1], [1, 2, 0]]
sage: twin_list_iet([['a','b','c'],['c','a','b']])
[[1, 2, 0], [2, 0, 1]]
```

```
sage: twin_list_iet([['a','b','c'],['c','b','a']])
[[2, 1, 0], [2, 1, 0]]
```

sage.dynamics.interval_exchanges.template.**twin_list_li**(*a=None*)

Returns the twin list of intervals

INPUT:

- a - two lists of labels

OUTPUT:

list – a list of two lists of couples of integers

TESTS:

```
sage: from sage.dynamics.interval_exchanges.template import twin_list_li
sage: twin_list_li([['a','a','b','b'],[]])
[[(0, 1), (0, 0), (0, 3), (0, 2)], []]
sage: twin_list_li([['a','a','b'],['b']])
[[(0, 1), (0, 0), (1, 0)], [(0, 2)]]
sage: twin_list_li([['a','a'],['b','b']])
[[(0, 1), (0, 0)], [(1, 1), (1, 0)]]
sage: twin_list_li([['a'], ['a','b','b']])
[[(1, 0)], [(0, 0), (1, 2), (1, 1)]]
sage: twin_list_li([[], ['a','a','b','b']])
[[], [(1, 1), (1, 0), (1, 3), (1, 2)]]
```

1.5 Interval Exchange Transformations and Linear Involution

An interval exchage transformation is a map defined on an interval (see help(iet.IntervalExchangeTransformation) for a more complete help.

EXAMPLES:

Initialization of a simple iet with integer lengths:

```
sage: T = iet.IntervalExchangeTransformation(Permutation([3,2,1]), [3,1,2])
sage: print T
Interval exchange transformation of [0, 6[ with permutation
1 2 3
3 2 1
```

Rotation corresponds to iet with two intervals:

```
sage: p = iet.Permutation('a b', 'b a')
sage: T = iet.IntervalExchangeTransformation(p, [1, (sqrt(5)-1)/2])
sage: print T.in_which_interval(0)
a
sage: print T.in_which_interval(T(0))
a
sage: print T.in_which_interval(T(T(0)))
b
sage: print T.in_which_interval(T(T(T(0))))
a
```

There are two plotting methods for iet:

```
sage: p = iet.Permutation('a b c','c b a')
sage: T = iet.IntervalExchangeTransformation(p, [1, 2, 3])
```

class sage.dynamics.interval_exchanges.iet.**IntervalExchangeTransformation**(*permutation=None*,
lengths=None)

> Bases: sage.structure.sage_object.SageObject

> Interval exchange transformation

> INPUT:

>> •permutation - a permutation (LabelledPermutationIET)

>> •lengths - the list of lengths

> EXAMPLES:

> Direct initialization:
> ```
> sage: p = iet.IET(('a b c','c b a'),{'a':1,'b':1,'c':1})
> sage: p.permutation()
> a b c
> c b a
> sage: p.lengths()
> [1, 1, 1]
> ```

> Initialization from a iet.Permutation:
> ```
> sage: perm = iet.Permutation('a b c','c b a')
> sage: l = [0.5,1,1.2]
> sage: t = iet.IET(perm,l)
> sage: t.permutation() == perm
> True
> sage: t.lengths() == l
> True
> ```

> Initialization from a Permutation:
> ```
> sage: p = Permutation([3,2,1])
> sage: iet.IET(p, [1,1,1])
> Interval exchange transformation of [0, 3[ with permutation
> 1 2 3
> 3 2 1
> ```

> If it is not possible to convert lengths to real values an error is raised:
> ```
> sage: iet.IntervalExchangeTransformation(('a b','b a'),['e','f'])
> Traceback (most recent call last):
> ...
> TypeError: unable to convert x (='e') into a real number
> ```

> The value for the lengths must be positive:
> ```
> sage: iet.IET(('a b','b a'),[-1,-1])
> Traceback (most recent call last):
> ...
> ValueError: lengths must be positive
> ```

> **domain_singularities**()
>> Returns the list of singularities of T

>> OUTPUT:

list – positive reals that corresponds to singularities in the top interval

EXAMPLES:
```
sage: t = iet.IET(("a b","b a"), [1, sqrt(2)])
sage: t.domain_singularities()
[0, 1, sqrt(2) + 1]
```

in_which_interval(*x*, *interval=0*)

Returns the letter for which x is in this interval.

INPUT:

- •x - a positive number

- •interval - (default: 'top') 'top' or 'bottom'

OUTPUT:

label – a label corresponding to an interval

TEST:
```
sage: t = iet.IntervalExchangeTransformation(('a b c','c b a'),[1,1,1])
sage: t.in_which_interval(0)
'a'
sage: t.in_which_interval(0.3)
'a'
sage: t.in_which_interval(1)
'b'
sage: t.in_which_interval(1.9)
'b'
sage: t.in_which_interval(2)
'c'
sage: t.in_which_interval(2.1)
'c'
sage: t.in_which_interval(3)
Traceback (most recent call last):
...
ValueError: your value does not lie in [0;l[
```

TESTS:
```
sage: t.in_which_interval(-2.9,'bottom')
Traceback (most recent call last):
...
ValueError: your value does not lie in [0;l[
```

inverse()

Returns the inverse iet.

OUTPUT:

iet – the inverse interval exchange transformation

EXAMPLES:
```
sage: p = iet.Permutation("a b","b a")
sage: s = iet.IET(p, [1,sqrt(2)-1])
sage: t = s.inverse()
sage: t.permutation()
b a
a b
sage: t.lengths()
```

```
[1, sqrt(2) - 1]
sage: t*s
Interval exchange transformation of [0, sqrt(2)[ with permutation
aa bb
aa bb
```

We can verify with the method .is_identity():
```
sage: p = iet.Permutation("a b c d","d a c b")
sage: s = iet.IET(p, [1, sqrt(2), sqrt(3), sqrt(5)])
sage: (s * s.inverse()).is_identity()
True
sage: (s.inverse() * s).is_identity()
True
```

is_identity()
> Returns True if self is the identity.

> OUTPUT:

> boolean – the answer

> EXAMPLES:
```
sage: p = iet.Permutation("a b","b a")
sage: q = iet.Permutation("c d","d c")
sage: s = iet.IET(p, [1,5])
sage: t = iet.IET(q, [5,1])
sage: (s*t).is_identity()
True
sage: (t*s).is_identity()
True
```

length()
> Returns the total length of the interval.

> OUTPUT:

> real – the length of the interval

> EXAMPLES:
```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),[1,1])
sage: t.length()
2
```

lengths()
> Returns the list of lengths associated to this iet.

> OUTPUT:

> list – the list of lengths of subinterval

> EXAMPLES:
```
sage: p = iet.IntervalExchangeTransformation(('a b','b a'),[1,3])
sage: p.lengths()
[1, 3]
```

normalize(*total=1*)
> Returns a interval exchange transformation of normalized lengths.

The normalization consists in multiplying all lengths by a constant in such way that their sum is given by `total` (default is 1).

INPUT:

>   •`total` - (default: 1) The total length of the interval

OUTPUT:

iet – the normalized iet

EXAMPLES:
```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'), [1,3])
sage: t.length()
4
sage: s = t.normalize(2)
sage: s.length()
2
sage: s.lengths()
[1/2, 3/2]
```

TESTS:
```
sage: s = t.normalize('bla')
Traceback (most recent call last):
...
TypeError: unable to convert total (='bla') into a real number
sage: s = t.normalize(-691)
Traceback (most recent call last):
...
ValueError: the total length must be positive
```

permutation()
>   Returns the permutation associated to this iet.

>   OUTPUT:

>   permutation – the permutation associated to this iet

>   EXAMPLES:
```
sage: perm = iet.Permutation('a b c','c b a')
sage: p = iet.IntervalExchangeTransformation(perm,(1,2,1))
sage: p.permutation() == perm
True
```

plot(*position=(0, 0), vertical_alignment='center', horizontal_alignment='left', interval_height=0.1, labels_height=0.05, fontsize=14, labels=True, colors=None*)
>   Returns a picture of the interval exchange transformation.

>   INPUT:

>>   •`position` - a 2-uple of the position

>>   •`horizontal_alignment` - left (defaut), center or right

>>   •`labels` - boolean (defaut: True)

>>   •`fontsize` - the size of the label

>   OUTPUT:

>   2d plot – a plot of the two intervals (domain and range)

EXAMPLES:
```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),[1,1])
sage: t.plot_two_intervals()
```

plot_function(***d*)

Return a plot of the interval exchange transformation as a function.

INPUT:

- Any option that is accepted by line2d

OUTPUT:

2d plot – a plot of the iet as a function

EXAMPLES:
```
sage: t = iet.IntervalExchangeTransformation(('a b c d','d a c b'),[1,1,1,1])
sage: t.plot_function(rgbcolor=(0,1,0))
```

plot_two_intervals(*position=(0, 0)*, *vertical_alignment='center'*, *horizontal_alignment='left'*, *interval_height=0.1*, *labels_height=0.05*, *fontsize=14*, *labels=True*, *colors=None*)

Returns a picture of the interval exchange transformation.

INPUT:

- `position` - a 2-uple of the position

- `horizontal_alignment` - left (defaut), center or right

- `labels` - boolean (defaut: True)

- `fontsize` - the size of the label

OUTPUT:

2d plot – a plot of the two intervals (domain and range)

EXAMPLES:
```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),[1,1])
sage: t.plot_two_intervals()
```

range_singularities()

Returns the list of singularities of T^{-1}

OUTPUT:

list – real numbers that are singular for T^{-1}

EXAMPLES:
```
sage: t = iet.IET(("a b","b a"), [1, sqrt(2)])
sage: t.range_singularities()
[0, sqrt(2), sqrt(2) + 1]
```

rauzy_move(*side='right'*, *iterations=1*)

Performs a Rauzy move.

INPUT:

- `side` - 'left' (or 'l' or 0) or 'right' (or 'r' or 1)

- **iterations - integer (default :1) the number of iteration of Rauzy** moves to perform

OUTPUT:

iet – the Rauzy move of self

EXAMPLES:

```
sage: phi = QQbar((sqrt(5)-1)/2)
sage: t1 = iet.IntervalExchangeTransformation(('a b','b a'),[1,phi])
sage: t2 = t1.rauzy_move().normalize(t1.length())
sage: l2 = t2.lengths()
sage: l1 = t1.lengths()
sage: l2[0] == l1[1] and l2[1] == l1[0]
True
```

show()

Shows a picture of the interval exchange transformation

EXAMPLES:

```
sage: phi = QQbar((sqrt(5)-1)/2)
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),[1,phi])
sage: t.show()
```

singularities()

The list of singularities of T and T^{-1}.

OUTPUT:

list – two lists of positive numbers which corresponds to extremities of subintervals

EXAMPLE:

```
sage: t = iet.IntervalExchangeTransformation(('a b','b a'),[1/2,3/2])
sage: t.singularities()
[[0, 1/2, 2], [0, 3/2, 2]]
```

ABELIAN DIFFERENTIALS AND FLAT SURFACES

2.1 Strata of differentials on Riemann surfaces

The space of Abelian (or quadratic) differentials is stratified by the degrees of the zeroes (and simple poles for quadratic differentials). Each stratum has one, two or three connected components and each is associated to an (extended) Rauzy class. The `connected_components()` method (only available for Abelian stratum) give the decomposition of a stratum (which corresponds to the SAGE object `AbelianStratum`).

The work for Abelian differentials was done by Maxim Kontsevich and Anton Zorich in [KonZor03] and for quadratic differentials by Erwan Lanneau in [Lan08]. Zorich gave an algorithm to pass from a connected component of a stratum to the associated Rauzy class (for both interval exchange transformations and linear involutions) in [Zor08] and is implemented for Abelian stratum at different level (approximately one for each component):

- for connected stratum `representative()`

- for hyperellitic component `representative()`

- for non hyperelliptic component, the algorithm is the same as for connected component

- for odd component `representative()`

- for even component `representative()`

The inverse operation (pass from an interval exchange transformation to the connected component) is partially written in [KonZor03] and simply named here `connected_component()`.

All the code here was first available on Mathematica [ZS].

REFERENCES:

Note: The quadratic strata are not yet implemented.

AUTHORS:

- Vincent Delecroix (2009-09-29): initial version

EXAMPLES:

Construction of a stratum from a list of singularity degrees:

```
sage: a = AbelianStratum(1,1)
sage: print a
H(1, 1)
```

```
sage: print a.genus()
2
sage: print a.nintervals()
5
```

```
sage: a = AbelianStratum(4,3,2,1)
sage: print a
H(4, 3, 2, 1)
sage: print a.genus()
6
sage: print a.nintervals()
15
```

By convention, the degrees are always written in decreasing order:

```
sage: a1 = AbelianStratum(4,3,2,1)
sage: a1
H(4, 3, 2, 1)
sage: a2 = AbelianStratum(2,3,1,4)
sage: a2
H(4, 3, 2, 1)
sage: a1 == a2
True
```

It is also possible to consider stratum with an incoming or an outgoing separatrix marked (the aim of this consideration is to attach a specified degree at the left or the right of the associated interval exchange transformation):

```
sage: a_out = AbelianStratum(1, 1, marked_separatrix='out')
sage: a_out
H^out(1, 1)
sage: a_in = AbelianStratum(1, 1, marked_separatrix='in')
sage: a_in
H^in(1, 1)
sage: a_out == a_in
False
```

Get a list of strata with constraints on genus or on the number of intervals of a representative:

```
sage: for a in AbelianStrata(genus=3):
....:     print a
H(4)
H(3, 1)
H(2, 2)
H(2, 1, 1)
H(1, 1, 1, 1)
```

```
sage: for a in AbelianStrata(nintervals=5):
....:     print a
H^out(0, 2)
H^out(2, 0)
H^out(1, 1)
H^out(0, 0, 0, 0)
```

```
sage: for a in AbelianStrata(genus=2, nintervals=5):
....:     print a
H^out(0, 2)
H^out(2, 0)
```

```
H^out(1, 1)
```

Obtains the connected components of a stratum:

```
sage: a = AbelianStratum(0)
sage: print a.connected_components()
[H_hyp(0)]
```

```
sage: a = AbelianStratum(6)
sage: cc = a.connected_components()
sage: print cc
[H_hyp(6), H_odd(6), H_even(6)]
sage: for c in cc:
....:     print c, "\n", c.representative(alphabet=range(1,9))
H_hyp(6)
1 2 3 4 5 6 7 8
8 7 6 5 4 3 2 1
H_odd(6)
1 2 3 4 5 6 7 8
4 3 6 5 8 7 2 1
H_even(6)
1 2 3 4 5 6 7 8
6 5 4 3 8 7 2 1
```

```
sage: a = AbelianStratum(1, 1, 1, 1)
sage: print a.connected_components()
[H_c(1, 1, 1, 1)]
sage: c = a.connected_components()[0]
sage: print c.representative(alphabet="abcdefghi")
a b c d e f g h i
e d c f i h g b a
```

The zero attached on the left of the associated Abelian permutation corresponds to the first singularity degree:

```
sage: a = AbelianStratum(4, 2, marked_separatrix='out')
sage: b = AbelianStratum(2, 4, marked_separatrix='out')
sage: print a == b
False
sage: print a, ":", a.connected_components()
H^out(4, 2) : [H_odd^out(4, 2), H_even^out(4, 2)]
sage: print b, ":", b.connected_components()
H^out(2, 4) : [H_odd^out(2, 4), H_even^out(2, 4)]
sage: a_odd, a_even = a.connected_components()
sage: b_odd, b_even = b.connected_components()
```

The representatives are hence different:

```
sage: print a_odd.representative(alphabet=range(1,10))
1 2 3 4 5 6 7 8 9
4 3 6 5 7 9 8 2 1
sage: print b_odd.representative(alphabet=range(1,10))
1 2 3 4 5 6 7 8 9
4 3 5 7 6 9 8 2 1
```

```
sage: print a_even.representative(alphabet=range(1,10))
1 2 3 4 5 6 7 8 9
6 5 4 3 7 9 8 2 1
```

```
sage: print b_even.representative(alphabet=range(1,10))
1 2 3 4 5 6 7 8 9
7 6 5 4 3 9 8 2 1
```

You can retrieve the decomposition of the irreducible Abelian permutations into Rauzy diagrams from the classification of strata:

```
sage: a = AbelianStrata(nintervals=4)
sage: l = sum([stratum.connected_components() for stratum in a], [])
sage: n = map(lambda x: x.rauzy_diagram().cardinality(), l)
sage: for c,i in zip(l,n):
....:     print c, ":", i
H_hyp^out(2) : 7
H_hyp^out(0, 0, 0) : 6
sage: print sum(n)
13
```

```
sage: a = AbelianStrata(nintervals=5)
sage: l = sum([stratum.connected_components() for stratum in a], [])
sage: n = map(lambda x: x.rauzy_diagram().cardinality(), l)
sage: for c,i in zip(l,n):
....:     print c, ":", i
H_hyp^out(0, 2) : 11
H_hyp^out(2, 0) : 35
H_hyp^out(1, 1) : 15
H_hyp^out(0, 0, 0, 0) : 10
sage: print sum(n)
71
```

```
sage: a = AbelianStrata(nintervals=6)
sage: l = sum([stratum.connected_components() for stratum in a], [])
sage: n = map(lambda x: x.rauzy_diagram().cardinality(), l)
sage: for c,i in zip(l,n):
....:     print c, ":", i
H_hyp^out(4) : 31
H_odd^out(4) : 134
H_hyp^out(0, 2, 0) : 66
H_hyp^out(2, 0, 0) : 105
H_hyp^out(0, 1, 1) : 20
H_hyp^out(1, 1, 0) : 90
H_hyp^out(0, 0, 0, 0, 0) : 15
sage: print sum(n)
461
```

sage.dynamics.flat_surfaces.strata.**AbelianStrata**(*genus=None*,   *nintervals=None*,
                                                                                                  marked_separatrix=None)

> Abelian strata.

> INPUT:

>> •`genus` - a non negative integer or `None`

>> •`nintervals` - a non negative integer or `None`

>> •`marked_separatrix` - 'no' (for no marking), 'in' (for marking an incoming separatrix) or 'out' (for marking an outgoing separatrix)

> EXAMPLES:

> Abelian strata with a given genus:

```
sage: for s in AbelianStrata(genus=1): print s
H(0)

sage: for s in AbelianStrata(genus=2): print s
H(2)
H(1, 1)

sage: for s in AbelianStrata(genus=3): print s
H(4)
H(3, 1)
H(2, 2)
H(2, 1, 1)
H(1, 1, 1, 1)

sage: for s in AbelianStrata(genus=4): print s
H(6)
H(5, 1)
H(4, 2)
H(4, 1, 1)
H(3, 3)
H(3, 2, 1)
H(3, 1, 1, 1)
H(2, 2, 2)
H(2, 2, 1, 1)
H(2, 1, 1, 1, 1)
H(1, 1, 1, 1, 1, 1)
```

Abelian strata with a given number of intervals:

```
sage: for s in AbelianStrata(nintervals=2): print s
H^out(0)

sage: for s in AbelianStrata(nintervals=3): print s
H^out(0, 0)

sage: for s in AbelianStrata(nintervals=4): print s
H^out(2)
H^out(0, 0, 0)

sage: for s in AbelianStrata(nintervals=5): print s
H^out(0, 2)
H^out(2, 0)
H^out(1, 1)
H^out(0, 0, 0, 0)
```

Abelian strata with both constraints:

```
sage: for s in AbelianStrata(genus=2, nintervals=4): print s
H^out(2)

sage: for s in AbelianStrata(genus=5, nintervals=12): print s
H^out(8, 0, 0)
H^out(0, 8, 0)
H^out(0, 7, 1)
H^out(1, 7, 0)
H^out(7, 1, 0)
H^out(0, 6, 2)
H^out(2, 6, 0)
H^out(6, 2, 0)
```

```
H^out(1, 6, 1)
H^out(6, 1, 1)
H^out(0, 5, 3)
H^out(3, 5, 0)
H^out(5, 3, 0)
H^out(1, 5, 2)
H^out(2, 5, 1)
H^out(5, 2, 1)
H^out(0, 4, 4)
H^out(4, 4, 0)
H^out(1, 4, 3)
H^out(3, 4, 1)
H^out(4, 3, 1)
H^out(2, 4, 2)
H^out(4, 2, 2)
H^out(2, 3, 3)
H^out(3, 3, 2)
```

class `sage.dynamics.flat_surfaces.strata.`**`AbelianStrata_all`**(*category=None*, **keys*, ***opts*)

    Bases: `sage.combinat.combinat.InfiniteAbstractCombinatorialClass`

    Abelian strata.

class `sage.dynamics.flat_surfaces.strata.`**`AbelianStrata_d`**(*nintervals=None*, *marked_separatrix=None*)

    Bases: `sage.combinat.combinat.CombinatorialClass`

    Strata with constraint number of intervals.

    INPUT:

        •`nintervals` - an integer greater than 1

        •`marked_separatrix` - 'no', 'out' or 'in'

class `sage.dynamics.flat_surfaces.strata.`**`AbelianStrata_g`**(*genus=None*, *marked_separatrix=None*)

    Bases: `sage.combinat.combinat.CombinatorialClass`

    Stratas of genus g surfaces.

    INPUT:

        •`genus` - a non negative integer

        •`marked_separatrix` - 'no', 'out' or 'in'

class `sage.dynamics.flat_surfaces.strata.`**`AbelianStrata_gd`**(*genus=None*, *nintervals=None*, *marked_separatrix=None*)

    Bases: `sage.combinat.combinat.CombinatorialClass`

    Abelian strata of prescribed genus and number of intervals.

    INPUT:

        •`genus` - integer: the genus of the surfaces

        •`nintervals` - integer: the number of intervals

        •`marked_separatrix` - 'no', 'in' or 'out'

class sage.dynamics.flat_surfaces.strata.**AbelianStratum**(*l*, ***d*)

  Bases: sage.structure.sage_object.SageObject

  Stratum of Abelian differentials.

  A stratum with a marked outgoing separatrix corresponds to Rauzy diagram with left induction, a stratum with marked incoming separatrix correspond to Rauzy diagram with right induction. If there is no marked separatrix, the associated Rauzy diagram is the extended Rauzy diagram (consideration of the sage.dynamics.interval_exchanges.template.Permutation.symmetric() operation of Boissy-Lanneau).

  When you want to specify a marked separatrix, the degree on which it is is the first term of your degrees list.

  INPUT:

    •marked_separatrix - None (default) or 'in' (for incoming separatrix) or 'out' (for outgoing separatrix).

  EXAMPLES:

  Creation of an Abelian stratum and get its connected components:

```
sage: a = AbelianStratum(2, 2)
sage: print a
H(2, 2)
sage: a.connected_components()
[H_hyp(2, 2), H_odd(2, 2)]
```

  Specification of marked separatrix:

```
sage: a = AbelianStratum(4,2,marked_separatrix='in')
sage: print a
H^in(4, 2)
sage: b = AbelianStratum(2,4,marked_separatrix='in')
sage: print b
H^in(2, 4)
sage: a == b
False
```

```
sage: a = AbelianStratum(4,2,marked_separatrix='out')
sage: print a
H^out(4, 2)
sage: b = AbelianStratum(2,4,marked_separatrix='out')
sage: print b
H^out(2, 4)
sage: a == b
False
```

  Get a representative of a connected component:

```
sage: a = AbelianStratum(2,2)
sage: a_hyp, a_odd = a.connected_components()
sage: print a_hyp.representative()
1 2 3 4 5 6 7
7 6 5 4 3 2 1
sage: print a_odd.representative()
0 1 2 3 4 5 6
3 2 4 6 5 1 0
```

  You can choose the alphabet:

```
sage: print a_odd.representative(alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ")
A B C D E F G
D C E G F B A
```

By default, you get a reduced permutation, but you can specify that you want a labelled one:

```
sage: p_reduced = a_odd.representative()
sage: p_labelled = a_odd.representative(reduced=False)
```

connected_components()

Lists the connected components of the Stratum.

OUTPUT:

list – a list of connected components of stratum

EXAMPLES:

```
sage: AbelianStratum(0).connected_components()
[H_hyp(0)]

sage: AbelianStratum(2).connected_components()
[H_hyp(2)]
sage: AbelianStratum(1,1).connected_components()
[H_hyp(1, 1)]

sage: AbelianStratum(4).connected_components()
[H_hyp(4), H_odd(4)]
sage: AbelianStratum(3,1).connected_components()
[H_c(3, 1)]
sage: AbelianStratum(2,2).connected_components()
[H_hyp(2, 2), H_odd(2, 2)]
sage: AbelianStratum(2,1,1).connected_components()
[H_c(2, 1, 1)]
sage: AbelianStratum(1,1,1,1).connected_components()
[H_c(1, 1, 1, 1)]
```

genus()

Returns the genus of the stratum.

OUTPUT:

integer – the genus

EXAMPLES:

```
sage: AbelianStratum(0).genus()
1
sage: AbelianStratum(1,1).genus()
2
sage: AbelianStratum(3,2,1).genus()
4
```

is_connected()

Tests if the strata is connected.

OUTPUT:

boolean – `True` if it is connected else `False`

EXAMPLES:

```
sage: AbelianStratum(2).is_connected()
True
sage: AbelianStratum(2).connected_components()
[H_hyp(2)]

sage: AbelianStratum(2,2).is_connected()
False
sage: AbelianStratum(2,2).connected_components()
[H_hyp(2, 2), H_odd(2, 2)]
```

nintervals()

Returns the number of intervals of any iet of the strata.

OUTPUT:

integer – the number of intervals for any associated iet

EXAMPLES:
```
sage: AbelianStratum(0).nintervals()
2
sage: AbelianStratum(0,0).nintervals()
3
sage: AbelianStratum(2).nintervals()
4
sage: AbelianStratum(1,1).nintervals()
5
```

sage.dynamics.flat_surfaces.strata.**CCA**
    alias of ConnectedComponentOfAbelianStratum

class sage.dynamics.flat_surfaces.strata.**ConnectedComponentOfAbelianStratum**(*parent*)
    Bases: sage.structure.sage_object.SageObject

Connected component of Abelian stratum.

> **Warning:** Internal class! Do not use directly!

TESTS:

Tests for outgoing marked separatrices:
```
sage: a = AbelianStratum(4,2,0,marked_separatrix='out')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_out_degree()
4
sage: a_even.representative().attached_out_degree()
4

sage: a = AbelianStratum(2,4,0,marked_separatrix='out')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_out_degree()
2
sage: a_even.representative().attached_out_degree()
2

sage: a = AbelianStratum(0,4,2,marked_separatrix='out')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_out_degree()
0
```

```
sage: a_even.representative().attached_out_degree()
0

sage: a = AbelianStratum(3,2,1,marked_separatrix='out')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_out_degree()
3

sage: a = AbelianStratum(2,3,1,marked_separatrix='out')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_out_degree()
2

sage: a = AbelianStratum(1,3,2,marked_separatrix='out')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_out_degree()
1
```

Tests for incoming separatrices:
```
sage: a = AbelianStratum(4,2,0,marked_separatrix='in')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_in_degree()
4
sage: a_even.representative().attached_in_degree()
4

sage: a = AbelianStratum(2,4,0,marked_separatrix='in')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_in_degree()
2
sage: a_even.representative().attached_in_degree()
2

sage: a = AbelianStratum(0,4,2,marked_separatrix='in')
sage: a_odd, a_even = a.connected_components()
sage: a_odd.representative().attached_in_degree()
0
sage: a_even.representative().attached_in_degree()
0

sage: a = AbelianStratum(3,2,1,marked_separatrix='in')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_in_degree()
3

sage: a = AbelianStratum(2,3,1,marked_separatrix='in')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_in_degree()
2

sage: a = AbelianStratum(1,3,2,marked_separatrix='in')
sage: a_c = a.connected_components()[0]
sage: a_c.representative().attached_in_degree()
1
```

genus()
    Returns the genus of the surfaces in this connected component.

    OUTPUT:

integer – the genus of the surface

EXAMPLES:
```
sage: a = AbelianStratum(6,4,2,0,0)
sage: c_odd, c_even = a.connected_components()
sage: c_odd.genus()
7
sage: c_even.genus()
7

sage: a = AbelianStratum([1]*8)
sage: c = a.connected_components()[0]
sage: c.genus()
5
```

nintervals()

Returns the number of intervals of the representative.

OUTPUT:

integer – the number of intervals in any representative

EXAMPLES:
```
sage: a = AbelianStratum(6,4,2,0,0)
sage: c_odd, c_even = a.connected_components()
sage: c_odd.nintervals()
18
sage: c_even.nintervals()
18

sage: a = AbelianStratum([1]*8)
sage: c = a.connected_components()[0]
sage: c.nintervals()
17
```

parent()

The stratum of this component

OUTPUT:

stratum - the stratum where this component leaves

EXAMPLES:
```
sage: p = iet.Permutation('a b','b a')
sage: c = p.connected_component()
sage: c.parent()
H(0)
```

rauzy_diagram(*reduced=True*)

Returns the Rauzy diagram associated to this connected component.

OUTPUT:

rauzy diagram – the Rauzy diagram associated to this stratum

EXAMPLES:
```
sage: c = AbelianStratum(0).connected_components()[0]
sage: r = c.rauzy_diagram()
```

representative (*reduced=True*, *alphabet=None*)
>   Returns the Zorich representative of this connected component.
>
>   Zorich constructs explicitly interval exchange transformations for each stratum in [Zor08].
>
>   INPUT:
>
>   •`reduced` - boolean (default: `True`): whether you obtain a reduced or labelled permutation
>
>   •`alphabet` - an alphabet or `None`: whether you want to specify an alphabet for your permutation
>
>   OUTPUT:
>
>   permutation – a permutation which lives in this component
>
>   EXAMPLES:
> ```
> sage: c = AbelianStratum(1,1,1,1).connected_components()[0]
> sage: print c
> H_c(1, 1, 1, 1)
> sage: p = c.representative(alphabet=range(9))
> sage: print p
> 0 1 2 3 4 5 6 7 8
> 4 3 2 5 8 7 6 1 0
> sage: p.connected_component()
> H_c(1, 1, 1, 1)
> ```

sage.dynamics.flat_surfaces.strata.**EvenCCA**
>   alias of `EvenConnectedComponentOfAbelianStratum`

class sage.dynamics.flat_surfaces.strata.**EvenConnectedComponentOfAbelianStratum** (*parent*)
>   Bases: `sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum`
>
>   Connected component of Abelian stratum with even spin structure.
>
> > **Warning:** Internal class! Do not use directly!
>
>   **representative** (*reduced=True*, *alphabet=None*)
> >   Returns the Zorich representative of this connected component.
> >
> >   Zorich constructs explicitly interval exchange transformations for each stratum in [Zor08].
> >
> >   EXAMPLES:
> > ```
> > sage: c = AbelianStratum(6).connected_components()[2]
> > sage: c
> > H_even(6)
> > sage: p = c.representative(alphabet=range(8))
> > sage: p
> > 0 1 2 3 4 5 6 7
> > 5 4 3 2 7 6 1 0
> > sage: p.connected_component()
> > H_even(6)
> >
> > sage: c = AbelianStratum(4,4).connected_components()[2]
> > sage: c
> > H_even(4, 4)
> > sage: p = c.representative(alphabet=range(11))
> > sage: p
> > 0 1 2 3 4 5 6 7 8 9 10
> > 5 4 3 2 6 8 7 10 9 1 0
> > sage: p.connected_component()
> > H_even(4, 4)
> > ```

sage.dynamics.flat_surfaces.strata.**HypCCA**
> alias of HypConnectedComponentOfAbelianStratum

class sage.dynamics.flat_surfaces.strata.**HypConnectedComponentOfAbelianStratum**(*parent*)
> Bases: sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum

> Hyperelliptic component of Abelian stratum.

> > **Warning:** Internal class! Do not use directly!

> **representative**(*reduced=True*, *alphabet=None*)
> > Returns the Zorich representative of this connected component.

> > Zorich constructs explicitly interval exchange transformations for each stratum in [Zor08].

> > INPUT:

> > > • reduced - boolean (defaut: True): whether you obtain a reduced or labelled permutation

> > > • alphabet - alphabet or None (defaut: None): whether you want to specify an alphabet for your representative

> > EXAMPLES:
> > ```
> > sage: c = AbelianStratum(0).connected_components()[0]
> > sage: c
> > H_hyp(0)
> > sage: p = c.representative(alphabet="01")
> > sage: p
> > 0 1
> > 1 0
> > sage: p.connected_component()
> > H_hyp(0)
> >
> > sage: c = AbelianStratum(0,0).connected_components()[0]
> > sage: c
> > H_hyp(0, 0)
> > sage: p = c.representative(alphabet="abc")
> > sage: p
> > a b c
> > c b a
> > sage: p.connected_component()
> > H_hyp(0, 0)
> >
> > sage: c = AbelianStratum(2).connected_components()[0]
> > sage: c
> > H_hyp(2)
> > sage: p = c.representative(alphabet="ABCD")
> > sage: p
> > A B C D
> > D C B A
> > sage: p.connected_component()
> > H_hyp(2)
> >
> > sage: c = AbelianStratum(1,1).connected_components()[0]
> > sage: c
> > H_hyp(1, 1)
> > sage: p = c.representative(alphabet="01234")
> > sage: p
> > 0 1 2 3 4
> > 4 3 2 1 0
> > ```


2.1. Strata of differentials on Riemann surfaces

```
sage: p.connected_component()
H_hyp(1, 1)
```

sage.dynamics.flat_surfaces.strata.**NonHypCCA**
> alias of NonHypConnectedComponentOfAbelianStratum

class sage.dynamics.flat_surfaces.strata.**NonHypConnectedComponentOfAbelianStratum**(*parent*)
> Bases: sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum

> Non hyperelliptic component of Abelian stratum.

> > **Warning:** Internal class! Do not use directly!

sage.dynamics.flat_surfaces.strata.**OddCCA**
> alias of OddConnectedComponentOfAbelianStratum

class sage.dynamics.flat_surfaces.strata.**OddConnectedComponentOfAbelianStratum**(*parent*)
> Bases: sage.dynamics.flat_surfaces.strata.ConnectedComponentOfAbelianStratum

> Connected component of an Abelian stratum with odd spin parity.

> > **Warning:** Internal class! Do not use directly!

> **representative**(*reduced=True*, *alphabet=None*)
> > Returns the Zorich representative of this connected component.

> > Zorich constructs explicitly interval exchange transformations for each stratum in [Zor08].

> > EXAMPLES:
> > ```
> > sage: a = AbelianStratum(6).connected_components()[1]
> > sage: print a.representative(alphabet=range(8))
> > 0 1 2 3 4 5 6 7
> > 3 2 5 4 7 6 1 0
> >
> > sage: a = AbelianStratum(4,4).connected_components()[1]
> > sage: print a.representative(alphabet=range(11))
> > 0 1 2 3 4 5 6 7 8 9 10
> > 3 2 5 4 6 8 7 10 9 1 0
> > ```

2.2 Strata of quadratic differentials on Riemann surfaces

class sage.dynamics.flat_surfaces.quadratic_strata.**QuadraticStratum**(**l*)
> Bases: sage.structure.sage_object.SageObject

> Stratum of quadratic differentials.

> **genus**()
> > Returns the genus.

> > EXAMPLES:
> > ```
> > sage: QuadraticStratum(-1,-1,-1,-1).genus()
> > 0
> > ```

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[BL08] Corentin Boissy and Erwan Lanneau, "Dynamics and geometry of the Rauzy-Veech induction for quadratic differentials" (arxiv:0710.5614) to appear in Ergodic Theory and Dynamical Systems

[DN90] Claude Danthony and Arnaldo Nogueira "Measured foliations on nonorientable surfaces", Annales scientifiques de l'Ecole Normale Superieure, Ser. 4, 23, no. 3 (1990) p 469-494

[N85] Arnaldo Nogueira, "Almost all Interval Exchange Transformations with Flips are Nonergodic" (Ergod. Th. & Dyn. Systems, Vol 5., (1985), 257-271

[R79] Gerard Rauzy, "Echanges d'intervalles et transformations induites", Acta Arith. 34, no. 3, 203-212, 1980

[V78] William Veech, "Interval exchange transformations", J. Analyse Math. 33, 222-272

[Z] Anton Zorich, "Generalized Permutation software" (http://perso.univ-rennes1.fr/anton.zorich)

[Yoc05] Jean-Christophe Yoccoz "Echange d'Intervalles", Cours au college de France

[MMY03] Jean-Christophe Yoccoz, Stefano Marmi and Pierre Moussa "On the cohomological equation for interval exchange maps", Arxiv math/0304469v1

[KonZor03] M. Kontsevich, A. Zorich "Connected components of the moduli space of Abelian differentials with prescripebd singularities" Invent. math. 153, 631-678 (2003)

[Lan08] E. Lanneau "Connected components of the strata of the moduli spaces of quadratic differentials", Annales sci. de l'ENS, serie 4, fascicule 1, 41, 1-56 (2008)

[Zor08] A. Zorich "Explicit Jenkins-Strebel representatives of all strata of Abelian and quadratic differentials", Journal of Modern Dynamics, vol. 2, no 1, 139-185 (2008) (http://www.math.psu.edu/jmd)

[ZS] Anton Zorich, "Generalized Permutation software" (http://perso.univ-rennes1.fr/anton.zorich/Software/software_en.html)

PYTHON MODULE INDEX

d

INDEX

A

C

D

E

F

G

H

S

T

V

W