Sage Reference Manual: Cell complexes and their homology

Release 7.5

The Sage Development Team

Jan 12, 2017

CONTENTS

Sage includes some tools for algebraic topology: from the algebraic side, chain complexes and their homology, and from the topological side, simplicial complexes, Δ-complexes, and cubical complexes. A class of generic cell complexes is also available, mainly for developers who want to use it as a base for other types of cell complexes.

CHAPTER

ONE

CHAIN COMPLEXES

AUTHORS:

- John H. Palmieri (2009-04)

This module implements bounded chain complexes of free R-modules, for any commutative ring R (although the interesting things, like homology, only work if R is the integers or a field).

Fix a ring R. A chain complex over R is a collection of R-modules $\{C_n\}$ indexed by the integers, with R-module maps $d_n : C_n \to C_{n+1}$ such that $d_{n+1} \circ d_n = 0$ for all n. The maps d_n are called *differentials*.

One can vary this somewhat: the differentials may decrease degree by one instead of increasing it: sometimes a chain complex is defined with $d_n : C_n \to C_{n-1}$ for each n. Indeed, the differentials may change dimension by any fixed integer.

Also, the modules may be indexed over an abelian group other than the integers, e.g., $\mathbf{Z}^m$ for some integer $m \geq 1$, in which case the differentials may change the grading by any element of that grading group. The elements of the grading group are generally called degrees, so C_n is the module in degree n and so on.

In this implementation, the ring R must be commutative and the modules C_n must be free R-modules. As noted above, homology calculations will only work if the ring R is either $\mathbf{Z}$ or a field. The modules may be indexed by any free abelian group. The differentials may increase degree by 1 or decrease it, or indeed change it by any fixed amount: this is controlled by the `degree_of_differential` parameter used in defining the chain complex.

`sage.homology.chain_complex.` **`ChainComplex`** (*data=None*, *base_ring=None*, *grading_group=None*, *degree_of_differential=1*, *degree=1*, *check=True*)

Define a chain complex.

INPUT:

- `data` – the data defining the chain complex; see below for more details.

The following keyword arguments are supported:

- `base_ring` – a commutative ring (optional), the ring over which the chain complex is defined. If this is not specified, it is determined by the data defining the chain complex.
- `grading_group` – a additive free abelian group (optional, default `ZZ`), the group over which the chain complex is indexed.
- `degree_of_differential` – element of grading_group (optional, default `1`). The degree of the differential.
- `degree` – alias for `degree_of_differential` .
- `check` – boolean (optional, default `True`). If `True` , check that each consecutive pair of differentials are composable and have composite equal to zero.

OUTPUT:

A chain complex.

Warning: Right now, homology calculations will only work if the base ring is either **Z** or a field, so please take this into account when defining a chain complex.

Use data to define the chain complex. This may be in any of the following forms.

1. a dictionary with integers (or more generally, elements of grading_group) for keys, and with `data[n]` a matrix representing (via left multiplication) the differential coming from degree n. (Note that the shape of the matrix then determines the rank of the free modules C_n and C_{n+d}.)
2. a list/tuple/iterable of the form $[C_0, d_0, C_1, d_1, C_2, d_2, ...]$, where each C_i is a free module and each d_i is a matrix, as above. This only makes sense if `grading_group` is **Z** and `degree` is 1.
3. a list/tuple/iterable of the form $[r_0, d_0, r_1, d_1, r_2, d_2, \ldots]$, where r_i is the rank of the free module C_i and each d_i is a matrix, as above. This only makes sense if `grading_group` is **Z** and `degree` is 1.
4. a list/tuple/iterable of the form $[d_0, d_1, d_2, \ldots]$ where each d_i is a matrix, as above. This only makes sense if `grading_group` is **Z** and `degree` is 1.

Note: In fact, the free modules C_i in case 2 and the ranks r_i in case 3 are ignored: only the matrices are kept, and from their shapes, the ranks of the modules are determined. (Indeed, if `data` is a list or tuple, then any element which is not a matrix is discarded; thus the list may have any number of different things in it, and all of the non-matrices will be ignored.) No error checking is done to make sure, for instance, that the given modules have the appropriate ranks for the given matrices. However, as long as `check` is True, the code checks to see if the matrices are composable and that each appropriate composite is zero.

If the base ring is not specified, then the matrices are examined to determine a ring over which they are all naturally defined, and this becomes the base ring for the complex. If no such ring can be found, an error is raised. If the base ring is specified, then the matrices are converted automatically to this ring when defining the chain complex. If some matrix cannot be converted, then an error is raised.

EXAMPLES:

```
sage: ChainComplex()
Trivial chain complex over Integer Ring

sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C
Chain complex with at most 2 nonzero terms over Integer Ring

sage: m = matrix(ZZ, 2, 2, [0, 1, 0, 0])
sage: D = ChainComplex([m, m], base_ring=GF(2)); D
Chain complex with at most 3 nonzero terms over Finite Field of size 2
sage: D == loads(dumps(D))
True
sage: D.differential(0)==m, m.is_immutable(), D.differential(0).is_immutable()
(True, False, True)
```

Note that when a chain complex is defined in Sage, new differentials may be created: every nonzero module in the chain complex must have a differential coming from it, even if that differential is zero:

```
sage: IZ = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: IZ.differential()  # the differentials in the chain complex
```

```
{-1: [], 0: [1], 1: []}
sage: IZ.differential(1).parent()
Full MatrixSpace of 0 by 1 dense matrices over Integer Ring
sage: mat = ChainComplex({0: matrix(ZZ, 3, 4)}).differential(1)
sage: mat.nrows(), mat.ncols()
(0, 3)
```

Defining the base ring implicitly:

```
sage: ChainComplex([matrix(QQ, 3, 1), matrix(ZZ, 4, 3)])
Chain complex with at most 3 nonzero terms over Rational Field
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1), matrix(ZZ, 4, 3)])
Chain complex with at most 3 nonzero terms over Finite Field in a of size 5^3
```

If the matrices are defined over incompatible rings, an error results:

```
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1), matrix(QQ, 4, 3)])
Traceback (most recent call last):
...
TypeError: unable to find a common ring for all elements
```

If the base ring is given explicitly but is not compatible with the matrices, an error results:

```
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1)], base_ring=QQ)
Traceback (most recent call last):
...
TypeError: unable to convert 0 to a rational
```

class sage.homology.chain_complex. **ChainComplex_class** (*grading_group*, *degree_of_differential*, *base_ring*, *differentials*)

Bases: sage.structure.parent.Parent

See *ChainComplex()* for full documentation.

The differentials are required to be in the following canonical form:

- All differentials that are not 0×0 must be specified (even if they have zero rows or zero columns), and
- Differentials that are 0×0 must not be specified.
- Immutable matrices over the base_ring

This and more is ensured by the assertions in the constructor. The *ChainComplex()* factory function must ensure that only valid input is passed.

EXAMPLES:

```
sage: C = ChainComplex(); C
Trivial chain complex over Integer Ring

sage: D = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: D
Chain complex with at most 2 nonzero terms over Integer Ring
```

Element
alias of *Chain_class*

betti (*deg=None*, *base_ring=None*)
The Betti number the chain complex.

That is, write the homology in this degree as a direct sum of a free module and a torsion module; the Betti number is the rank of the free summand.

INPUT:

- `deg` – an element of the grading group for the chain complex or None (default `None`); if `None` , then return every Betti number, as a dictionary indexed by degree, or if an element of the grading group, then return the Betti number in that degree
- `base_ring` – a commutative ring (optional, default is the base ring for the chain complex); compute homology with these coefficients – must be either the integers or a field

OUTPUT:

The Betti number in degree `deg` – the rank of the free part of the homology module in this degree.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.betti(0)
2
sage: [C.betti(n) for n in range(5)]
[2, 1, 0, 0, 0]
sage: C.betti()
{0: 2, 1: 1}

sage: D = ChainComplex({0:matrix(GF(5), [[3, 1],[1, 2]])})
sage: D.betti()
{0: 1, 1: 1}
```

cartesian_product (**factors*, ***kwds*)

Return the direct sum (Cartesian product) of `self` with `D` .

Let C and D be two chain complexes with differentials ∂_C and ∂_D, respectively, of the same degree (so they must also have the same grading group). The direct sum $S = C \oplus D$ is a chain complex given by $S_i = C_i \oplus D_i$ with differential $\partial = \partial_C \oplus \partial_D$.

INPUT:

- `subdivide` – (default: `False`) whether to subdivide the the differential matrices

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: C = ChainComplex([matrix([[-y],[x]]), matrix([[x, y]])])
sage: D = ChainComplex([matrix([[x-y]]), matrix([[0], [0]])])
sage: ascii_art(C.cartesian_product(D))
            [x y 0]       [   -y     0]
            [0 0 0]       [    x     0]
            [0 0 0]       [    0 x - y]
 0 <-- C_2 <-------- C_1 <-------------- C_0 <-- 0

sage: D = ChainComplex({1:matrix([[x-y]]), 4:matrix([[x], [y]])})
sage: ascii_art(D)
            [x]
            [y]                     [x - y]
 0 <-- C_5 <---- C_4 <-- 0 <-- C_2 <-------- C_1 <-- 0
sage: ascii_art(cartesian_product([C, D]))
                                                        [-y]
            [x]                     [    x     y     0]  [ x]
            [y]                     [    0     0 x - y]  [ 0]
 0 <-- C_5 <---- C_4 <-- 0 <-- C_2 <------------------- C_1 <----- C_0 <-- 0
```

The degrees of the differentials must agree:

```
sage: C = ChainComplex({1:matrix([[x]])}, degree_of_differential=-1)
sage: D = ChainComplex({1:matrix([[x]])}, degree_of_differential=1)
sage: C.cartesian_product(D)
Traceback (most recent call last):
...
ValueError: the degrees of the differentials must match
```

TESTS:

```
sage: C = ChainComplex({2:matrix([[-1],[2]]), 1:matrix([[2, 1]])},
....:                  degree_of_differential=-1)
sage: ascii_art(C.cartesian_product(C, subdivide=True))
                            [-1| 0]
                            [ 2| 0]
            [2 1|0 0]       [--+--]
            [---+---]       [ 0|-1]
            [0 0|2 1]       [ 0| 2]
 0 <-- C_0 <---------- C_1 <-------- C_2 <-- 0
```

```
sage: R.<x,y,z> = QQ[]
sage: C1 = ChainComplex({1:matrix([[x]])})
sage: C2 = ChainComplex({1:matrix([[y]])})
sage: C3 = ChainComplex({1:matrix([[z]])})
sage: ascii_art(cartesian_product([C1, C2, C3]))
            [x 0 0]
            [0 y 0]
            [0 0 z]
 0 <-- C_2 <-------- C_1 <-- 0
sage: ascii_art(C1.cartesian_product([C2, C3], subdivide=True))
            [x|0|0]
            [-+-+-]
            [0|y|0]
            [-+-+-]
            [0|0|z]
 0 <-- C_2 <-------- C_1 <-- 0
```

```
sage: R.<x> = ZZ[]
sage: G = AdditiveAbelianGroup([0,7])
sage: d = {G(vector([1,1])):matrix([[x]])}
sage: C = ChainComplex(d, grading_group=G, degree=G(vector([2,1])))
sage: ascii_art(C.cartesian_product(C))
                [x 0]
                [0 x]
 0 <-- C_(3, 2) <------ C_(1, 1) <-- 0
```

degree_of_differential ( )

Return the degree of the differentials of the complex

OUTPUT:

An element of the grading group.

EXAMPLES:

```
sage: D = ChainComplex({0: matrix(ZZ, 2, 2, [1,0,0,2])})
sage: D.degree_of_differential()
1
```

differential (*dim=None*)

The differentials which make up the chain complex.

INPUT:

- `dim` – element of the grading group (optional, default `None`); if this is `None` , return a dictionary of all of the differentials, or if this is a single element, return the differential starting in that dimension

OUTPUT:

Either a dictionary of all of the differentials or a single differential (i.e., a matrix).

EXAMPLES:

```
sage: D = ChainComplex({0: matrix(ZZ, 2, 2, [1,0,0,2])})
sage: D.differential()
{-1: [], 0: [1 0]
 [0 2], 1: []}
sage: D.differential(0)
[1 0]
[0 2]
sage: C = ChainComplex({0: identity_matrix(ZZ, 40)})
sage: C.differential()
{-1: 40 x 0 dense matrix over Integer Ring,
 0: 40 x 40 dense matrix over Integer Ring,
 1: []}
```

dual ( )

The dual chain complex to `self` .

Since all modules in `self` are free of finite rank, the dual in dimension n is isomorphic to the original chain complex in dimension n, and the corresponding boundary matrix is the transpose of the matrix in the original complex. This converts a chain complex to a cochain complex and vice versa.

EXAMPLES:

```
sage: C = ChainComplex({2: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.degree_of_differential()
1
sage: C.differential(2)
[3 0 0]
[0 0 0]
sage: C.dual().degree_of_differential()
-1
sage: C.dual().differential(3)
[3 0]
[0 0]
[0 0]
```

free_module (*degree=None*)

Return the free module at fixed `degree` , or their sum.

INPUT:

- `degree` – an element of the grading group or `None` (default).

OUTPUT:

The free module C_n at the given degree n. If the degree is not specified, the sum $\bigoplus C_n$ is returned.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0]), 1:␣
↪matrix(ZZ, [[0, 1]])})
sage: C.free_module()
Ambient free module of rank 6 over the principal ideal domain Integer Ring
sage: C.free_module(0)
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: C.free_module(1)
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: C.free_module(2)
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

free_module_rank (*degree*)

Return the rank of the free module at the given `degree` .

INPUT:

- `degree` – an element of the grading group

OUTPUT:

Integer. The rank of the free module C_n at the given degree n.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0]), 1:␣
↪matrix(ZZ, [[0, 1]])})
sage: [C.free_module_rank(i) for i in range(-2, 5)]
[0, 0, 3, 2, 1, 0, 0]
```

grading_group ( )

Return the grading group.

OUTPUT:

The discrete abelian group that indexes the individual modules of the complex. Usually **Z**.

EXAMPLES:

```
sage: G = AdditiveAbelianGroup([0, 3])
sage: C = ChainComplex(grading_group=G, degree=G(vector([1,2])))
sage: C.grading_group()
Additive abelian group isomorphic to Z + Z/3
sage: C.degree_of_differential()
(1, 2)
```

homology (*deg=None*, *base_ring=None*, *generators=False*, *verbose=False*, *algorithm='pari'*)

The homology of the chain complex.

INPUT:

- `deg` – an element of the grading group for the chain complex (default: `None`); the degree in which to compute homology – if this is `None` , return the homology in every degree in which the chain complex is possibly nonzero.
- `base_ring` – a commutative ring (optional, default is the base ring for the chain complex); must be either the integers **Z** or a field
- `generators` – boolean (optional, default `False`); if `True` , return generators for the homology groups along with the groups. See trac ticket #6100

- `verbose` - boolean (optional, default `False`); if `True`, print some messages as the homology is computed
- `algorithm` - string (optional, default `'pari'`); the options are:
  - `'auto'`
  - `'chomp'`
  - `'dhsw'`
  - `'pari'`
  - `'no_chomp'`

see below for descriptions

OUTPUT:

If the degree is specified, the homology in degree `deg`. Otherwise, the homology in every dimension as a dictionary indexed by dimension.

ALGORITHM:

If `algorithm` is set to `'auto'`, then use CHomP if available. CHomP is available at the web page http://chomp.rutgers.edu/. It is also an optional package for Sage. If `algorithm` is `chomp`, always use chomp.

CHomP computes homology, not cohomology, and only works over the integers or finite prime fields. Therefore if any of these conditions fails, or if CHomP is not present, or if `algorithm` is set to 'no_chomp', go to plan B: if `self` has a `_homology` method – each simplicial complex has this, for example – then call that. Such a method implements specialized algorithms for the particular type of cell complex.

Otherwise, move on to plan C: compute the chain complex of `self` and compute its homology groups. To do this: over a field, just compute ranks and nullities, thus obtaining dimensions of the homology groups as vector spaces. Over the integers, compute Smith normal form of the boundary matrices defining the chain complex according to the value of `algorithm`. If `algorithm` is `'auto'` or `'no_chomp'`, then for each relatively small matrix, use the standard Sage method, which calls the Pari package. For any large matrix, reduce it using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm [DHSW2003]: see *dhsw_snf()* for details.

Finally, `algorithm` may also be `'pari'` or `'dhsw'`, which forces the named algorithm to be used regardless of the size of the matrices and regardless of whether CHomP is available.

As of this writing, `'pari'` is the fastest standard option. The optional CHomP package may be better still.

Warning: This only works if the base ring is the integers or a field. Other values will return an error.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.homology()
{0: Z x Z, 1: Z x C3}
sage: C.homology(deg=1, base_ring = GF(3))
Vector space of dimension 2 over Finite Field of size 3
sage: D = ChainComplex({0: identity_matrix(ZZ, 4), 4: identity_matrix(ZZ, 30)}
↪)
sage: D.homology()
{0: 0, 1: 0, 4: 0, 5: 0}
```

Generators: generators are given as a list of cycles, each of which is an element in the appropriate free module, and hence is represented as a vector:

```
sage: C.homology(1, generators=True)  # optional - CHomP
(Z x C3, [(0, 1), (1, 0)])
```

Tests for trac ticket #6100, the Klein bottle with generators:

```
sage: d0 = matrix(ZZ, 0,1)
sage: d1 = matrix(ZZ, 1,3, [[0,0,0]])
sage: d2 = matrix(ZZ, 3,2, [[1,1], [1,-1], [-1,1]])
sage: C_k = ChainComplex({0:d0, 1:d1, 2:d2}, degree=-1)
sage: C_k.homology(generators=true)   # optional - CHomP
{0: (Z, [(1)]), 1: (Z x C2, [(0, 0, 1), (0, 1, -1)]), 2: 0}
```

From a torus using a field:

```
sage: T = simplicial_complexes.Torus()
sage: C_t = T.chain_complex()
sage: C_t.homology(base_ring=QQ, generators=True)
{0: [(Vector space of dimension 1 over Rational Field,
   Chain(0:(0, 0, 0, 0, 0, 0, 1)))],
 1: [(Vector space of dimension 1 over Rational Field,
   Chain(1:(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 1))),
  (Vector space of dimension 1 over Rational Field,
   Chain(1:(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 1, 0, -1,␣
↪0)))],
 2: [(Vector space of dimension 1 over Rational Field,
   Chain(2:(1, -1, 1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1)))]}
```

nonzero_degrees ( )

Return the degrees in which the module is non-trivial.

See also *ordered_degrees()* .

OUTPUT:

The tuple containing all degrees n (grading group elements) such that the module C_n of the chain is non-trivial.

EXAMPLES:

```
sage: one = matrix(ZZ, [[1]])
sage: D = ChainComplex({0: one, 2: one, 6:one})
sage: ascii_art(D)
           [1]                                [1]       [0]       [1]
0 <-- C_7 <---- C_6 <-- 0  ...  0 <-- C_3 <---- C_2 <---- C_1 <---- C_0 <-- 0
sage: D.nonzero_degrees()
(0, 1, 2, 3, 6, 7)
```

ordered_degrees (*start=None*, *exclude_first=False*)

Sort the degrees in the order determined by the differential

INPUT:

- `start` – (default: `None`) a degree (element of the grading group) or `None`
- `exclude_first` – boolean (optional; default: `False`); whether to exclude the lowest degree – this is a handy way to just get the degrees of the non-zero modules, as the domain of the first differential is zero.

OUTPUT:

If `start` has been specified, the longest tuple of degrees

- containing `start` (unless `start` would be the first and `exclude_first=True`),
- in ascending order relative to *`degree_of_differential()`*, and
- such that none of the corresponding differentials are 0×0.

If `start` has not been specified, a tuple of such tuples of degrees. One for each sequence of non-zero differentials. They are returned in sort order.

EXAMPLES:

```
sage: one = matrix(ZZ, [[1]])
sage: D = ChainComplex({0: one, 2: one, 6:one})
sage: ascii_art(D)
            [1]                                 [1]        [0]        [1]
0 <-- C_7 <---- C_6 <-- 0  ...  0 <-- C_3 <---- C_2 <---- C_1 <---- C_0 <-- 0
sage: D.ordered_degrees()
((-1, 0, 1, 2, 3), (5, 6, 7))
sage: D.ordered_degrees(exclude_first=True)
((0, 1, 2, 3), (6, 7))
sage: D.ordered_degrees(6)
(5, 6, 7)
sage: D.ordered_degrees(5, exclude_first=True)
(6, 7)
```

random_element ( )

Return a random element.

EXAMPLES:

```
sage: D = ChainComplex({0: matrix(ZZ, 2, 2, [1,0,0,2])})
sage: D.random_element()    # random output
Chain with 1 nonzero terms over Integer Ring
```

rank (*degree*, *ring=None*)

Return the rank of a differential

INPUT:

- `degree` – an element δ of the grading group. Which differential d_δ we want to know the rank of
- `ring` – (optional) a commutative ring S; if specified, the rank is computed after changing to this ring

OUTPUT:

The rank of the differential $d_\delta \otimes_R S$, where R is the base ring of the chain complex.

EXAMPLES:

```
sage: C = ChainComplex({0:matrix(ZZ, [[2]])})
sage: C.differential(0)
[2]
sage: C.rank(0)
1
sage: C.rank(0, ring=GF(2))
0
```

shift (*n=1*)

Shift this chain complex n times.

INPUT:

- `n` – an integer (optional, default 1)

The *shift* operation is also sometimes called *translation* or *suspension*.

To shift a chain complex by n, shift its entries up by n (if it is a chain complex) or down by n (if it is a cochain complex); that is, shifting by 1 always shifts in the opposite direction of the differential. In symbols, if C is a chain complex and $C[n]$ is its n-th shift, then $C[n]_j = C_{j-n}$. The differential in the shift $C[n]$ is obtained by multiplying each differential in C by $(-1)^n$.

Caveat: different sources use different conventions for shifting: what we call $C[n]$ might be called $C[-n]$ in some places. See for example. https://ncatlab.org/nlab/show/suspension+of+a+chain+complex (which uses $C[n]$ as we do but acknowledges $C[-n]$) or 1.2.8 in [Wei1994] (which uses $C[-n]$).

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1).chain_complex()
sage: S1.shift(1).differential(2) == -S1.differential(1)
True
sage: S1.shift(2).differential(3) == S1.differential(1)
True
sage: S1.shift(3).homology(4)
Z
```

For cochain complexes, shifting goes in the other direction. Topologically, this makes sense if we grade the cochain complex for a space negatively:

```
sage: T = simplicial_complexes.Torus()
sage: co_T = T.chain_complex()._flip_()
sage: co_T.homology()
{-2: Z, -1: Z x Z, 0: Z}
sage: co_T.degree_of_differential()
1
sage: co_T.shift(2).homology()
{-4: Z, -3: Z x Z, -2: Z}
```

You can achieve the same result by tensoring (on the left, to get the signs right) with a rank one free module in degree `-n * deg`, if `deg` is the degree of the differential:

```
sage: C = ChainComplex({-2: matrix(ZZ, 0, 1)})
sage: C.tensor(co_T).homology()
{-4: Z, -3: Z x Z, -2: Z}
```

tensor (**factors*, ***kwds*)

Return the tensor product of `self` with `D`.

Let C and D be two chain complexes with differentials ∂_C and ∂_D, respectively, of the same degree (so they must also have the same grading group). The tensor product $S = C \otimes D$ is a chain complex given by

$$S_i = \bigoplus_{a+b=i} C_a \otimes D_b$$

with differential

$$\partial(x \otimes y) = \partial_C x \otimes y + (-1)^{|a| \cdot |\partial_D|} x \otimes \partial_D y$$

for $x \in C_a$ and $y \in D_b$, where $|a|$ is the degree of a and $|\partial_D|$ is the degree of ∂_D.

Warning: If the degree of the differential is even, then this may not result in a valid chain complex.

INPUT:

- `subdivide` – (default: `False`) whether to subdivide the the differential matrices

Todo

Make subdivision work correctly on multiple factors.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: C1 = ChainComplex({1:matrix([[x]])}, degree_of_differential=-1)
sage: C2 = ChainComplex({1:matrix([[y]])}, degree_of_differential=-1)
sage: C3 = ChainComplex({1:matrix([[z]])}, degree_of_differential=-1)
sage: ascii_art(C1.tensor(C2))
                        [ x]
            [y x]       [-y]
 0 <-- C_0 <------ C_1 <----- C_2 <-- 0
sage: ascii_art(C1.tensor(C2).tensor(C3))
                          [ y  x  0]       [ x]
                          [-z  0  x]       [-y]
            [z y x]       [ 0 -z -y]       [ z]
 0 <-- C_0 <-------- C_1 <---------- C_2 <----- C_3 <-- 0
```

```
sage: C = ChainComplex({2:matrix([[-y],[x]]), 1:matrix([[x, y]])},
....:                  degree_of_differential=-1); ascii_art(C)
                        [-y]
            [x y]       [ x]
 0 <-- C_0 <------ C_1 <----- C_2 <-- 0
sage: T = C.tensor(C)
sage: T.differential(1)
[x y x y]
sage: T.differential(2)
[-y  x  0  y  0  0]
[ x  0  x  0  y  0]
[ 0 -x -y  0  0 -y]
[ 0  0  0 -x -y  x]
sage: T.differential(3)
[ x  y  0  0]
[ y  0 -y  0]
[-x  0  0 -y]
[ 0  y  x  0]
[ 0 -x  0  x]
[ 0  0  x  y]
sage: T.differential(4)
[-y]
[ x]
[-y]
[ x]
```

The degrees of the differentials must agree:

```
sage: C1p = ChainComplex({1:matrix([[x]])}, degree_of_differential=1)
sage: C1.tensor(C1p)
```

```
Traceback (most recent call last):
...
ValueError: the degrees of the differentials must match
```

TESTS:

```
sage: R.<x,y,z> = QQ[]
sage: C1 = ChainComplex({1:matrix([[x]])})
sage: C2 = ChainComplex({1:matrix([[y]])})
sage: C3 = ChainComplex({1:matrix([[z]])})
sage: ascii_art(tensor([C1, C2, C3]))
                          [-y -z  0]       [ z]
                          [ x  0 -z]       [-y]
            [x y z]       [ 0  x  y]       [ x]
 0 <-- C_6 <-------- C_5 <---------- C_4 <----- C_3 <-- 0
```

```
sage: R.<x,y> = ZZ[]
sage: G = AdditiveAbelianGroup([0,7])
sage: d1 = {G(vector([1,1])):matrix([[x]])}
sage: C1 = ChainComplex(d1, grading_group=G, degree=G(vector([2,1])))
sage: d2 = {G(vector([3,0])):matrix([[y]])}
sage: C2 = ChainComplex(d2, grading_group=G, degree=G(vector([2,1])))
sage: ascii_art(C1.tensor(C2))
                                    [y]
                  [ x -y]           [x]
 0 <-- C_(8, 3) <-------- C_(6, 2) <---- C_(4, 1) <-- 0
```

Check that trac ticket #21760 is fixed:

```
sage: C = ChainComplex({0: matrix(ZZ, 0, 2)}, degree=-1)
sage: ascii_art(C)
 0 <-- C_0 <-- 0
sage: T = C.tensor(C)
sage: ascii_art(T)
 0 <-- C_0 <-- 0
sage: T.free_module_rank(0)
4
```

torsion_list (*max_prime*, *min_prime=2*)

Look for torsion in this chain complex by computing its mod p homology for a range of primes p.

INPUT:

- `max_prime` – prime number; search for torsion mod p for all p strictly less than this number
- `min_prime` – prime (optional, default 2); search for torsion mod p for primes at least as big as this

Return a list of pairs (p, d) where p is a prime at which there is torsion and d is a list of dimensions in which this torsion occurs.

The base ring for the chain complex must be the integers; if not, an error is raised.

ALGORITHM:

let C denote the chain complex. Let P equal `max_prime` . Compute the mod P homology of C, and use this as the base-line computation: the assumption is that this is isomorphic to the integral homology tensored with $\mathbf{F}_P$. Then compute the mod p homology for a range of primes p, and record whenever the answer differs from the base-line answer.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.homology()
{0: Z x Z, 1: Z x C3}
sage: C.torsion_list(11)
[(3, [1])]
sage: C = ChainComplex([matrix(ZZ, 1, 1, [2]), matrix(ZZ, 1, 1), matrix(1, 1,␣
↪[3])])
sage: C.homology(1)
C2
sage: C.homology(3)
C3
sage: C.torsion_list(5)
[(2, [1]), (3, [3])]
```

class sage.homology.chain_complex. **Chain_class** (*parent*, *vectors*, *check=True*)

Bases: sage.structure.element.ModuleElement

A Chain in a Chain Complex

A chain is collection of module elements for each module C_n of the chain complex (C_n, d_n). There is no restriction on how the differentials d_n act on the elements of the chain.

Note: You must use the chain complex to construct chains.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])}, base_ring=GF(7))
sage: C.category()
Category of chain complexes over Finite Field of size 7
```

TESTS:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: c = C({0:vector([0, 1, 2]), 1:vector([3, 4])})
sage: TestSuite(c).run()
```

is_boundary ( )

Return whether the chain is a boundary.

OUTPUT:

Boolean. Whether the elements of the chain are in the image of the differentials.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: c = C({0:vector([0, 1, 2]), 1:vector([3, 4])})
sage: c.is_boundary()
False
sage: z3 = C({1:(1, 0)})
sage: z3.is_cycle()
True
sage: (2*z3).is_boundary()
False
sage: (3*z3).is_boundary()
True
```

is_cycle ( )

Return whether the chain is a cycle.

OUTPUT:

Boolean. Whether the elements of the chain are in the kernel of the differentials.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: c = C({0:vector([0, 1, 2]), 1:vector([3, 4])})
sage: c.is_cycle()
True
```

vector (*degree*)

Return the free module element in `degree` .

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: c = C({0:vector([1, 2, 3]), 1:vector([4, 5])})
sage: c.vector(0)
(1, 2, 3)
sage: c.vector(1)
(4, 5)
sage: c.vector(2)
()
```

CHAPTER

TWO

CHAINS AND COCHAINS

This module implements formal linear combinations of cells of a given cell complex (*Chains*) and their dual (*Cochains*). It is closely related to the *sage.homology.chain_complex* module. The main differences are that chains and cochains here are of homogeneous dimension only, and that they reference their cell complex.

class sage.homology.chains. **CellComplexReference** (*cell_complex*, *degree*, *cells=None*)

Bases: object

Auxiliary base class for chains and cochains

INPUT:

- cell_complex – The cell complex to reference
- degree – integer. The degree of the (co)chains
- cells – tuple of cells or None . Does not necessarily have to be the cells in the given degree, for computational purposes this could also be any collection that is in one-to-one correspondence with the cells. If None , the cells of the complex in the given degree are used.

EXAMPLES:

```
sage: X = simplicial_complexes.Simplex(2)
sage: from sage.homology.chains import CellComplexReference
sage: c = CellComplexReference(X, 1)
sage: c.cell_complex() is X
True
```

cell_complex ( )

Return the underlying cell complex

OUTPUT:

A cell complex.

EXAMPLES:

```
sage: X = simplicial_complexes.Simplex(2)
sage: X.n_chains(1).cell_complex() is X
True
```

degree ( )

Return the dimension of the cells

OUTPUT:

Integer. The dimension of the cells.

EXAMPLES:

```
sage: X = simplicial_complexes.Simplex(2)
sage: X.n_chains(1).degree()
1
```

class sage.homology.chains. **Chains** (*cell_complex*, *degree*, *cells=None*, *base_ring=None*)
Bases: *sage.homology.chains.CellComplexReference*, sage.combinat.free_module.CombinatorialI

Class for the free module of chains in a given degree.

INPUT:

- n_cells – tuple of n-cells, which thus forms a basis for this module
- base_ring – optional (default **Z**)

One difference between chains and cochains is notation. In a simplicial complex, for example, a simplex (0,1,2) is written as "(0,1,2)" in the group of chains but as "\chi_(0,1,2)" in the group of cochains.

Also, since the free modules of chains and cochains are dual, there is a pairing $\langle c, z \rangle$, sending a cochain c and a chain z to a scalar.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: C_2 = S2.n_chains(1)
sage: C_2_co = S2.n_chains(1, cochains=True)
sage: x = C_2.basis()[Simplex((0,2))]
sage: y = C_2.basis()[Simplex((1,3))]
sage: z = x+2*y
sage: a = C_2_co.basis()[Simplex((1,3))]
sage: b = C_2_co.basis()[Simplex((0,3))]
sage: c = 3*a-2*b
sage: z
(0, 2) + 2*(1, 3)
sage: c
-2*\chi_(0, 3) + 3*\chi_(1, 3)
sage: c.eval(z)
6
```

class **Element** (*M*, *x*)
Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

boundary ( )
Return the boundary of the chain

OUTPUT:

The boundary as a chain in one degree lower.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ)
sage: from sage.homology.cubical_complex import Cube
sage: chain = C1(Cube([[1, 1], [0, 1]])) - 2 * C1(Cube([[0, 1], [0, 0]]))
sage: chain
-2*[0,1] x [0,0] + [1,1] x [0,1]
sage: chain.boundary()
2*[0,0] x [0,0] - 3*[1,1] x [0,0] + [1,1] x [1,1]
```

is_boundary ( )

Test whether the chain is a boundary

OUTPUT:

Boolean. Whether the chain is the *boundary()* of a chain in one degree higher.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ)
sage: from sage.homology.cubical_complex import Cube
sage: chain = C1(Cube([[1, 1], [0, 1]])) - C1(Cube([[0, 1], [0, 0]]))
sage: chain.is_boundary()
False
```

TESTS:

```
sage: chain.is_coboundary()
Traceback (most recent call last):
...
AttributeError: 'Chains_with_category.element_class' object has no␣
↪attribute 'is_coboundary'
```

is_cycle ( )

Test whether the chain is a cycle

OUTPUT:

Boolean. Whether the *boundary()* vanishes.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ)
sage: from sage.homology.cubical_complex import Cube
sage: chain = C1(Cube([[1, 1], [0, 1]])) - C1(Cube([[0, 1], [0, 0]]))
sage: chain.is_cycle()
False
```

TESTS:

```
sage: chain.is_cocycle()
Traceback (most recent call last):
...
AttributeError: 'Chains_with_category.element_class' object has no␣
↪attribute 'is_cocycle'
```

to_complex ( )

Return the corresponding chain complex element

OUTPUT:

An element of the chain complex, see *sage.homology.chain_complex* .

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ)
sage: from sage.homology.cubical_complex import Cube
sage: chain = C1(Cube([[1, 1], [0, 1]]))
sage: chain.to_complex()
Chain(1:(0, 0, 0, 1))
sage: ascii_art(_)
   d_0  [0]  d_1  [0]  d_2       d_3
0 <---- [0] <---- [0] <---- [0] <---- 0
        [0]       [0]
        [0]       [1]
```

Chains. **chain_complex** ( )

Return the chain complex.

OUTPUT:

Chain complex, see *sage.homology.chain_complex* .

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: CC = square.n_chains(2, QQ).chain_complex(); CC
Chain complex with at most 3 nonzero terms over Rational Field
sage: ascii_art(CC)
            [ 0  0  1  1]       [-1]
            [ 0  1  0 -1]       [ 1]
            [ 1  0 -1  0]       [-1]
            [-1 -1  0  0]       [ 1]
 0 <-- C_0 <-------------- C_1 <----- C_2 <-- 0
```

Chains. **dual** ( )

Return the cochains.

OUTPUT:

The cochains of the same cells with the same base ring.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: chains = square.n_chains(1, ZZ);  chains
Free module generated by {[0,0] x [0,1], [0,1] x [0,0], [0,1] x [1,1], [1,1]␣
 ↪x [0,1]} over Integer Ring
sage: chains.dual()
Free module generated by {[0,0] x [0,1], [0,1] x [0,0], [0,1] x [1,1], [1,1]␣
 ↪x [0,1]} over Integer Ring
sage: type(chains)
<class 'sage.homology.chains.Chains_with_category'>
sage: type(chains.dual())
<class 'sage.homology.chains.Cochains_with_category'>
```

class sage.homology.chains. **Cochains** (*cell_complex*, *degree*, *cells=None*, *base_ring=None*)

Bases: *sage.homology.chains.CellComplexReference*, sage.combinat.free_module.CombinatorialI

Class for the free module of cochains in a given degree.

INPUT:

- `n_cells` – tuple of n-cells, which thus forms a basis for this module
- `base_ring` – optional (default $\mathbf{Z}$)

One difference between chains and cochains is notation. In a simplicial complex, for example, a simplex `(0,1,2)` is written as "(0,1,2)" in the group of chains but as "\chi_(0,1,2)" in the group of cochains.

Also, since the free modules of chains and cochains are dual, there is a pairing $\langle c, z \rangle$, sending a cochain c and a chain z to a scalar.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: C_2 = S2.n_chains(1)
sage: C_2_co = S2.n_chains(1, cochains=True)
sage: x = C_2.basis()[Simplex((0,2))]
sage: y = C_2.basis()[Simplex((1,3))]
sage: z = x+2*y
sage: a = C_2_co.basis()[Simplex((1,3))]
sage: b = C_2_co.basis()[Simplex((0,3))]
sage: c = 3*a-2*b
sage: z
(0, 2) + 2*(1, 3)
sage: c
-2*\chi_(0, 3) + 3*\chi_(1, 3)
sage: c.eval(z)
6
```

class **Element** (*M*, *x*)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

coboundary ( )

Return the coboundary of this cochain

OUTPUT:

The coboundary as a cochain in one degree higher.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ, cochains=True)
sage: from sage.homology.cubical_complex import Cube
sage: cochain = C1(Cube([[1, 1], [0, 1]])) - 2 * C1(Cube([[0, 1], [0,␣
↪0]]))
sage: cochain
```

```
-2*\chi_[0,1] x [0,0] + \chi_[1,1] x [0,1]
sage: cochain.coboundary()
-\chi_[0,1] x [0,1]
```

cup_product (*cochain*)

Return the cup product with another cochain

INPUT:

- `cochain` – cochain over the same cell complex

EXAMPLES:

```
sage: T2 = simplicial_complexes.Torus()
sage: C1 = T2.n_chains(1, base_ring=ZZ, cochains=True)
sage: def l(i, j):
....:     return C1(Simplex([i, j]))
sage: l1 = l(1, 3) + l(1, 4) + l(1, 6) + l(2, 4) - l(4, 5) + l(5, 6)
sage: l2 = l(1, 6) - l(2, 3) - l(2, 5) + l(3, 6) - l(4, 5) + l(5, 6)
```

The two one-cocycles are cohomology generators:

```
sage: l1.is_cocycle(), l1.is_coboundary()
(True, False)
sage: l2.is_cocycle(), l2.is_coboundary()
(True, False)
```

Their cup product is a two-cocycle that is again non-trivial in cohomology:

```
sage: l12 = l1.cup_product(l2)
sage: l12
\chi_(1, 3, 6) - \chi_(2, 4, 5) - \chi_(4, 5, 6)
sage: l1.parent().degree(), l2.parent().degree(), l12.parent().degree()
(1, 1, 2)
sage: l12.is_cocycle(), l12.is_coboundary()
(True, False)
```

eval (*other*)

Evaluate this cochain on the chain `other` .

INPUT:

- `other` – a chain for the same cell complex in the same dimension with the same base ring

OUTPUT: scalar

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: C_2 = S2.n_chains(1)
sage: C_2_co = S2.n_chains(1, cochains=True)
sage: x = C_2.basis()[Simplex((0,2))]
sage: y = C_2.basis()[Simplex((1,3))]
sage: z = x+2*y
sage: a = C_2_co.basis()[Simplex((1,3))]
sage: b = C_2_co.basis()[Simplex((0,3))]
sage: c = 3*a-2*b
sage: z
(0, 2) + 2*(1, 3)
sage: c
-2*\chi_(0, 3) + 3*\chi_(1, 3)
```

```
sage: c.eval(z)
6
```

TESTS:

```
sage: z.eval(c) # z is not a cochain
Traceback (most recent call last):
...
AttributeError: 'Chains_with_category.element_class' object has no␣
↪attribute 'eval'
sage: c.eval(c) # can't evaluate a cochain on a cochain
Traceback (most recent call last):
...
ValueError: argument is not a chain
```

is_coboundary ( )

Test whether the cochain is a coboundary

OUTPUT:

Boolean. Whether the cochain is the *coboundary()* of a cochain in one degree lower.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ, cochains=True)
sage: from sage.homology.cubical_complex import Cube
sage: cochain = C1(Cube([[1, 1], [0, 1]])) - C1(Cube([[0, 1], [0, 0]]))
sage: cochain.is_coboundary()
True
```

TESTS:

```
sage: cochain.is_boundary()
Traceback (most recent call last):
...
AttributeError: 'Cochains_with_category.element_class' object has no␣
↪attribute 'is_boundary'
```

is_cocycle ( )

Test whether the cochain is a cocycle

OUTPUT:

Boolean. Whether the *coboundary()* vanishes.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ, cochains=True)
sage: from sage.homology.cubical_complex import Cube
sage: cochain = C1(Cube([[1, 1], [0, 1]])) - C1(Cube([[0, 1], [0, 0]]))
sage: cochain.is_cocycle()
True
```

TESTS:

```
sage: cochain.is_cycle()
Traceback (most recent call last):
```

```
...
AttributeError: 'Cochains_with_category.element_class' object has no␣
↪attribute 'is_cycle'
```

to_complex ( )

Return the corresponding cochain complex element

OUTPUT:

An element of the cochain complex, see *sage.homology.chain_complex* .

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ, cochains=True)
sage: from sage.homology.cubical_complex import Cube
sage: cochain = C1(Cube([[1, 1], [0, 1]]))
sage: cochain.to_complex()
Chain(1:(0, 0, 0, 1))
sage: ascii_art(_)
   d_2       d_1  [0]  d_0  [0]  d_-1
0 <---- [0] <---- [0] <---- [0] <----- 0
                  [0]       [0]
                  [1]       [0]
```

Cochains. **cochain_complex** ( )

Return the cochain complex.

OUTPUT:

Cochain complex, see *sage.homology.chain_complex* .

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C2 = square.n_chains(2, QQ, cochains=True)
sage: C2.cochain_complex()
Chain complex with at most 3 nonzero terms over Rational Field
sage: ascii_art(C2.cochain_complex())
                                [ 0  0  1 -1]
                                [ 0  1  0 -1]
                                [ 1  0 -1  0]
           [-1  1 -1  1]        [ 1 -1  0  0]
 0 <-- C_2 <-------------- C_1 <-------------- C_0 <-- 0
```

Cochains. **dual** ( )

Return the chains

OUTPUT:

The chains of the same cells with the same base ring.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: cochains = square.n_chains(1, ZZ, cochains=True);  cochains
Free module generated by {[0,0] x [0,1], [0,1] x [0,0], [0,1] x [1,1], [1,1]␣
↪x [0,1]} over Integer Ring
sage: cochains.dual()
Free module generated by {[0,0] x [0,1], [0,1] x [0,0], [0,1] x [1,1], [1,1]␣
↪x [0,1]} over Integer Ring
```

```
sage: type(cochains)
<class 'sage.homology.chains.Cochains_with_category'>
sage: type(cochains.dual())
<class 'sage.homology.chains.Chains_with_category'>
```

CHAPTER

THREE

MORPHISMS OF CHAIN COMPLEXES

AUTHORS:

- Benjamin Antieau <d.ben.antieau@gmail.com> (2009.06)
- Travis Scrimshaw (2012-08-18): Made all simplicial complexes immutable to work with the homset cache.

This module implements morphisms of chain complexes. The input is a dictionary whose keys are in the grading group of the chain complex and whose values are matrix morphisms.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(1)
sage: S
Minimal triangulation of the 1-sphere
sage: C = S.chain_complex()
sage: C.differential()
{0: [], 1: [-1 -1  0]
 [ 1  0 -1]
 [ 0  1  1], 2: []}
sage: f = {0:zero_matrix(ZZ,3,3),1:zero_matrix(ZZ,3,3)}
sage: G = Hom(C,C)
sage: x = G(f)
sage: x
Chain complex endomorphism of Chain complex with at most 2 nonzero terms over Integer␣
↪Ring
sage: x._matrix_dictionary
{0: [0 0 0]
[0 0 0]
[0 0 0], 1: [0 0 0]
[0 0 0]
[0 0 0]}
```

class sage.homology.chain_complex_morphism. **ChainComplexMorphism** (*matrices*, *C*, *D*, *check=True*)

Bases: sage.categories.morphism.Morphism

An element of this class is a morphism of chain complexes.

dual ( )

The dual chain map to this one.

That is, the map from the dual of the codomain of this one to the dual of its domain, represented in each degree by the transpose of the corresponding matrix.

EXAMPLES:

```
sage: X = simplicial_complexes.Simplex(1)
sage: Y = simplicial_complexes.Simplex(0)
sage: g = Hom(X,Y)({0:0, 1:0})
sage: f = g.associated_chain_complex_morphism()
sage: f.in_degree(0)
[1 1]
sage: f.dual()
Chain complex morphism:
  From: Chain complex with at most 1 nonzero terms over Integer Ring
  To: Chain complex with at most 2 nonzero terms over Integer Ring
sage: f.dual().in_degree(0)
[1]
[1]
sage: ascii_art(f.domain())
            [-1]
            [ 1]
 0 <-- C_0 <----- C_1 <-- 0
sage: ascii_art(f.dual().codomain())
            [-1  1]
 0 <-- C_1 <-------- C_0 <-- 0
```

in_degree (*n*)

The matrix representing this morphism in degree n

INPUT:

- n – degree

EXAMPLES:

```
sage: C = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: zero_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: f = Hom(C,D)({0: identity_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: f.in_degree(0)
[1]
```

Note that if the matrix is not specified in the definition of the map, it is assumed to be zero:

```
sage: f.in_degree(2)
[]
sage: f.in_degree(2).nrows(), f.in_degree(2).ncols()
(1, 0)
sage: C.free_module(2)
Ambient free module of rank 0 over the principal ideal domain Integer Ring
sage: D.free_module(2)
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

is_identity ( )

True if this is the identity map.

EXAMPLES:

```
sage: S = SimplicialComplex(is_mutable=False)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: x = i.associated_chain_complex_morphism()
sage: x.is_identity()
True
```

is_injective ( )

True if this map is injective.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: H = Hom(S1, S1)
sage: flip = H({0:0, 1:2, 2:1})
sage: flip.associated_chain_complex_morphism().is_injective()
True

sage: pt = simplicial_complexes.Simplex(0)
sage: inclusion = Hom(pt, S1)({0:2})
sage: inclusion.associated_chain_complex_morphism().is_injective()
True
sage: inclusion.associated_chain_complex_morphism(cochain=True).is_injective()
False
```

is_surjective ( )

True if this map is surjective.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: H = Hom(S1, S1)
sage: flip = H({0:0, 1:2, 2:1})
sage: flip.associated_chain_complex_morphism().is_surjective()
True

sage: pt = simplicial_complexes.Simplex(0)
sage: inclusion = Hom(pt, S1)({0:2})
sage: inclusion.associated_chain_complex_morphism().is_surjective()
False
sage: inclusion.associated_chain_complex_morphism(cochain=True).is_
↪surjective()
True
```

to_matrix (*deg=None*)

The matrix representing this chain map.

If the degree `deg` is specified, return the matrix in that degree; otherwise, return the (block) matrix for the whole chain map.

INPUT:

- `deg` – (optional, default `None`) the degree

EXAMPLES:

```
sage: C = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: zero_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: f = Hom(C,D)({0: identity_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: f.to_matrix(0)
[1]
sage: f.to_matrix()
[1|0|]
[-+-+]
[0|0|]
[-+-+]
[0|0|]
```

sage.homology.chain_complex_morphism. **is_ChainComplexMorphism** (*x*)

Returns `True` if and only if `x` is a chain complex morphism.

EXAMPLES:

```
sage: from sage.homology.chain_complex_morphism import is_ChainComplexMorphism
sage: S = simplicial_complexes.Sphere(14)
sage: H = Hom(S,S)
sage: i = H.identity()  # long time (8s on sage.math, 2011)
sage: S = simplicial_complexes.Sphere(6)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: x = i.associated_chain_complex_morphism()
sage: x # indirect doctest
Chain complex morphism:
  From: Chain complex with at most 7 nonzero terms over Integer Ring
  To: Chain complex with at most 7 nonzero terms over Integer Ring
sage: is_ChainComplexMorphism(x)
True
```

CHAPTER

FOUR

CHAIN HOMOTOPIES AND CHAIN CONTRACTIONS

Chain homotopies are standard constructions in homological algebra: given chain complexes C and D and chain maps $f, g : C \to D$, say with differential of degree -1, a *chain homotopy* H between f and g is a collection of maps $H_n : C_n \to D_{n+1}$ satisfying

$$\partial_D H + H\partial_C = f - g.$$

The presence of a chain homotopy defines an equivalence relation (*chain homotopic*) on chain maps. If f and g are chain homotopic, then one can show that f and g induce the same map on homology.

Chain contractions are not as well known. The papers [MAR2009], [RMA2009], and [PR2015] provide some references. Given two chain complexes C and D, a *chain contraction* is a chain homotopy $H : C \to C$ for which there are chain maps $\pi : C \to D$ ("projection") and $\iota : D \to C$ ("inclusion") such that

- H is a chain homotopy between 1_C and $\iota\pi$,
- $\pi\iota = 1_D$,
- $\pi H = 0$,
- $H\iota = 0$,
- $HH = 0$.

Such a chain homotopy provides a strong relation between the chain complexes C and D; for example, their homology groups are isomorphic.

class sage.homology.chain_homotopy. **ChainContraction** (*matrices*, *pi*, *iota*)

Bases: *sage.homology.chain_homotopy.ChainHomotopy*

A chain contraction.

An algebraic gradient vector field $H : C \to C$ (that is a chain homotopy satisfying $HH = 0$) for which there are chain maps $\pi : C \to D$ ("projection") and $\iota : D \to C$ ("inclusion") such that

- H is a chain homotopy between 1_C and $\iota\pi$,
- $\pi\iota = 1_D$,
- $\pi H = 0$,
- $H\iota = 0$.

`H` is defined by a dictionary `matrices` of matrices.

INPUT:

- `matrices` – dictionary of matrices, keyed by dimension
- `pi` – a chain map $C \to D$
- `iota` – a chain map $D \to C$

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainContraction
sage: C = ChainComplex({0: zero_matrix(ZZ, 1), 1: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: matrix(ZZ, 0, 1)})
```

The chain complex C is chain homotopy equivalent to D, which is just a copy of $\mathbf{Z}$ in degree 0, and we construct a chain contraction:

```
sage: pi = Hom(C,D)({0: identity_matrix(ZZ, 1)})
sage: iota = Hom(D,C)({0: identity_matrix(ZZ, 1)})
sage: H = ChainContraction({0: zero_matrix(ZZ, 0, 1), 1: zero_matrix(ZZ, 1), 2:␣
↪identity_matrix(ZZ, 1)}, pi, iota)
```

dual ( )

The chain contraction dual to this one.

This is useful when switching from homology to cohomology.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: phi, M = S2.algebraic_topological_model(QQ)
sage: phi.iota()
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Rational Field
  To: Chain complex with at most 3 nonzero terms over Rational Field
```

Lifting the degree zero homology class gives a single vertex, but the degree zero cohomology class needs to be detected on every vertex, and vice versa for degree 2:

```
sage: phi.iota().in_degree(0)
[0]
[0]
[0]
[1]
sage: phi.dual().iota().in_degree(0)
[1]
[1]
[1]
[1]
sage: phi.iota().in_degree(2)
[-1]
[ 1]
[-1]
[ 1]
sage: phi.dual().iota().in_degree(2)
[0]
[0]
[0]
[1]
```

iota ( )

The chain map ι associated to this chain contraction.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: phi, M = S2.algebraic_topological_model(QQ)
```

```
sage: phi.iota()
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Rational Field
  To: Chain complex with at most 3 nonzero terms over Rational Field
```

Lifting the degree zero homology class gives a single vertex:

```
sage: phi.iota().in_degree(0)
[0]
[0]
[0]
[1]
```

Lifting the degree two homology class gives the signed sum of all of the 2-simplices:

```
sage: phi.iota().in_degree(2)
[-1]
[ 1]
[-1]
[ 1]
```

pi ( )

The chain map π associated to this chain contraction.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: phi, M = S2.algebraic_topological_model(QQ)
sage: phi.pi()
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Rational Field
  To: Chain complex with at most 3 nonzero terms over Rational Field
sage: phi.pi().in_degree(0)  # Every vertex represents a homology class.
[1 1 1 1]
sage: phi.pi().in_degree(1)  # No homology in degree 1.
[]
```

The degree 2 homology generator is detected on a single simplex:

```
sage: phi.pi().in_degree(2)
[0 0 0 1]
```

class sage.homology.chain_homotopy. **ChainHomotopy** (*matrices*, *f*, *g=None*)

Bases: sage.categories.morphism.Morphism

A chain homotopy.

A chain homotopy H between chain maps $f, g : C \to D$ is a sequence of maps $H_n : C_n \to D_{n+1}$ (if the chain complexes are graded homologically) satisfying

$$\partial_D H + H\partial_C = f - g.$$

INPUT:

- matrices – dictionary of matrices, keyed by dimension
- f – chain map $C \to D$
- g (optional) – chain map $C \to D$

The dictionary `matrices` defines `H` by specifying the matrix defining it in each degree: the entry m corresponding to key i gives the linear transformation $C_i \to D_{i+1}$.

If f is specified but not g, then g can be recovered from the defining formula. That is, if g is not specified, then it is defined to be $f - \partial_D H - H\partial_C$.

Note that the degree of the differential on the chain complex C must agree with that for D, and those degrees determine the "degree" of the chain homotopy map: if the degree of the differential is d, then the chain homotopy consists of a sequence of maps $C_n \to C_{n-d}$. The keys in the dictionary `matrices` specify the starting degrees.

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: zero_matrix(ZZ, 1)})
sage: f = Hom(C,D)({0: identity_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: g = Hom(C,D)({0: zero_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1), 1: identity_matrix(ZZ, 1)}, f,␣
↪g)
```

Note that the maps f and g are stored in the attributes `H._f` and `H._g`:

```
sage: H._f
Chain complex morphism:
  From: Chain complex with at most 2 nonzero terms over Integer Ring
  To: Chain complex with at most 2 nonzero terms over Integer Ring
sage: H._f.in_degree(0)
[1]
sage: H._g.in_degree(0)
[0]
```

A non-example:

```
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1), 1: zero_matrix(ZZ, 1)}, f, g)
Traceback (most recent call last):
...
ValueError: the data do not define a valid chain homotopy
```

dual ( )

Dual chain homotopy to this one.

That is, if this one is a chain homotopy between chain maps $f, g : C \to D$, then its dual is a chain homotopy between the dual of f and the dual of g, from D^* to C^*. It is represented in each degree by the transpose of the corresponding matrix.

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({1: matrix(ZZ, 0, 2)}) # one nonzero term in degree 1
sage: D = ChainComplex({0: matrix(ZZ, 0, 1)}) # one nonzero term in degree 0
sage: f = Hom(C, D)({})
sage: H = ChainHomotopy({1: matrix(ZZ, 1, 2, (3,1))}, f, f)
sage: H.in_degree(1)
[3 1]
sage: H.dual().in_degree(0)
[3]
[1]
```

in_degree (*n*)

The matrix representing this chain homotopy in degree `n`.

INPUT:

- n – degree

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({1: matrix(ZZ, 0, 2)}) # one nonzero term in degree 1
sage: D = ChainComplex({0: matrix(ZZ, 0, 1)}) # one nonzero term in degree 0
sage: f = Hom(C, D)({})
sage: H = ChainHomotopy({1: matrix(ZZ, 1, 2, (3,1))}, f, f)
sage: H.in_degree(1)
[3 1]
```

This returns an appropriately sized zero matrix if the chain homotopy is not defined in degree n:

```
sage: H.in_degree(-3)
[]
```

is_algebraic_gradient_vector_field ( )

An algebraic gradient vector field is a linear map $H : C \to C$ such that $HH = 0$.

(Some authors also require that $H\partial H = H$, whereas some make this part of the definition of "homology gradient vector field. We have made the second choice.) See Molina-Abril and Réal [MAR2009] and Réal and Molina-Abril [RMA2009] for this and related terminology.

See also *is_homology_gradient_vector_field()* .

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({0: zero_matrix(ZZ, 1), 1: identity_matrix(ZZ, 1)})
```

The chain complex C is chain homotopy equivalent to a copy of $\mathbf{Z}$ in degree 0. Two chain maps $C \to C$ will be chain homotopic as long as they agree in degree 0.

```
sage: f = Hom(C,C)({0: identity_matrix(ZZ, 1), 1: matrix(ZZ, 1, 1, [3]), 2:␣
↪matrix(ZZ, 1, 1, [3])})
sage: g = Hom(C,C)({0: identity_matrix(ZZ, 1), 1: matrix(ZZ, 1, 1, [2]), 2:␣
↪matrix(ZZ, 1, 1, [2])})
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1), 1: zero_matrix(ZZ, 1), 2:␣
↪identity_matrix(ZZ, 1)}, f, g)
sage: H.is_algebraic_gradient_vector_field()
True
```

A chain homotopy which is not an algebraic gradient vector field:

```
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1), 1: identity_matrix(ZZ, 1),␣
↪2: identity_matrix(ZZ, 1)}, f, g)
sage: H.is_algebraic_gradient_vector_field()
False
```

is_homology_gradient_vector_field ( )

A homology gradient vector field is an algebraic gradient vector field $H : C \to C$ (i.e., a chain homotopy satisfying $HH = 0$) such that $\partial H\partial = \partial$ and $H\partial H = H$.

See Molina-Abril and Réal [MAR2009] and Réal and Molina-Abril [RMA2009] for this and related terminology.

See also *is_algebraic_gradient_vector_field()* .

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({0: zero_matrix(ZZ, 1), 1: identity_matrix(ZZ, 1)})

sage: f = Hom(C,C)({0: identity_matrix(ZZ, 1), 1: matrix(ZZ, 1, 1, [3]), 2:␣
↪matrix(ZZ, 1, 1, [3])})
sage: g = Hom(C,C)({0: identity_matrix(ZZ, 1), 1: matrix(ZZ, 1, 1, [2]), 2:␣
↪matrix(ZZ, 1, 1, [2])})
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1), 1: zero_matrix(ZZ, 1), 2:␣
↪identity_matrix(ZZ, 1)}, f, g)
sage: H.is_homology_gradient_vector_field()
True
```

CHAPTER

FIVE

HOMSPACES BETWEEN CHAIN COMPLEXES

Note that some significant functionality is lacking. Namely, the homspaces are not actually modules over the base ring. It will be necessary to enrich some of the structure of chain complexes for this to be naturally available. On other hand, there are various overloaded operators. `__mul__` acts as composition. One can `__add__` , and one can `__mul__` with a ring element on the right.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(2)
sage: T = simplicial_complexes.Torus()
sage: C = S.chain_complex(augmented=True,cochain=True)
sage: D = T.chain_complex(augmented=True,cochain=True)
sage: G = Hom(C,D)
sage: G
Set of Morphisms from Chain complex with at most 4 nonzero terms over Integer Ring to␣
↪Chain complex with at most 4 nonzero terms over Integer Ring in Category of chain␣
↪complexes over Integer Ring

sage: S = simplicial_complexes.ChessboardComplex(3,3)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: x = i.associated_chain_complex_morphism(augmented=True)
sage: x
Chain complex morphism:
  From: Chain complex with at most 4 nonzero terms over Integer Ring
  To: Chain complex with at most 4 nonzero terms over Integer Ring
sage: x._matrix_dictionary
{-1: [1], 0: [1 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 1], 1: [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
```

```
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1], 2: [1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]}

sage: S = simplicial_complexes.Sphere(2)
sage: A = Hom(S,S)
sage: i = A.identity()
sage: x = i.associated_chain_complex_morphism()
sage: x
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Integer Ring
  To: Chain complex with at most 3 nonzero terms over Integer Ring
sage: y = x*4
sage: z = y*y
sage: (y+z)
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Integer Ring
  To: Chain complex with at most 3 nonzero terms over Integer Ring
sage: f = x._matrix_dictionary
sage: C = S.chain_complex()
sage: G = Hom(C,C)
sage: w = G(f)
sage: w == x
True
```

class sage.homology.chain_complex_homspace. **ChainComplexHomspace** (*X*, *Y*, *category=None*, *base=None*, *check=True*)

Bases: sage.categories.homset.Homset

Class of homspaces of chain complex morphisms.

EXAMPLES:

```
sage: T = SimplicialComplex([[1,2,3,4],[7,8,9]])
sage: C = T.chain_complex(augmented=True, cochain=True)
sage: G = Hom(C,C)
sage: G
Set of Morphisms from Chain complex with at most 5 nonzero terms over Integer␣
↪Ring to Chain complex with at most 5 nonzero terms over Integer Ring in␣
↪Category of chain complexes over Integer Ring
```

sage.homology.chain_complex_homspace. **is_ChainComplexHomspace** (*x*)

Returns `True` if and only if `x` is a morphism of chain complexes.

EXAMPLES:

```
sage: from sage.homology.chain_complex_homspace import is_ChainComplexHomspace
sage: T = SimplicialComplex([[1,2,3,4],[7,8,9]])
```

```
sage: C = T.chain_complex(augmented=True, cochain=True)
sage: G = Hom(C,C)
sage: is_ChainComplexHomspace(G)
True
```

CHAPTER

SIX

FINITE SIMPLICIAL COMPLEXES

AUTHORS:

- John H. Palmieri (2009-04)
- D. Benjamin Antieau (2009-06): added is_connected, generated_subcomplex, remove_facet, and is_flag_complex methods; cached the output of the graph() method.
- Travis Scrimshaw (2012-08-17): Made *SimplicialComplex* have an immutable option, and added `__hash__()` function which checks to make sure it is immutable. Made *SimplicialComplex.remove_face()* into a mutator. Deprecated the `vertex_set` parameter.
- Christian Stump (2011-06): implementation of is_cohen_macaulay
- Travis Scrimshaw (2013-02-16): Allowed *SimplicialComplex* to make mutable copies.
- Simon King (2014-05-02): Let simplicial complexes be objects of the category of simplicial complexes.

This module implements the basic structure of finite simplicial complexes. Given a set V of "vertices", a simplicial complex on V is a collection K of subsets of V satisfying the condition that if S is one of the subsets in K, then so is every subset of S. The subsets S are called the 'simplices' of K.

A simplicial complex K can be viewed as a purely combinatorial object, as described above, but it also gives rise to a topological space $|K|$ (its *geometric realization*) as follows: first, the points of V should be in general position in euclidean space. Next, if $\{v\}$ is in K, then the vertex v is in $|K|$. If $\{v, w\}$ is in K, then the line segment from v to w is in $|K|$. If $\{u, v, w\}$ is in K, then the triangle with vertices u, v, and w is in $|K|$. In general, $|K|$ is the union of the convex hulls of simplices of K. Frequently, one abuses notation and uses K to denote both the simplicial complex and the associated topological space.

For any simplicial complex K and any commutative ring R there is an associated chain complex, with differential of degree -1. The n^{th} term is the free R-module with basis given by the n-simplices of K. The differential is determined by its value on any simplex: on the n-simplex with vertices $(v_0, v_1, ..., v_n)$, the differential is the alternating sum with i^{th} summand $(-1)^i$ multiplied by the $(n-1)$-simplex obtained by omitting vertex v_i.

In the implementation here, the vertex set must be finite. To define a simplicial complex, specify its vertex set: this should be a list, tuple, or set, or it can be a non-negative integer n, in which case the vertex set is $(0, ..., n)$. Also specify the facets: the maximal faces.

Note: The elements of the vertex set are not automatically contained in the simplicial complex: each one is only included if and only if it is a vertex of at least one of the specified facets.

Note: This class derives from *GenericCellComplex* , and so inherits its methods. Some of those methods are not listed here; see the *Generic Cell Complex* page instead.

EXAMPLES:

```
sage: SimplicialComplex([[1], [3, 7]])
Simplicial complex with vertex set (1, 3, 7) and facets {(3, 7), (1,)}
sage: SimplicialComplex()    # the empty simplicial complex
Simplicial complex with vertex set () and facets {()}
sage: X = SimplicialComplex([[0,1], [1,2], [2,3], [3,0]])
sage: X
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2), (2, 3), (0, 3),␣
↪(0, 1)}
sage: X.stanley_reisner_ring()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring by the␣
↪ideal (x1*x3, x0*x2)
sage: X.is_pure()
True
```

Sage can perform a number of operations on simplicial complexes, such as the join and the product, and it can also compute homology:

```
sage: S = SimplicialComplex([[0,1], [1,2], [0,2]]) # circle
sage: T = S.product(S)  # torus
sage: T
Simplicial complex with 9 vertices and 18 facets
sage: T.homology()   # this computes reduced homology
{0: 0, 1: Z x Z, 2: Z}
sage: T.euler_characteristic()
0
```

Sage knows about some basic combinatorial data associated to a simplicial complex:

```
sage: X = SimplicialComplex([[0,1], [1,2], [2,3], [0,3]])
sage: X.f_vector()
[1, 4, 4]
sage: X.face_poset()
Finite poset containing 8 elements
sage: X.stanley_reisner_ring()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring by the␣
↪ideal (x1*x3, x0*x2)
```

Mutability (see trac ticket #12587):

```
sage: S = SimplicialComplex([[1,4], [2,4]])
sage: S.add_face([1,3])
sage: S.remove_face([1,3]); S
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(2, 4), (1, 4), (3,)}
sage: hash(S)
Traceback (most recent call last):
...
ValueError: This simplicial complex must be immutable. Call set_immutable().
```

```
sage: S = SimplicialComplex([[1,4], [2,4]])
sage: S.set_immutable()
sage: S.add_face([1,3])
Traceback (most recent call last):
...
ValueError: This simplicial complex is not mutable
sage: S.remove_face([1,3])
Traceback (most recent call last):
...
ValueError: This simplicial complex is not mutable
sage: hash(S) == hash(S)
True

sage: S2 = SimplicialComplex([[1,4], [2,4]], is_mutable=False)
sage: hash(S2) == hash(S)
True
```

We can also make mutable copies of an immutable simplicial complex (see trac ticket #14142):

```
sage: S = SimplicialComplex([[1,4], [2,4]])
sage: S.set_immutable()
sage: T = copy(S)
sage: T.is_mutable()
True
sage: S == T
True
```

class sage.homology.simplicial_complex. **Simplex** (*X*)

Bases: sage.structure.sage_object.SageObject

Define a simplex.

Topologically, a simplex is the convex hull of a collection of vertices in general position. Combinatorially, it is defined just by specifying a set of vertices. It is represented in Sage by the tuple of the vertices.

Parameters **X** (*integer or list, tuple, or other iterable*) – set of vertices

Returns simplex with those vertices

`X` may be a non-negative integer n, in which case the simplicial complex will have $n+1$ vertices $(0, 1, ..., n)$, or it may be anything which may be converted to a tuple, in which case the vertices will be that tuple. In the second case, each vertex must be hashable, so it should be a number, a string, or a tuple, for instance, but not a list.

Warning: The vertices should be distinct, and no error checking is done to make sure this is the case.

EXAMPLES:

```
sage: Simplex(4)
(0, 1, 2, 3, 4)
sage: Simplex([3, 4, 1])
(3, 4, 1)
sage: X = Simplex((3, 'a', 'vertex')); X
(3, 'a', 'vertex')
sage: X == loads(dumps(X))
True
```

Vertices may be tuples but not lists:

```
sage: Simplex([(1,2), (3,4)])
((1, 2), (3, 4))
sage: Simplex([[1,2], [3,4]])
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

alexander_whitney (*dim*)

Subdivide this simplex into a pair of simplices.

If this simplex has vertices v_0, v_1, ..., v_n, then subdivide it into simplices $(v_0, v_1, ..., v_{dim})$ and $(v_{dim}, v_{dim+1}, ..., v_n)$.

INPUT:

- `dim` – integer between 0 and one more than the dimension of this simplex

OUTPUT:

- a list containing just the triple `(1,left,right)` , where `left` and `right` are the two simplices described above.

This method allows one to construct a coproduct from the $p + q$-chains to the tensor product of the p-chains and the q-chains. The number 1 (a Sage integer) is the coefficient of `left tensor right` in this coproduct. (The corresponding formula is more complicated for the cubes that make up a cubical complex, and the output format is intended to be consistent for both cubes and simplices.)

Calling this method `alexander_whitney` is an abuse of notation, since the actual Alexander-Whitney map goes from $C(X \times Y) \to C(X) \otimes C(Y)$, where $C(-)$ denotes the chain complex of singular chains, but this subdivision of simplices is at the heart of it.

EXAMPLES:

```
sage: s = Simplex((0,1,3,4))
sage: s.alexander_whitney(0)
[(1, (0,), (0, 1, 3, 4))]
sage: s.alexander_whitney(2)
[(1, (0, 1, 3), (3, 4))]
```

dimension ( )

The dimension of this simplex.

The dimension of a simplex is the number of vertices minus 1.

EXAMPLES:

```
sage: Simplex(5).dimension() == 5
True
sage: Simplex(5).face(1).dimension()
4
```

face (*n*)

The n-th face of this simplex.

Parameters **n** (*integer*) – an integer between 0 and the dimension of this simplex

Returns the simplex obtained by removing the n-th vertex from this simplex

EXAMPLES:

```
sage: S = Simplex(4)
sage: S.face(0)
(1, 2, 3, 4)
sage: S.face(3)
(0, 1, 2, 4)
```

faces ( )

The list of faces (of codimension 1) of this simplex.

EXAMPLES:

```
sage: S = Simplex(4)
sage: S.faces()
[(1, 2, 3, 4), (0, 2, 3, 4), (0, 1, 3, 4), (0, 1, 2, 4), (0, 1, 2, 3)]
sage: len(Simplex(10).faces())
11
```

is_empty ( )

Return `True` iff this simplex is the empty simplex.

EXAMPLES:

```
sage: [Simplex(n).is_empty() for n in range(-1,4)]
[True, False, False, False, False]
```

is_face (*other*)

Return `True` iff this simplex is a face of other.

EXAMPLES:

```
sage: Simplex(3).is_face(Simplex(5))
True
sage: Simplex(5).is_face(Simplex(2))
False
sage: Simplex(['a', 'b', 'c']).is_face(Simplex(8))
False
```

join (*right*, *rename_vertices=True*)

The join of this simplex with another one.

The join of two simplices $[v_0, ..., v_k]$ and $[w_0, ..., w_n]$ is the simplex $[v_0, ..., v_k, w_0, ..., w_n]$.

Parameters

- **right** – the other simplex (the right-hand factor)
- **rename_vertices** (boolean; optional, default `True`) – If this is `True` , the vertices in the join will be renamed by this formula: vertex "v" in the left-hand factor –> vertex "Lv" in the join, vertex "w" in the right-hand factor –> vertex "Rw" in the join. If this is false, this tries to construct the join without renaming the vertices; this may cause problems if the two factors have any vertices with names in common.

EXAMPLES:

```
sage: Simplex(2).join(Simplex(3))
('L0', 'L1', 'L2', 'R0', 'R1', 'R2', 'R3')
sage: Simplex(['a', 'b']).join(Simplex(['x', 'y', 'z']))
('La', 'Lb', 'Rx', 'Ry', 'Rz')
```

```
sage: Simplex(['a', 'b']).join(Simplex(['x', 'y', 'z']), rename_
↪vertices=False)
('a', 'b', 'x', 'y', 'z')
```

product (*other*, *rename_vertices=True*)

The product of this simplex with another one, as a list of simplices.

Parameters

- **other** – the other simplex
- **rename_vertices** (boolean; optional, default `True`) – If this is `False` , then the vertices in the product are the set of ordered pairs (v, w) where v is a vertex in the left-hand factor (`self`) and w is a vertex in the right-hand factor (`other`). If this is `True` , then the vertices are renamed as "LvRw" (e.g., the vertex (1,2) would become "L1R2"). This is useful if you want to define the Stanley-Reisner ring of the complex: vertex names like (0,1) are not suitable for that, while vertex names like "L0R1" are.

Algorithm: see Hatcher, p. 277-278 [Hat2002] (who in turn refers to Eilenberg-Steenrod, p. 68): given `S = Simplex(m)` and `T = Simplex(n)` , then $S \times T$ can be triangulated as follows: for each path f from $(0, 0)$ to (m, n) along the integer grid in the plane, going up or right at each lattice point, associate an $(m + n)$-simplex with vertices v_0, v_1, ..., where v_k is the k^{th} vertex in the path f.

Note that there are $m+n$ choose n such paths. Note also that each vertex in the product is a pair of vertices (v, w) where v is a vertex in the left-hand factor and w is a vertex in the right-hand factor.

Note: This produces a list of simplices – not a *Simplex* , not a *SimplicialComplex* .

EXAMPLES:

```
sage: len(Simplex(2).product(Simplex(2)))
6
sage: Simplex(1).product(Simplex(1))
[('L0R0', 'L0R1', 'L1R1'), ('L0R0', 'L1R0', 'L1R1')]
sage: Simplex(1).product(Simplex(1), rename_vertices=False)
[((0, 0), (0, 1), (1, 1)), ((0, 0), (1, 0), (1, 1))]
```

set ( )

The frozenset attached to this simplex.

EXAMPLES:

```
sage: Simplex(3).set()
frozenset({0, 1, 2, 3})
```

tuple ( )

The tuple attached to this simplex.

EXAMPLES:

```
sage: Simplex(3).tuple()
(0, 1, 2, 3)
```

Although simplices are printed as if they were tuples, they are not the same type:

```
sage: type(Simplex(3).tuple())
<type 'tuple'>
```

```
sage: type(Simplex(3))
<class 'sage.homology.simplicial_complex.Simplex'>
```

class sage.homology.simplicial_complex. **SimplicialComplex** (*maximal_faces=None*, *from_characteristic_function=None*, *maximality_check=True*, *sort_facets=True*, *name_check=False*, *is_mutable=True*, *is_immutable=False*, *category=None*)

Bases: sage.structure.parent.Parent , *sage.homology.cell_complex.GenericCellComplex*

Define a simplicial complex.

Parameters

- **maximal_faces** – set of maximal faces
- **from_characteristic_function** – see below
- **maximality_check** (boolean; optional, default True) – see below
- **sort_facets** (boolean; optional, default True) – see below
- **name_check** (boolean; optional, default False) – see below
- **is_mutable** (boolean; optional, default True) – Set to False to make this immutable
- **category** (*category; optional, default finite simplicial complexes*) – the category of the simplicial complex

Returns a simplicial complex

maximal_faces should be a list or tuple or set (indeed, anything which may be converted to a set) whose elements are lists (or tuples, etc.) of vertices. Maximal faces are also known as 'facets'.

Alternatively, the maximal faces can be defined from a monotome boolean function on the subsets of a set X. While defining maximal_faces=None , you can thus set from_characteristic_function=(f,X) where X is the set of points and f a boolean monotone hereditary function that accepts a list of elements from X as input (see subsets_with_hereditary_property() for more information).

If maximality_check is True , check that each maximal face is, in fact, maximal. In this case, when producing the internal representation of the simplicial complex, omit those that are not. It is highly recommended that this be True ; various methods for this class may fail if faces which are claimed to be maximal are in fact not.

If sort_facets is True , sort the vertices in each facet. If the vertices in different facets are not ordered compatibly (e.g., if you have facets (1,3,5) and (5,3,8)), then homology calculations may have unpredictable results.

If name_check is True , check the names of the vertices to see if they can be easily converted to generators of a polynomial ring – use this if you plan to use the Stanley-Reisner ring for the simplicial complex.

EXAMPLES:

```
sage: SimplicialComplex([[1,2], [1,4]])
Simplicial complex with vertex set (1, 2, 4) and facets {(1, 2), (1, 4)}
sage: SimplicialComplex([[0,2], [0,3], [0]])
Simplicial complex with vertex set (0, 2, 3) and facets {(0, 2), (0, 3)}
sage: SimplicialComplex([[0,2], [0,3], [0]], maximality_check=False)
```

```
Simplicial complex with vertex set (0, 2, 3) and facets {(0, 2), (0, 3), (0,)}
sage: S = SimplicialComplex((('a', 'b'), ['a', 'c'], ('b', 'c')))
sage: S
Simplicial complex with vertex set ('a', 'b', 'c') and facets {('b', 'c'), ('a',
↪'c'), ('a', 'b')}
```

Finally, if there is only one argument and it is a simplicial complex, return that complex. If it is an object with a built-in conversion to simplicial complexes (via a `_simplicial_` method), then the resulting simplicial complex is returned:

```
sage: S = SimplicialComplex([[0,2], [0,3], [0,6]])
sage: SimplicialComplex(S) == S
True
sage: Tc = cubical_complexes.Torus(); Tc
Cubical complex with 16 vertices and 64 cubes
sage: Ts = SimplicialComplex(Tc); Ts
Simplicial complex with 16 vertices and 32 facets
sage: Ts.homology()
{0: 0, 1: Z x Z, 2: Z}
```

From a characteristic monotone boolean function, e.g. the simplicial complex of all subsets $S \subseteq \{0, 1, 2, 3, 4\}$ such that $sum(S) \leq 4$:

```
sage: SimplicialComplex(from_characteristic_function=(lambda x:sum(x)<=4,␣
↪range(5)))
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {(0, 4), (0, 1, 2),␣
↪(0, 1, 3)}
```

or e.g. the simplicial complex of all 168 hyperovals of the projective plane of order 4:

```
sage: l = designs.ProjectiveGeometryDesign(2,1,GF(4,name='a'))
sage: f = lambda S: not any(len(set(S).intersection(x))>2 for x in l)
sage: SimplicialComplex(from_characteristic_function=(f, l.ground_set()))
Simplicial complex with 21 vertices and 168 facets
```

TESTS:

Check that we can make mutable copies (see trac ticket #14142):

```
sage: S = SimplicialComplex([[0,2], [0,3]], is_mutable=False)
sage: S.is_mutable()
False
sage: C = copy(S)
sage: C.is_mutable()
True
sage: SimplicialComplex(S, is_mutable=True).is_mutable()
True
sage: SimplicialComplex(S, is_immutable=False).is_mutable()
True
```

Warning: Simplicial complexes are not proper parents as they do not possess element classes. In particular, parents are assumed to be hashable (and hence immutable) by the coercion framework. However this is close enough to being a parent with elements being the faces of `self` that we currently allow this abuse.

add_face (*face*)

Add a face to this simplicial complex.

Parameters `face` – a subset of the vertex set

This *changes* the simplicial complex, adding a new face and all of its subfaces.

EXAMPLES:

```
sage: X = SimplicialComplex([[0,1], [0,2]])
sage: X.add_face([0,1,2,]); X
Simplicial complex with vertex set (0, 1, 2) and facets {(0, 1, 2)}
sage: Y = SimplicialComplex(); Y
Simplicial complex with vertex set () and facets {()}
sage: Y.add_face([0,1])
sage: Y.add_face([1,2,3])
sage: Y
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2, 3), (0, 1)}
```

If you add a face which is already present, there is no effect:

```
sage: Y.add_face([1,3]); Y
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2, 3), (0, 1)}
```

TESTS:

Check that the bug reported at trac ticket #14354 has been fixed:

```
sage: T = SimplicialComplex([range(1,5)]).n_skeleton(1)
sage: T.homology(algorithm='no_chomp')
{0: 0, 1: Z x Z x Z}
sage: T.add_face([1,2,3])
sage: T.homology(algorithm='no_chomp')
{0: 0, 1: Z x Z, 2: 0}
```

Check that the `_faces` cache is treated correctly (trac ticket #20758):

```
sage: T = SimplicialComplex([range(1,5)]).n_skeleton(1)
sage: _ = T.faces() # populate the _faces attribute
sage: _ = T.homology() # add more to _faces
sage: T.add_face((1,2,3))
sage: all(Simplex((1,2,3)) in T._faces[L][2] for L in T._faces)
True
```

Check that the `__enlarged` cache is treated correctly (trac ticket #20758):

```
sage: T = SimplicialComplex([range(1,5)]).n_skeleton(1)
sage: T.homology(algorithm='no_chomp') # to populate the __enlarged attribute
{0: 0, 1: Z x Z x Z}
sage: T.add_face([1,2,3])
sage: len(T._SimplicialComplex__enlarged) > 0
True
```

Check we've fixed the bug reported at trac ticket #14578:

```
sage: t0 = SimplicialComplex()
sage: t0.add_face(('a', 'b'))
sage: t0.add_face(('c', 'd', 'e'))
sage: t0.add_face(('e', 'f', 'c'))
sage: t0.homology()
{0: Z, 1: 0, 2: 0}
```

alexander_dual (*is_mutable=True*)

The Alexander dual of this simplicial complex: according to the Macaulay2 documentation, this is the simplicial complex whose faces are the complements of its nonfaces.

Thus find the minimal nonfaces and take their complements to find the facets in the Alexander dual.

Parameters **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

EXAMPLES:

```
sage: Y = SimplicialComplex([[i] for i in range(5)]); Y
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {(4,), (2,),␣
↪(3,), (0,), (1,)}
sage: Y.alexander_dual()
Simplicial complex with vertex set (0, 1, 2, 3, 4) and 10 facets
sage: X = SimplicialComplex([[0,1], [1,2], [2,3], [3,0]])
sage: X.alexander_dual()
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 3), (0, 2)}
```

alexander_whitney (*simplex*, *dim_left*)

Subdivide this simplex into a pair of simplices.

If this simplex has vertices v_0, v_1, ..., v_n, then subdivide it into simplices $(v_0, v_1, ..., v_{dim})$ and $(v_{dim}, v_{dim+1}, ..., v_n)$.

See *Simplex.alexander_whitney()* for more details. This method just calls that one.

INPUT:

- `simplex` – a simplex in this complex
- `dim` – integer between 0 and one more than the dimension of this simplex

OUTPUT: a list containing just the triple `(1,left,right)` , where `left` and `right` are the two simplices described above.

EXAMPLES:

```
sage: s = Simplex((0,1,3,4))
sage: X = SimplicialComplex([s])
sage: X.alexander_whitney(s, 0)
[(1, (0,), (0, 1, 3, 4))]
sage: X.alexander_whitney(s, 2)
[(1, (0, 1, 3), (3, 4))]
```

algebraic_topological_model (*base_ring=None*)

Algebraic topological model for this simplicial complex with coefficients in `base_ring` .

The term "algebraic topological model" is defined by Pilarczyk and Réal [PR2015].

INPUT:

- `base_ring` - coefficient ring (optional, default `QQ`). Must be a field.

Denote by C the chain complex associated to this simplicial complex. The algebraic topological model is a chain complex M with zero differential, with the same homology as C, along with chain maps $\pi : C \to M$ and $\iota : M \to C$ satisfying $\iota\pi = 1_M$ and $\pi\iota$ chain homotopic to 1_C. The chain homotopy ϕ must satisfy

- $\phi\phi = 0$,
- $\pi\phi = 0$,
- $\phi\iota = 0$.

Such a chain homotopy is called a *chain contraction*.

OUTPUT: a pair consisting of

- chain contraction `phi` associated to C, M, π, and ι
- the chain complex M

Note that from the chain contraction `phi` , one can recover the chain maps π and ι via `phi.pi()` and `phi.iota()` . Then one can recover C and M from, for example, `phi.pi().domain()` and `phi.pi().codomain()` , respectively.

EXAMPLES:

```
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: phi, M = RP2.algebraic_topological_model(GF(2))
sage: M.homology()
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: T = simplicial_complexes.Torus()
sage: phi, M = T.algebraic_topological_model(QQ)
sage: M.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

automorphism_group ( )

Return the automorphism group of the simplicial complex.

This is done by creating a bipartite graph, whose vertices are vertices and facets of the simplicial complex, and computing its automorphism group.

Warning: Since trac ticket #14319 the domain of the automorphism group is equal to the graph's vertex set, and the `translation` argument has become useless.

EXAMPLES:

```
sage: S = simplicial_complexes.Simplex(3)
sage: S.automorphism_group().is_isomorphic(SymmetricGroup(4))
True

sage: P = simplicial_complexes.RealProjectivePlane()
sage: P.automorphism_group().is_isomorphic(AlternatingGroup(5))
True

sage: Z = SimplicialComplex([['1','2'],['2','3','a']])
sage: Z.automorphism_group().is_isomorphic(CyclicPermutationGroup(2))
True
sage: group = Z.automorphism_group()
sage: group.domain()
{'1', '2', '3', 'a'}
```

Check that trac ticket #17032 is fixed:

```
sage: s = SimplicialComplex([[(0,1),(2,3)]])
sage: s.automorphism_group().cardinality()
2
```

barycentric_subdivision ( )

The barycentric subdivision of this simplicial complex.

See http://en.wikipedia.org/wiki/Barycentric_subdivision for a definition.

EXAMPLES:

```
sage: triangle = SimplicialComplex([[0,1], [1,2], [0, 2]])
sage: hexagon = triangle.barycentric_subdivision()
sage: hexagon
Simplicial complex with 6 vertices and 6 facets
sage: hexagon.homology(1) == triangle.homology(1)
True
```

Barycentric subdivisions can get quite large, since each n-dimensional facet in the original complex produces $(n+1)!$ facets in the subdivision:

```
sage: S4 = simplicial_complexes.Sphere(4)
sage: S4
Minimal triangulation of the 4-sphere
sage: S4.barycentric_subdivision()
Simplicial complex with 62 vertices and 720 facets
```

cells (*subcomplex=None*)

The faces of this simplicial complex, in the form of a dictionary of sets keyed by dimension. If the optional argument `subcomplex` is present, then return only the faces which are *not* in the subcomplex.

Parameters **subcomplex** (optional, default `None`) – a subcomplex of this simplicial complex. Return faces which are not in this subcomplex.

EXAMPLES:

```
sage: Y = SimplicialComplex([[1,2], [1,4]])
sage: Y.faces()
{-1: {()}, 0: {(1,), (2,), (4,)}, 1: {(1, 2), (1, 4)}}
sage: L = SimplicialComplex([[1,2]])
sage: Y.faces(subcomplex=L)
{-1: set(), 0: {(4,)}, 1: {(1, 4)}}
```

chain_complex (*subcomplex=None*, *augmented=False*, *verbose=False*, *check=False*, *dimensions=None*, *base_ring=Integer Ring*, *cochain=False*)

The chain complex associated to this simplicial complex.

Parameters

- **dimensions** – if `None` , compute the chain complex in all dimensions. If a list or tuple of integers, compute the chain complex in those dimensions, setting the chain groups in all other dimensions to zero.
- **base_ring** (optional, default `ZZ`) – commutative ring
- **subcomplex** (*optional, default empty*) – a subcomplex of this simplicial complex. Compute the chain complex relative to this subcomplex.
- **augmented** (boolean; optional, default `False`) – If `True` , return the augmented chain complex (that is, include a class in dimension -1 corresponding to the empty cell). This is ignored if `dimensions` is specified.
- **cochain** (boolean; optional, default `False`) – If `True` , return the cochain complex (that is, the dual of the chain complex).

- **verbose** (boolean; optional, default `False`) – If `True`, print some messages as the chain complex is computed.
- **check** (boolean; optional, default `False`) – If `True`, make sure that the chain complex is actually a chain complex: the differentials are composable and their product is zero.

Note: If subcomplex is nonempty, then the argument `augmented` has no effect: the chain complex relative to a nonempty subcomplex is zero in dimension -1.

The rows and columns of the boundary matrices are indexed by the lists given by the *n_cells()* method, which by default are sorted.

EXAMPLES:

```
sage: circle = SimplicialComplex([[0,1], [1,2], [0, 2]])
sage: circle.chain_complex()
Chain complex with at most 2 nonzero terms over Integer Ring
sage: circle.chain_complex()._latex_()
'\Bold{Z}^{3} \xrightarrow{d_{1}} \Bold{Z}^{3}'
sage: circle.chain_complex(base_ring=QQ, augmented=True)
Chain complex with at most 3 nonzero terms over Rational Field
```

cone (*is_mutable=True*)

The cone on this simplicial complex.

Parameters **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

The cone is the simplicial complex formed by adding a new vertex C and simplices of the form $[C, v_0, ..., v_k]$ for every simplex $[v_0, ..., v_k]$ in the original simplicial complex. That is, the cone is the join of the original complex with a one-point simplicial complex.

EXAMPLES:

```
sage: S = SimplicialComplex([[0], [1]])
sage: S.cone()
Simplicial complex with vertex set ('L0', 'L1', 'R0') and facets {('L0', 'R0
↪'), ('L1', 'R0')}
```

connected_component (*simplex=None*)

Return the connected component of this simplicial complex containing `simplex`. If `simplex` is omitted, then return the connected component containing the zeroth vertex in the vertex list. (If the simplicial complex is empty, raise an error.)

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: S1 == S1.connected_component()
True
sage: X = S1.disjoint_union(S1)
sage: X == X.connected_component()
False
sage: v0 = X.vertices()[0]
sage: v1 = X.vertices()[-1]
sage: X.connected_component(Simplex([v0])) == X.connected_
↪component(Simplex([v1]))
False
```

```
sage: S0 = simplicial_complexes.Sphere(0)
sage: S0.vertices()
(0, 1)
sage: S0.connected_component()
Simplicial complex with vertex set (0,) and facets {(0,)}
sage: S0.connected_component(Simplex((1,)))
Simplicial complex with vertex set (1,) and facets {(1,)}

sage: SimplicialComplex([[]]).connected_component()
Traceback (most recent call last):
...
ValueError: the empty simplicial complex has no connected components.
```

connected_sum (*other*, *is_mutable=True*)

The connected sum of this simplicial complex with another one.

Parameters

- **other** – another simplicial complex
- **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

Returns the connected sum `self # other`

Warning: This does not check that `self` and `other` are manifolds, only that their facets all have the same dimension. Since a (more or less) random facet is chosen from each complex and then glued together, this method may return random results if applied to non-manifolds, depending on which facet is chosen.

Algorithm: a facet is chosen from each surface, and removed. The vertices of these two facets are relabeled to `(0,1,...,dim)` . Of the remaining vertices, the ones from the left-hand factor are renamed by prepending an "L", and similarly the remaining vertices in the right-hand factor are renamed by prepending an "R".

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: S1.connected_sum(S1.connected_sum(S1)).homology()
{0: 0, 1: Z}
sage: P = simplicial_complexes.RealProjectivePlane(); P
Minimal triangulation of the real projective plane
sage: P.connected_sum(P)     # the Klein bottle
Simplicial complex with 9 vertices and 18 facets
```

The notation '+' may be used for connected sum, also:

```
sage: P + P     # the Klein bottle
Simplicial complex with 9 vertices and 18 facets
sage: (P + P).homology()[1]
Z x C2
```

delta_complex (*sort_simplices=False*)

Returns `self` as a Δ-complex. The Δ-complex is essentially identical to the simplicial complex: it has same simplices with the same boundaries.

Parameters **sort_simplices** (boolean; optional, default `False`) – if `True` , sort the list of simplices in each dimension

EXAMPLES:

```
sage: T = simplicial_complexes.Torus()
sage: Td = T.delta_complex()
sage: Td
Delta complex with 7 vertices and 43 simplices
sage: T.homology() == Td.homology()
True
```

disjoint_union (*right*, *rename_vertices=True*, *is_mutable=True*)

The disjoint union of this simplicial complex with another one.

Parameters

- **right** – the other simplicial complex (the right-hand factor)
- **rename_vertices** (*boolean; optional, default True*) – If this is True, the vertices in the disjoint union will be renamed by the formula: vertex "v" in the left-hand factor –> vertex "Lv" in the disjoint union, vertex "w" in the right-hand factor –> vertex "Rw" in the disjoint union. If this is false, this tries to construct the disjoint union without renaming the vertices; this will cause problems if the two factors have any vertices with names in common.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: S2 = simplicial_complexes.Sphere(2)
sage: S1.disjoint_union(S2).homology()
{0: Z, 1: Z, 2: Z}
```

f_triangle ( )

Compute the f-triangle of `self` .

The f-triangle is given by $f_{i,j}$ being the number of faces F of size j such that $i = \max_{G \subseteq F} |G|$.

EXAMPLES:

```
sage: X = SimplicialComplex([[1,2,3], [3,4,5], [1,4], [1,5], [2,4], [2,5]])
sage: X.f_triangle()  ## this complex is not pure
[[0],
 [0, 0],
 [0, 0, 4],
 [1, 5, 6, 2]]
```

A complex is pure if and only if the last row is nonzero:

```
sage: X = SimplicialComplex([[1,2,3], [3,4,5], [1,4,5]])
sage: X.f_triangle()
[[0], [0, 0], [0, 0, 0], [1, 5, 8, 3]]
```

face (*simplex*, *i*)

The i-th face of `simplex` in this simplicial complex

INPUT:

•`simplex` – a simplex in this simplicial complex

•`i` – integer

EXAMPLES:

```
sage: S = SimplicialComplex([[0,1,4], [0,1,2]])
sage: S.face(Simplex((0,2)), 0)
(2,)

sage: S.face(Simplex((0,3)), 0)
Traceback (most recent call last):
...
ValueError: this simplex is not in this simplicial complex
```

face_iterator (*increasing=True*)

An iterator for the faces in this simplicial complex.

INPUT:

- `increasing` – (optional, default `True`) if `True` , return faces in increasing order of dimension, thus starting with the empty face. Otherwise it returns faces in decreasing order of dimension.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: [f for f in S1.face_iterator()]
[(), (2,), (0,), (1,), (1, 2), (0, 2), (0, 1)]
```

faces (*subcomplex=None*)

The faces of this simplicial complex, in the form of a dictionary of sets keyed by dimension. If the optional argument `subcomplex` is present, then return only the faces which are *not* in the subcomplex.

Parameters **subcomplex** (optional, default `None`) – a subcomplex of this simplicial complex. Return faces which are not in this subcomplex.

EXAMPLES:

```
sage: Y = SimplicialComplex([[1,2], [1,4]])
sage: Y.faces()
{-1: {()}, 0: {(1,), (2,), (4,)}, 1: {(1, 2), (1, 4)}}
sage: L = SimplicialComplex([[1,2]])
sage: Y.faces(subcomplex=L)
{-1: set(), 0: {(4,)}, 1: {(1, 4)}}
```

facets ( )

The maximal faces (a.k.a. facets) of this simplicial complex.

This just returns the set of facets used in defining the simplicial complex, so if the simplicial complex was defined with no maximality checking, none is done here, either.

EXAMPLES:

```
sage: Y = SimplicialComplex([[0,2], [1,4]])
sage: Y.maximal_faces()
{(1, 4), (0, 2)}
```

`facets` is a synonym for `maximal_faces` :

```
sage: S = SimplicialComplex([[0,1], [0,1,2]])
sage: S.facets()
{(0, 1, 2)}
```

fixed_complex (G)

Return the fixed simplicial complex $Fix(G)$ for a subgroup G.

INPUT:

- `G` – a subgroup of the automorphism group of the simplicial complex or a list of elements of the automorphism group

OUTPUT:

- a simplicial complex $Fix(G)$

Vertices in $Fix(G)$ are the orbits of G (acting on vertices of `self`) that form a simplex in `self` . More generally, simplices in $Fix(G)$ correspond to simplices in `self` that are union of such orbits.

A basic example:

```
sage: S4 = simplicial_complexes.Sphere(4)
sage: S3 = simplicial_complexes.Sphere(3)
sage: fix = S4.fixed_complex([S4.automorphism_group()([(0,1)])])
sage: fix
Simplicial complex with vertex set (0, 2, 3, 4, 5) and 5 facets
sage: fix.is_isomorphic(S3)
True
```

Another simple example:

```
sage: T = SimplicialComplex([[1,2,3],[2,3,4]])
sage: G = T.automorphism_group()
sage: T.fixed_complex([G([(1,4)])])
Simplicial complex with vertex set (2, 3) and facets {(2, 3)}
```

A more sophisticated example:

```
sage: RP2 = simplicial_complexes.ProjectivePlane()
sage: CP2 = simplicial_complexes.ComplexProjectivePlane()
sage: G = CP2.automorphism_group()
sage: H = G.subgroup([G([(2,3),(5,6),(8,9)])])
sage: CP2.fixed_complex(H).is_isomorphic(RP2)
True
```

flip_graph ( )

If `self` is pure, then it returns the flip graph of `self` , otherwise, it returns `None` .

The flip graph of a pure simplicial complex is the (undirected) graph with vertices being the facets, such that two facets are joined by an edge if they meet in a codimension 1 face.

The flip graph is used to detect if `self` is a pseudomanifold.

EXAMPLES:

```
sage: S0 = simplicial_complexes.Sphere(0)
sage: G = S0.flip_graph()
sage: G.vertices(); G.edges(labels=False)
[(0,), (1,)]
[((0,), (1,))]

sage: G = (S0.wedge(S0)).flip_graph()
sage: G.vertices(); G.edges(labels=False)
[(0,), ('L1',), ('R1',)]
[((0,), ('L1',)), ((0,), ('R1',)), (('L1',), ('R1',))]

sage: S1 = simplicial_complexes.Sphere(1)
sage: S2 = simplicial_complexes.Sphere(2)
```

```
sage: G = (S1.wedge(S1)).flip_graph()
sage: G.vertices(); G.edges(labels=False)
[(0, 'L1'), (0, 'L2'), (0, 'R1'), (0, 'R2'), ('L1', 'L2'), ('R1', 'R2')]
[((0, 'L1'), (0, 'L2')),
 ((0, 'L1'), (0, 'R1')),
 ((0, 'L1'), (0, 'R2')),
 ((0, 'L1'), ('L1', 'L2')),
 ((0, 'L2'), (0, 'R1')),
 ((0, 'L2'), (0, 'R2')),
 ((0, 'L2'), ('L1', 'L2')),
 ((0, 'R1'), (0, 'R2')),
 ((0, 'R1'), ('R1', 'R2')),
 ((0, 'R2'), ('R1', 'R2'))]

sage: (S1.wedge(S2)).flip_graph() is None
True

sage: G = S2.flip_graph()
sage: G.vertices(); G.edges(labels=False)
[(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]
[((0, 1, 2), (0, 1, 3)),
 ((0, 1, 2), (0, 2, 3)),
 ((0, 1, 2), (1, 2, 3)),
 ((0, 1, 3), (0, 2, 3)),
 ((0, 1, 3), (1, 2, 3)),
 ((0, 2, 3), (1, 2, 3))]

sage: T = simplicial_complexes.Torus()
sage: G = T.suspension(4).flip_graph()
sage: len(G.vertices()); len(G.edges(labels=False))
46
161
```

fundamental_group (*base_point=None*, *simplify=True*)

Return the fundamental group of this simplicial complex.

INPUT:

- `base_point` (optional, default None) – if this complex is not path-connected, then specify a vertex; the fundamental group is computed with that vertex as a base point. If the complex is path-connected, then you may specify a vertex or leave this as its default setting of `None` . (If this complex is path-connected, then this argument is ignored.)
- `simplify` (bool, optional True) – if False, then return a presentation of the group in terms of generators and relations. If True, the default, simplify as much as GAP is able to.

Algorithm: we compute the edge-path group – see Wikipedia article Fundamental_group. Choose a spanning tree for the 1-skeleton, and then the group's generators are given by the edges in the 1-skeleton; there are two types of relations: $e = 1$ if e is in the spanning tree, and for every 2-simplex, if its edges are e_0, e_1, and e_2, then we impose the relation $e_0 e_1^{-1} e_2 = 1$.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: S1.fundamental_group()
Finitely presented group < e |  >
```

If we pass the argument `simplify=False` , we get generators and relations in a form which is not usually very helpful. Here is the cyclic group of order 2, for instance:

```
sage: RP2 = simplicial_complexes.RealProjectiveSpace(2)
sage: C2 = RP2.fundamental_group(simplify=False)
sage: C2
Finitely presented group < e0, e1, e2, e3, e4, e5, e6, e7, e8, e9 | e0, e3,␣
↪e4, e7, e9, e5*e2^-1*e0, e7*e2^-1*e1, e8*e3^-1*e1, e8*e6^-1*e4, e9*e6^-1*e5␣
↪>
sage: C2.simplified()
Finitely presented group < e1 | e1^2 >
```

This is the same answer given if the argument `simplify` is True (the default):

```
sage: RP2.fundamental_group()
Finitely presented group < e1 | e1^2 >
```

You must specify a base point to compute the fundamental group of a non-connected complex:

```
sage: K = S1.disjoint_union(RP2)
sage: K.fundamental_group()
Traceback (most recent call last):
...
ValueError: this complex is not connected, so you must specify a base point.
sage: v0 = list(K.vertices())[0]
sage: K.fundamental_group(base_point=v0)
Finitely presented group < e |  >
sage: v1 = list(K.vertices())[-1]
sage: K.fundamental_group(base_point=v1)
Finitely presented group < e1 | e1^2 >
```

Some other examples:

```
sage: S1.wedge(S1).fundamental_group()
Finitely presented group < e0, e1 | >
sage: simplicial_complexes.Torus().fundamental_group()
Finitely presented group < e1, e4 | e4^-1*e1^-1*e4*e1 >
sage: simplicial_complexes.MooreSpace(5).fundamental_group()
Finitely presented group < e0 | e0^5 >
```

g_vector ( )

The g-vector of this simplicial complex.

If the h-vector of the complex is $(h_0, h_1, ..., h_d, h_{d+1})$ – see *h_vector()* – then its g-vector $(g_0, g_1, ..., g_{[(d+1)/2]})$ is defined by $g_0 = 1$ and $g_i = h_i - h_{i-1}$ for $i > 0$.

EXAMPLES:

```
sage: S3 = simplicial_complexes.Sphere(3).barycentric_subdivision()
sage: S3.f_vector()
[1, 30, 150, 240, 120]
sage: S3.h_vector()
[1, 26, 66, 26, 1]
sage: S3.g_vector()
[1, 25, 40]
```

generated_subcomplex (*sub_vertex_set*, *is_mutable=True*)

Returns the largest sub-simplicial complex of `self` containing exactly `sub_vertex_set` as vertices.

Parameters

- **sub_vertex_set** – The sub-vertex set.

- **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(2)
sage: S
Minimal triangulation of the 2-sphere
sage: S.generated_subcomplex([0,1,2])
Simplicial complex with vertex set (0, 1, 2) and facets {(0, 1, 2)}
```

graph ( )

The 1-skeleton of this simplicial complex, as a graph.

Warning: This may give the wrong answer if the simplicial complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: S = SimplicialComplex([[0,1,2,3]])
sage: G = S.graph(); G
Graph on 4 vertices
sage: G.edges()
[(0, 1, None), (0, 2, None), (0, 3, None), (1, 2, None), (1, 3, None), (2, 3,␣
↪None)]
```

h_triangle ( )

Compute the h-triangle of `self`.

The h-triangle of a simplicial complex Δ is given by

$$h_{i,j} = \sum_{k=0}^{j} (-1)^{j-k} \binom{i-k}{j-k} f_{i,k},$$

where $f_{i,k}$ is the f-triangle of Δ.

EXAMPLES:

```
sage: X = SimplicialComplex([[1,2,3], [3,4,5], [1,4], [1,5], [2,4], [2,5]])
sage: X.h_triangle()
[[0],
 [0, 0],
 [0, 0, 4],
 [1, 2, -1, 0]]
```

h_vector ( )

The h-vector of this simplicial complex.

If the complex has dimension d and $(f_{-1}, f_0, f_1, ..., f_d)$ is its f-vector (with $f_{-1} = 1$, representing the empty simplex), then the h-vector $(h_0, h_1, ..., h_d, h_{d+1})$ is defined by

$$\sum_{i=0}^{d+1} h_i x^{d+1-i} = \sum_{i=0}^{d+1} f_{i-1}(x-1)^{d+1-i}.$$

Alternatively,

$$h_j = \sum_{i=-1}^{j-1} (-1)^{j-i-1} \binom{d-i}{j-i-1} f_i.$$

EXAMPLES:

The f- and h-vectors of the boundary of an octahedron are computed in Wikipedia's page on simplicial complexes, http://en.wikipedia.org/wiki/Simplicial_complex:

```
sage: square = SimplicialComplex([[0,1], [1,2], [2,3], [0,3]])
sage: S0 = SimplicialComplex([[0], [1]])
sage: octa = square.join(S0) # boundary of an octahedron
sage: octa.f_vector()
[1, 6, 12, 8]
sage: octa.h_vector()
[1, 3, 3, 1]
```

is_cohen_macaulay (*base_ring=Rational Field*, *ncpus=0*)

Return `True` if `self` is Cohen-Macaulay.

A simplicial complex Δ is Cohen-Macaulay over R iff $\tilde{H}_i(\mathrm{lk}_\Delta(F); R) = 0$ for all $F \in \Delta$ and $i < \dim \mathrm{lk}_\Delta(F)$. Here, Δ is `self` and R is `base_ring` , and lk denotes the link operator on `self` .

INPUT:

- `base_ring` – (default: `QQ`) the base ring.
- `ncpus` – (default: 0) number of cpus used for the computation. If this is 0, determine the number of cpus automatically based on the hardware being used.

For finite simplicial complexes, this is equivalent to the statement that the Stanley-Reisner ring of `self` is Cohen-Macaulay.

EXAMPLES:

Spheres are Cohen-Macaulay:

```
sage: S = SimplicialComplex([[1,2],[2,3],[3,1]])
sage: S.is_cohen_macaulay(ncpus=3)
True
```

The following example is taken from Bruns, Herzog - Cohen-Macaulay rings, Figure 5.3:

```
sage: S = SimplicialComplex([[1,2,3],[1,4,5]])
sage: S.is_cohen_macaulay(ncpus=3)
...
False
```

The choice of base ring can matter. The real projective plane $\mathbf{R}P^2$ has $H_1(\mathbf{R}P^2) = \mathbf{Z}/2$, hence is CM over $\mathbf{Q}$ but not over $\mathbf{Z}$.

```
sage: X = simplicial_complexes.RealProjectivePlane()
sage: X.is_cohen_macaulay()
True
sage: X.is_cohen_macaulay(ZZ)
False
```

is_connected ( )

Returns `True` if and only if `self` is connected.

Warning: This may give the wrong answer if the simplicial complex was constructed with `maximality_check` set to `False` .

EXAMPLES:

```
sage: V = SimplicialComplex([[0,1,2],[3]])
sage: V
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 1, 2), (3,)}
sage: V.is_connected()
False

sage: X = SimplicialComplex([[0,1,2]])
sage: X.is_connected()
True

sage: U = simplicial_complexes.ChessboardComplex(3,3)
sage: U.is_connected()
True

sage: W = simplicial_complexes.Sphere(3)
sage: W.is_connected()
True

sage: S = SimplicialComplex([[0,1],[2,3]])
sage: S.is_connected()
False
```

is_flag_complex ( )

Returns `True` if and only if `self` is a flag complex.

A flag complex is a simplicial complex that is the largest simplicial complex on its 1-skeleton. Thus a flag complex is the clique complex of its graph.

EXAMPLES:

```
sage: h = Graph({0:[1,2,3,4],1:[2,3,4],2:[3]})
sage: x = h.clique_complex()
sage: x
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {(0, 1, 4), (0,␣
↪1, 2, 3)}
sage: x.is_flag_complex()
True

sage: X = simplicial_complexes.ChessboardComplex(3,3)
sage: X.is_flag_complex()
True
```

is_immutable ( )

Return `True` if immutable.

EXAMPLES:

```
sage: S = SimplicialComplex([[1,4], [2,4]])
sage: S.is_immutable()
False
sage: S.set_immutable()
sage: S.is_immutable()
True
```

is_isomorphic (*other*, *certificate=False*)

Check whether two simplicial complexes are isomorphic.

INPUT:

- `certificate` – if `True`, then output is `(a,b)`, where `a` is a boolean and `b` is either a map or `None`

This is done by creating two graphs and checking whether they are isomorphic.

EXAMPLES:

```
sage: Z1 = SimplicialComplex([[0,1],[1,2],[2,3,4],[4,5]])
sage: Z2 = SimplicialComplex([['a','b'],['b','c'],['c','d','e'],['e','f']])
sage: Z3 = SimplicialComplex([[1,2,3]])
sage: Z1.is_isomorphic(Z2)
True
sage: Z1.is_isomorphic(Z2, certificate=True)
(True, {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f'})
sage: Z3.is_isomorphic(Z2)
False
```

We check that trac ticket #20751 is fixed:

```
sage: C1 = SimplicialComplex([[1,2,3], [1,2,4], [1,3,4]])
sage: C2 = SimplicialComplex([['j','k','l'], ['j','l','m'], ['j','k','m']])
sage: C1.is_isomorphic(C2, certificate=True)
(True, {1: 'j', 2: 'k', 3: 'l', 4: 'm'})
```

TESTS:

```
sage: Z1 = SimplicialComplex([[0,1],[1,2],[2,3,4],[4,5]])
sage: Z2 = SimplicialComplex([['a','b'],['b','c'],['c','d','e'],['e','f']])
sage: Z1.is_isomorphic(Z2, certify=True)
doctest...: DeprecationWarning: use the option 'certificate' instead of
↪'certify'
See http://trac.sagemath.org/21111 for details.
(True, {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f'})
```

is_mutable ( )

Return `True` if mutable.

EXAMPLES:

```
sage: S = SimplicialComplex([[1,4], [2,4]])
sage: S.is_mutable()
True
sage: S.set_immutable()
sage: S.is_mutable()
False
sage: S2 = SimplicialComplex([[1,4], [2,4]], is_mutable=False)
sage: S2.is_mutable()
False
sage: S3 = SimplicialComplex([[1,4], [2,4]], is_mutable=False)
sage: S3.is_mutable()
False
```

is_pseudomanifold ( )

Return True if self is a pseudomanifold.

A pseudomanifold is a simplicial complex with the following properties:

- it is pure of some dimension d (all of its facets are d-dimensional)
- every $(d-1)$-dimensional simplex is the face of exactly two facets

•for every two facets S and T, there is a sequence of facets

$$S = f_0, f_1, ..., f_n = T$$

such that for each i, f_i and f_{i-1} intersect in a $(d-1)$-simplex.

By convention, S^0 is the only 0-dimensional pseudomanifold.

EXAMPLES:

```
sage: S0 = simplicial_complexes.Sphere(0)
sage: S0.is_pseudomanifold()
True
sage: (S0.wedge(S0)).is_pseudomanifold()
False
sage: S1 = simplicial_complexes.Sphere(1)
sage: S2 = simplicial_complexes.Sphere(2)
sage: (S1.wedge(S1)).is_pseudomanifold()
False
sage: (S1.wedge(S2)).is_pseudomanifold()
False
sage: S2.is_pseudomanifold()
True
sage: T = simplicial_complexes.Torus()
sage: T.suspension(4).is_pseudomanifold()
True
```

is_pure ( )

Return `True` iff this simplicial complex is pure.

A simplicial complex is pure if and only if all of its maximal faces have the same dimension.

Warning: This may give the wrong answer if the simplicial complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: U = SimplicialComplex([[1,2], [1, 3, 4]])
sage: U.is_pure()
False
sage: X = SimplicialComplex([[0,1], [0,2], [1,2]])
sage: X.is_pure()
True
```

Demonstration of the warning:

```
sage: S = SimplicialComplex([[0,1], [0]], maximality_check=False)
sage: S.is_pure()
False
```

is_shellable (*certificate=False*)

Return if `self` is shellable.

A simplicial complex is shellable if there exists a shelling order.

Note:

1.This method can check all orderings of the facets by brute force, hence can be very slow.

2.This is shellability in the general (nonpure) sense of Bjorner and Wachs [BW1996]. This method does not check purity.

See also:

is_shelling_order()

INPUT:

- `certificate` – (default: `False`) if `True` then returns the shelling order (if it exists)

EXAMPLES:

```
sage: X = SimplicialComplex([[1,2,5], [2,3,5], [3,4,5], [1,4,5]])
sage: X.is_shellable()
True
sage: order = X.is_shellable(True); order
((2, 3, 5), (1, 2, 5), (1, 4, 5), (3, 4, 5))
sage: X.is_shelling_order(order)
True

sage: X = SimplicialComplex([[1,2,3], [3,4,5]])
sage: X.is_shellable()
False
```

Examples from Figure 1 in [BW1996]:

```
sage: X = SimplicialComplex([[1,2,3], [3,4], [4,5], [5,6], [4,6]])
sage: X.is_shellable()
True

sage: X = SimplicialComplex([[1,2,3], [3,4], [4,5,6]])
sage: X.is_shellable()
False
```

REFERENCES:

- Wikipedia article Shelling_(topology)

is_shelling_order (*shelling_order*, *certificate=False*)

Return if the order of the facets given by `shelling_order` is a shelling order for `self` .

A sequence of facets $(F_i)_{i=1}^N$ of a simplicial complex of dimension d is a *shelling order* if for all $i = 2, 3, 4, \ldots$, the complex

$$X_i = \left(\bigcup_{j=1}^{i-1} F_j \right) \cap F_i$$

is pure and of dimension $\dim F_i - 1$.

INPUT:

- `shelling_order` – an ordering of the facets of `self`
- `certificate` – (default: `False`) if `True` then returns the index of the first facet that violate the condition

See also:

is_shellable()

EXAMPLES:

```
sage: facets = [[1,2,5],[2,3,5],[3,4,5],[1,4,5]]
sage: X = SimplicialComplex(facets)
sage: X.is_shelling_order(facets)
True

sage: b = [[1,2,5], [3,4,5], [2,3,5], [1,4,5]]
sage: X.is_shelling_order(b)
False
sage: X.is_shelling_order(b, True)
(False, 1)
```

A non-pure example:

```
sage: facets = [[1,2,3], [3,4], [4,5], [5,6], [4,6]]
sage: X = SimplicialComplex(facets)
sage: X.is_shelling_order(facets)
True
```

REFERENCES:

- [BW1996]

join (*right*, *rename_vertices=True*, *is_mutable=True*)

The join of this simplicial complex with another one.

The join of two simplicial complexes S and T is the simplicial complex $S * T$ with simplices of the form $[v_0, ..., v_k, w_0, ..., w_n]$ for all simplices $[v_0, ..., v_k]$ in S and $[w_0, ..., w_n]$ in T.

Parameters

- **right** – the other simplicial complex (the right-hand factor)
- **rename_vertices** (boolean; optional, default `True`) – If this is True, the vertices in the join will be renamed by the formula: vertex "v" in the left-hand factor –> vertex "Lv" in the join, vertex "w" in the right-hand factor –> vertex "Rw" in the join. If this is false, this tries to construct the join without renaming the vertices; this will cause problems if the two factors have any vertices with names in common.
- **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

EXAMPLES:

```
sage: S = SimplicialComplex([[0], [1]])
sage: T = SimplicialComplex([[2], [3]])
sage: S.join(T)
Simplicial complex with vertex set ('L0', 'L1', 'R2', 'R3') and 4 facets
sage: S.join(T, rename_vertices=False)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 3), (1, 2),␣
↪(0, 2), (0, 3)}
```

The notation '*' may be used, as well:

```
sage: S * S
Simplicial complex with vertex set ('L0', 'L1', 'R0', 'R1') and 4 facets
sage: S * S * S * S * S * S * S * S
Simplicial complex with 16 vertices and 256 facets
```

link (*simplex*, *is_mutable=True*)

The link of a simplex in this simplicial complex.

The link of a simplex F is the simplicial complex formed by all simplices G which are disjoint from F but for which $F \cup G$ is a simplex.

Parameters

- **simplex** – a simplex in this simplicial complex.
- **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

EXAMPLES:

```
sage: X = SimplicialComplex([[0,1,2], [1,2,3]])
sage: X.link(Simplex([0]))
Simplicial complex with vertex set (1, 2) and facets {(1, 2)}
sage: X.link([1,2])
Simplicial complex with vertex set (0, 3) and facets {(3,), (0,)}
sage: Y = SimplicialComplex([[0,1,2,3]])
sage: Y.link([1])
Simplicial complex with vertex set (0, 2, 3) and facets {(0, 2, 3)}
```

maximal_faces()

The maximal faces (a.k.a. facets) of this simplicial complex.

This just returns the set of facets used in defining the simplicial complex, so if the simplicial complex was defined with no maximality checking, none is done here, either.

EXAMPLES:

```
sage: Y = SimplicialComplex([[0,2], [1,4]])
sage: Y.maximal_faces()
{(1, 4), (0, 2)}
```

`facets` is a synonym for `maximal_faces`:

```
sage: S = SimplicialComplex([[0,1], [0,1,2]])
sage: S.facets()
{(0, 1, 2)}
```

minimal_nonfaces()

Set consisting of the minimal subsets of the vertex set of this simplicial complex which do not form faces.

Algorithm: Proceeds through the faces of the complex increasing the dimension, starting from dimension 0, and add the faces that are not contained in the complex and that are not already contained in a previously seen minimal non-face.

This is used in computing the *Stanley-Reisner ring* and the *Alexander dual*.

EXAMPLES:

```
sage: X = SimplicialComplex([[1,3],[1,2]])
sage: X.minimal_nonfaces()
{(2, 3)}
sage: Y = SimplicialComplex([[0,1], [1,2], [2,3], [3,0]])
sage: Y.minimal_nonfaces()
{(1, 3), (0, 2)}
```

TESTS:

```
sage: SC = SimplicialComplex([(0,1,2),(0,2,3),(2,3,4),(1,2,4),
 ↪                            (1,4,5),(0,3,6),(3,6,7),(4,5,7)])
sage: SC.minimal_nonfaces() #  This was taking a long time before :trac:
 ↪`20078`
```

```
{(3, 4, 7), (0, 7), (0, 4), (0, 5), (3, 5), (1, 7), (2, 5), (5, 6),
(1, 3), (4, 6), (2, 7), (2, 6), (1, 6)}
```

n_cells (*n*, *subcomplex=None*, *sort=None*)

List of cells of dimension `n` of this cell complex.

If the optional argument `subcomplex` is present, then return the `n` -dimensional faces which are *not* in the subcomplex. Sort the list if the argument `sort` is `True` . If `sort` is `None` (the default), then sort depending on the value of the `sort_facets` parameter (from the initialization of the simplicial complex).

Note: This list is sorted to provide reliable indexing for the rows and columns of the matrices of differentials in the associateed chain complex.

EXAMPLES:

```
sage: S = Set(range(1,5))
sage: Z = SimplicialComplex(S.subsets())
sage: Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 2, 3, 4)}
sage: Z.n_cells(2)
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
sage: K = SimplicialComplex([[1,2,3], [2,3,4]])
sage: Z.n_cells(2, subcomplex=K)
[(1, 2, 4), (1, 3, 4)]
sage: S = SimplicialComplex([[complex(i), complex(1)]], sort_facets=False)
sage: S.n_cells(0)
[(1j,), ((1+0j),)]
```

n_faces (*n*, *subcomplex=None*)

The set of simplices of dimension `n` of this simplicial complex. If the optional argument `subcomplex` is present, then return the `n` -dimensional faces which are *not* in the subcomplex.

Parameters

- **n** – non-negative integer
- **subcomplex** (optional, default `None`) – a subcomplex of this simplicial complex. Return `n` -dimensional faces which are not in this subcomplex.

Note: This method is not used elsewhere in Sage. The current usage: if order doesn't matter, for example to test membership, use *faces()* . If the order of the cells matters, use *n_cells()* .

EXAMPLES:

```
sage: S = Set(range(1,5))
sage: Z = SimplicialComplex(S.subsets())
sage: Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 2, 3, 4)}
sage: Z.n_faces(2)
{(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)}
sage: K = SimplicialComplex([[1,2,3], [2,3,4]])
sage: Z.n_faces(2, subcomplex=K)
{(1, 2, 4), (1, 3, 4)}
```

n_skeleton (*n*)

The n-skeleton of this simplicial complex.

The n-skeleton of a simplicial complex is obtained by discarding all of the simplices in dimensions larger than n.

Parameters **n** – non-negative integer

EXAMPLES:

```
sage: X = SimplicialComplex([[0,1], [1,2,3], [0,2,3]])
sage: X.n_skeleton(1)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(2, 3), (0, 2),␣
↪(1, 3), (1, 2), (0, 3), (0, 1)}
sage: X.set_immutable()
sage: X.n_skeleton(2)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (1, 2,␣
↪3), (0, 1)}
sage: X.n_skeleton(4)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (1, 2,␣
↪3), (0, 1)}
```

product (*right*, *rename_vertices=True*, *is_mutable=True*)

The product of this simplicial complex with another one.

Parameters

- **right** – the other simplicial complex (the right-hand factor)
- **rename_vertices** (boolean; optional, default `True`) – If this is False, then the vertices in the product are the set of ordered pairs (v, w) where v is a vertex in `self` and w is a vertex in `right` . If this is `True` , then the vertices are renamed as "LvRw" (e.g., the vertex (1,2) would become "L1R2"). This is useful if you want to define the Stanley-Reisner ring of the complex: vertex names like (0,1) are not suitable for that, while vertex names like "L0R1" are.
- **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

The vertices in the product will be the set of ordered pairs (v, w) where v is a vertex in self and w is a vertex in right.

Warning: If `X` and `Y` are simplicial complexes, then `X*Y` returns their join, not their product.

EXAMPLES:

```
sage: S = SimplicialComplex([[0,1], [1,2], [0,2]]) # circle
sage: K = SimplicialComplex([[0,1]])    # edge
sage: S.product(K).vertices()  # cylinder
('L0R0', 'L0R1', 'L1R0', 'L1R1', 'L2R0', 'L2R1')
sage: S.product(K, rename_vertices=False).vertices()
((0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1))
sage: T = S.product(S)  # torus
sage: T
Simplicial complex with 9 vertices and 18 facets
sage: T.homology()
{0: 0, 1: Z x Z, 2: Z}
```

These can get large pretty quickly:

```
sage: T = simplicial_complexes.Torus(); T
Minimal triangulation of the torus
sage: K = simplicial_complexes.KleinBottle(); K
Minimal triangulation of the Klein bottle
sage: T.product(K)      # long time: 5 or 6 seconds
Simplicial complex with 56 vertices and 1344 facets
```

remove_face (*face*)

Remove a face from this simplicial complex and return the resulting simplicial complex.

Parameters **face** – a face of the simplicial complex

This *changes* the simplicial complex.

ALGORITHM:

The facets of the new simplicial complex are the facets of the original complex not containing `face` , together with those of `link(face)*boundary(face)` .

EXAMPLES:

```
sage: S = range(1,5)
sage: Z = SimplicialComplex([S]); Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 2, 3, 4)}
sage: Z.remove_face([1,2])
sage: Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 3, 4), (2, 3,␣
↪4)}

sage: S = SimplicialComplex([[0,1,2],[2,3]])
sage: S
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 1, 2), (2, 3)}
sage: S.remove_face([0,1,2])
sage: S
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2), (2, 3),␣
↪(0, 2), (0, 1)}
```

TESTS:

Check that the `_faces` cache is treated properly: see trac ticket #20758:

```
sage: T = SimplicialComplex([range(1,5)]).n_skeleton(1)
sage: _ = T.faces() # populate the _faces attribute
sage: _ = T.homology(algorithm='no_chomp') # add more to _faces
sage: T.add_face((1,2,3))
sage: T.remove_face((1,2,3))
sage: len(T._faces)
2
sage: T.remove_face((3,4))
sage: len(T._faces)
1
```

restriction_sets (*order*)

Return the restriction sets of the facets according to `order` .

A restriction set of a shelling order is the sequence of smallest new faces that are created during the shelling order.

See also:

is_shelling_order()

EXAMPLES:

```
sage: facets = [[1,2,5], [2,3,5], [3,4,5], [1,4,5]]
sage: X = SimplicialComplex(facets)
sage: X.restriction_sets(facets)
[(), (3,), (4,), (1, 4)]

sage: b = [[1,2,5], [3,4,5], [2,3,5], [1,4,5]]
sage: X.restriction_sets(b)
Traceback (most recent call last):
...
ValueError: not a shelling order
```

set_immutable ( )

Make this simplicial complex immutable.

EXAMPLES:

```
sage: S = SimplicialComplex([[1,4], [2,4]])
sage: S.is_mutable()
True
sage: S.set_immutable()
sage: S.is_mutable()
False
```

stanley_reisner_ring (*base_ring=Integer Ring*)

The Stanley-Reisner ring of this simplicial complex.

Parameters **base_ring** (optional, default `ZZ`) – a commutative ring

Returns a quotient of a polynomial algebra with coefficients in `base_ring` , with one generator for each vertex in the simplicial complex, by the ideal generated by the products of those vertices which do not form faces in it.

Thus the ideal is generated by the products corresponding to the minimal nonfaces of the simplicial complex.

Warning: This may be quite slow!

Also, this may behave badly if the vertices have the 'wrong' names. To avoid this, define the simplicial complex at the start with the flag `name_check` set to `True` .

More precisely, this is a quotient of a polynomial ring with one generator for each vertex. If the name of a vertex is a non-negative integer, then the corresponding polynomial generator is named `'x'` followed by that integer (e.g., `'x2'` , `'x3'` , `'x5'` , ...). Otherwise, the polynomial generators are given the same names as the vertices. Thus if the vertex set is `(2,'x2')` , there will be problems.

EXAMPLES:

```
sage: X = SimplicialComplex([[0,1], [1,2], [2,3], [0,3]])
sage: X.stanley_reisner_ring()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring␣
↪by the ideal (x1*x3, x0*x2)
sage: Y = SimplicialComplex([[0,1,2,3,4]]); Y
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {(0, 1, 2, 3,␣
↪4)}
sage: Y.add_face([0,1,2,3,4])
```

```
sage: Y.stanley_reisner_ring(base_ring=QQ)
Multivariate Polynomial Ring in x0, x1, x2, x3, x4 over Rational Field
```

suspension (*n=1*, *is_mutable=True*)

The suspension of this simplicial complex.

Parameters

- **n** (*optional, default 1*) – positive integer – suspend this many times.
- **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

The suspension is the simplicial complex formed by adding two new vertices S_0 and S_1 and simplices of the form $[S_0, v_0, ..., v_k]$ and $[S_1, v_0, ..., v_k]$ for every simplex $[v_0, ..., v_k]$ in the original simplicial complex. That is, the suspension is the join of the original complex with a two-point simplicial complex.

If the simplicial complex M happens to be a pseudomanifold (see *is_pseudomanifold()*), then this instead constructs Datta's one-point suspension (see [Dat2007], p. 434): choose a vertex u in M and choose a new vertex w to add. Denote the join of simplices by "$*$". The facets in the one-point suspension are of the two forms

•$u * \alpha$ where α is a facet of M not containing u

•$w * \beta$ where β is any facet of M.

EXAMPLES:

```
sage: S0 = SimplicialComplex([[0], [1]])
sage: S0.suspension() == simplicial_complexes.Sphere(1)
True
sage: S3 = S0.suspension(3)  # the 3-sphere
sage: S3.homology()
{0: 0, 1: 0, 2: 0, 3: Z}
```

For pseudomanifolds, the complex constructed here will be smaller than that obtained by taking the join with the 0-sphere: the join adds two vertices, while this construction only adds one.

```
sage: T = simplicial_complexes.Torus()
sage: T.join(S0).vertices()      # 9 vertices
('L0', 'L1', 'L2', 'L3', 'L4', 'L5', 'L6', 'R0', 'R1')
sage: T.suspension().vertices()  # 8 vertices
(0, 1, 2, 3, 4, 5, 6, 7)
```

vertices ( )

The vertex set, as a tuple, of this simplicial complex.

EXAMPLES:

```
sage: S = SimplicialComplex([[i] for i in range(16)] + [[0,1], [1,2]])
sage: S
Simplicial complex with 16 vertices and 15 facets
sage: S.vertices()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

wedge (*right*, *rename_vertices=True*, *is_mutable=True*)

The wedge (one-point union) of this simplicial complex with another one.

Parameters

- **right** – the other simplicial complex (the right-hand factor)

- **rename_vertices** (boolean; optional, default `True`) – If this is `True`, the vertices in the wedge will be renamed by the formula: first vertex in each are glued together and called "0". Otherwise, each vertex "v" in the left-hand factor –> vertex "Lv" in the wedge, vertex "w" in the right-hand factor –> vertex "Rw" in the wedge. If this is `False`, this tries to construct the wedge without renaming the vertices; this will cause problems if the two factors have any vertices with names in common.
- **is_mutable** (boolean; optional, default `True`) – Determines if the output is mutable

Note: This operation is not well-defined if `self` or `other` is not path-connected.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: S2 = simplicial_complexes.Sphere(2)
sage: S1.wedge(S2).homology()
{0: 0, 1: Z, 2: Z}
```

sage.homology.simplicial_complex. **facets_for_K3** ( )

Returns the facets for a minimal triangulation of the K3 surface.

This is a pure simplicial complex of dimension 4 with 16 vertices and 288 facets. The facets are obtained by constructing a few facets and a permutation group G, and then computing the G-orbit of those facets.

See Casella and Kühnel in [CK2001] and Spreer and Kühnel [SK2011]; the construction here uses the labeling from Spreer and Kühnel.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import facets_for_K3
sage: A = facets_for_K3()    # long time (a few seconds)
sage: SimplicialComplex(A) == simplicial_complexes.K3Surface()  # long time
True
```

sage.homology.simplicial_complex. **facets_for_RP4** ( )

Return the list of facets for a minimal triangulation of 4-dimensional real projective space.

We use vertices numbered 1 through 16, define two facets, and define a certain subgroup G of the symmetric group S_{16}. Then the set of all facets is the G-orbit of the two given facets.

See the description in Example 3.12 in Datta [Dat2007].

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import facets_for_RP4
sage: A = facets_for_RP4()    # long time (1 or 2 seconds)
sage: SimplicialComplex(A) == simplicial_complexes.RealProjectiveSpace(4) # long␣
↪time
True
```

sage.homology.simplicial_complex. **lattice_paths** (*t1*, *t2*, *length=None*)

Given lists (or tuples or ...) `t1` and `t2`, think of them as labelings for vertices: `t1` labeling points on the x-axis, `t2` labeling points on the y-axis, both increasing. Return the list of rectilinear paths along the grid defined by these points in the plane, starting from `(t1[0],t2[0])`, ending at `(t1[last],t2[last])`, and at each grid point, going either right or up. See the examples.

Parameters

- **t1** (*tuple, list, other iterable*) – labeling for vertices

- **t2** (*tuple, list, other iterable*) – labeling for vertices
- **length** (integer or None ; optional, default None) – if not None , then an integer, the length of the desired path.

Returns list of lists of vertices making up the paths as described above

Return type list of lists

This is used when triangulating the product of simplices. The optional argument length is used for Δ-complexes, to specify all simplices in a product: in the triangulation of a product of two simplices, there is a d-simplex for every path of length $d+1$ in the lattice. The path must start at the bottom left and end at the upper right, and it must use at least one point in each row and in each column, so if length is too small, there will be no paths.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import lattice_paths
sage: lattice_paths([0,1,2], [0,1,2])
[[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)],
 [(0, 0), (0, 1), (1, 1), (1, 2), (2, 2)],
 [(0, 0), (1, 0), (1, 1), (1, 2), (2, 2)],
 [(0, 0), (0, 1), (1, 1), (2, 1), (2, 2)],
 [(0, 0), (1, 0), (1, 1), (2, 1), (2, 2)],
 [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)]]
sage: lattice_paths(('a', 'b', 'c'), (0, 3, 5))
[[('a', 0), ('a', 3), ('a', 5), ('b', 5), ('c', 5)],
 [('a', 0), ('a', 3), ('b', 3), ('b', 5), ('c', 5)],
 [('a', 0), ('b', 0), ('b', 3), ('b', 5), ('c', 5)],
 [('a', 0), ('a', 3), ('b', 3), ('c', 3), ('c', 5)],
 [('a', 0), ('b', 0), ('b', 3), ('c', 3), ('c', 5)],
 [('a', 0), ('b', 0), ('c', 0), ('c', 3), ('c', 5)]]
sage: lattice_paths(list(range(3)), list(range(3)), length=2)
[]
sage: lattice_paths(list(range(3)), list(range(3)), length=3)
[[(0, 0), (1, 1), (2, 2)]]
sage: lattice_paths(list(range(3)), list(range(3)), length=4)
[[(0, 0), (1, 1), (1, 2), (2, 2)],
 [(0, 0), (0, 1), (1, 2), (2, 2)],
 [(0, 0), (1, 1), (2, 1), (2, 2)],
 [(0, 0), (1, 0), (2, 1), (2, 2)],
 [(0, 0), (0, 1), (1, 1), (2, 2)],
 [(0, 0), (1, 0), (1, 1), (2, 2)]]
```

sage.homology.simplicial_complex. **rename_vertex** (*n*, *keep*, *left=True*)

Rename a vertex: the vertices from the list keep get relabeled 0, 1, 2, ..., in order. Any other vertex (e.g. 4) gets renamed to by prepending an 'L' or an 'R' (thus to either 'L4' or 'R4'), depending on whether the argument left is True or False .

Parameters

- **n** – a 'vertex': either an integer or a string
- **keep** – a list of three vertices
- **left** (boolean; optional, default True) – if True , rename for use in left factor

This is used by the *connected_sum()* method for simplicial complexes.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import rename_vertex
sage: rename_vertex(6, [5, 6, 7])
1
sage: rename_vertex(3, [5, 6, 7, 8, 9])
'L3'
sage: rename_vertex(3, [5, 6, 7], left=False)
'R3'
```

CHAPTER

SEVEN

MORPHISMS OF SIMPLICIAL COMPLEXES

AUTHORS:

- Benjamin Antieau <d.ben.antieau@gmail.com> (2009.06)
- Travis Scrimshaw (2012-08-18): Made all simplicial complexes immutable to work with the homset cache.

This module implements morphisms of simplicial complexes. The input is given by a dictionary on the vertex set of a simplicial complex. The initialization checks that faces are sent to faces.

There is also the capability to create the fiber product of two morphisms with the same codomain.

EXAMPLES:

```
sage: S = SimplicialComplex([[0,2],[1,5],[3,4]], is_mutable=False)
sage: H = Hom(S,S.product(S, is_mutable=False))
sage: H.diagonal_morphism()
Simplicial complex morphism:
  From: Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and facets {(3, 4), (1,␣
↪5), (0, 2)}
  To: Simplicial complex with 36 vertices and 18 facets
  Defn: [0, 1, 2, 3, 4, 5] --> ['L0R0', 'L1R1', 'L2R2', 'L3R3', 'L4R4', 'L5R5']

sage: S = SimplicialComplex([[0,2],[1,5],[3,4]], is_mutable=False)
sage: T = SimplicialComplex([[0,2],[1,3]], is_mutable=False)
sage: f = {0:0,1:1,2:2,3:1,4:3,5:3}
sage: H = Hom(S,T)
sage: x = H(f)
sage: x.image()
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 3), (0, 2)}
sage: x.is_surjective()
True
sage: x.is_injective()
False
sage: x.is_identity()
False

sage: S = simplicial_complexes.Sphere(2)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: i.image()
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 2), (1,␣
↪2, 3), (0, 1, 3)}
sage: i.is_surjective()
True
sage: i.is_injective()
True
```

```
sage: i.is_identity()
True

sage: S = simplicial_complexes.Sphere(2)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: j = i.fiber_product(i)
sage: j
Simplicial complex morphism:
  From: Simplicial complex with 4 vertices and 4 facets
  To:   Minimal triangulation of the 2-sphere
  Defn: L1R1 |--> 1
        L3R3 |--> 3
        L2R2 |--> 2
        L0R0 |--> 0
sage: S = simplicial_complexes.Sphere(2)
sage: T = S.product(SimplicialComplex([[0,1]]), rename_vertices = False, is_
↪mutable=False)
sage: H = Hom(T,S)
sage: T
Simplicial complex with 8 vertices and 12 facets
sage: T.vertices()
((0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (3, 0), (3, 1))
sage: f = {(0, 0): 0, (0, 1): 0, (1, 0): 1, (1, 1): 1, (2, 0): 2, (2, 1): 2, (3, 0):␣
↪3, (3, 1): 3}
sage: x = H(f)
sage: U = simplicial_complexes.Sphere(1)
sage: G = Hom(U,S)
sage: U
Minimal triangulation of the 1-sphere
sage: g = {0:0,1:1,2:2}
sage: y = G(g)
sage: z = y.fiber_product(x)
sage: z                                          # this is the mapping path space
Simplicial complex morphism:
  From: Simplicial complex with 6 vertices and 6 facets
  To:   Minimal triangulation of the 2-sphere
  Defn: ['L2R(2, 0)', 'L2R(2, 1)', 'L0R(0, 0)', 'L0R(0, 1)', 'L1R(1, 0)', 'L1R(1, 1)
↪'] --> [2, 2, 0, 0, 1, 1]
```

class sage.homology.simplicial_complex_morphism. **SimplicialComplexMorphism**(*f*, *X*, *Y*)

Bases: sage.categories.morphism.Morphism

An element of this class is a morphism of simplicial complexes.

associated_chain_complex_morphism(*base_ring=Integer Ring*, *augmented=False*, *cochain=False*)

Returns the associated chain complex morphism of `self`.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(1)
sage: T = simplicial_complexes.Sphere(2)
sage: H = Hom(S,T)
sage: f = {0:0,1:1,2:2}
sage: x = H(f)
sage: x
```

```
Simplicial complex morphism:
  From: Minimal triangulation of the 1-sphere
  To:   Minimal triangulation of the 2-sphere
  Defn: 0 |--> 0
        1 |--> 1
        2 |--> 2
sage: a = x.associated_chain_complex_morphism()
sage: a
Chain complex morphism:
  From: Chain complex with at most 2 nonzero terms over Integer Ring
  To:   Chain complex with at most 3 nonzero terms over Integer Ring
sage: a._matrix_dictionary
{0: [1 0 0]
 [0 1 0]
 [0 0 1]
 [0 0 0], 1: [1 0 0]
 [0 1 0]
 [0 0 0]
 [0 0 1]
 [0 0 0]
 [0 0 0], 2: []}
sage: x.associated_chain_complex_morphism(augmented=True)
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Integer Ring
  To:   Chain complex with at most 4 nonzero terms over Integer Ring
sage: x.associated_chain_complex_morphism(cochain=True)
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Integer Ring
  To:   Chain complex with at most 2 nonzero terms over Integer Ring
sage: x.associated_chain_complex_morphism(augmented=True,cochain=True)
Chain complex morphism:
  From: Chain complex with at most 4 nonzero terms over Integer Ring
  To:   Chain complex with at most 3 nonzero terms over Integer Ring
sage: x.associated_chain_complex_morphism(base_ring=GF(11))
Chain complex morphism:
  From: Chain complex with at most 2 nonzero terms over Finite Field of size␣
 ↪11
  To:   Chain complex with at most 3 nonzero terms over Finite Field of size␣
 ↪11
```

Some simplicial maps which reverse the orientation of a few simplices:

```
sage: g = {0:1, 1:2, 2:0}
sage: H(g).associated_chain_complex_morphism()._matrix_dictionary
{0: [0 0 1]
 [1 0 0]
 [0 1 0]
 [0 0 0], 1: [ 0 -1  0]
 [ 0  0 -1]
 [ 0  0  0]
 [ 1  0  0]
 [ 0  0  0]
 [ 0  0  0], 2: []}
sage: X = SimplicialComplex([[0, 1]], is_mutable=False)
sage: Hom(X,X)({0:1, 1:0}).associated_chain_complex_morphism()._matrix_
 ↪dictionary
{0: [0 1]
 [1 0], 1: [-1]}
```

fiber_product (*other*, *rename_vertices=True*)

Fiber product of `self` and `other`. Both morphisms should have the same codomain. The method returns a morphism of simplicial complexes, which is the morphism from the space of the fiber product to the codomain.

EXAMPLES:

```
sage: S = SimplicialComplex([[0,1],[1,2]], is_mutable=False)
sage: T = SimplicialComplex([[0,2],[1]], is_mutable=False)
sage: U = SimplicialComplex([[0,1],[2]], is_mutable=False)
sage: H = Hom(S,U)
sage: G = Hom(T,U)
sage: f = {0:0,1:1,2:0}
sage: g = {0:0,1:1,2:1}
sage: x = H(f)
sage: y = G(g)
sage: z = x.fiber_product(y)
sage: z
Simplicial complex morphism:
  From: Simplicial complex with 4 vertices and facets {('L2R0',), ('L1R1',), (
↪'L0R0', 'L1R2')}
  To:   Simplicial complex with vertex set (0, 1, 2) and facets {(2,), (0, 1)}
  Defn: L1R2 |--> 1
        L1R1 |--> 1
        L2R0 |--> 0
        L0R0 |--> 0
```

image ( )

Computes the image simplicial complex of f.

EXAMPLES:

```
sage: S = SimplicialComplex([[0,1],[2,3]], is_mutable=False)
sage: T = SimplicialComplex([[0,1]], is_mutable=False)
sage: f = {0:0,1:1,2:0,3:1}
sage: H = Hom(S,T)
sage: x = H(f)
sage: x.image()
Simplicial complex with vertex set (0, 1) and facets {(0, 1)}

sage: S = SimplicialComplex(is_mutable=False)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: i.image()
Simplicial complex with vertex set () and facets {()}
sage: i.is_surjective()
True
sage: S = SimplicialComplex([[0,1]], is_mutable=False)
sage: T = SimplicialComplex([[0,1], [0,2]], is_mutable=False)
sage: f = {0:0,1:1}
sage: g = {0:0,1:1}
sage: k = {0:0,1:2}
sage: H = Hom(S,T)
sage: x = H(f)
sage: y = H(g)
sage: z = H(k)
sage: x == y
True
sage: x == z
```

```
False
sage: x.image()
Simplicial complex with vertex set (0, 1) and facets {(0, 1)}
sage: y.image()
Simplicial complex with vertex set (0, 1) and facets {(0, 1)}
sage: z.image()
Simplicial complex with vertex set (0, 2) and facets {(0, 2)}
```

induced_homology_morphism (*base_ring=None*, *cohomology=False*)

The map in (co)homology induced by this map

INPUT:

- `base_ring` – must be a field (optional, default `QQ`)
- `cohomology` – boolean (optional, default `False`). If `True` , the map induced in cohomology rather than homology.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(1)
sage: T = S.product(S, is_mutable=False)
sage: H = Hom(S,T)
sage: diag = H.diagonal_morphism()
sage: h = diag.induced_homology_morphism(QQ)
sage: h
Graded vector space morphism:
  From: Homology module of Minimal triangulation of the 1-sphere over␣
 ↪Rational Field
  To:   Homology module of Simplicial complex with 9 vertices and 18 facets␣
 ↪over Rational Field
  Defn: induced by:
    Simplicial complex morphism:
      From: Minimal triangulation of the 1-sphere
      To:   Simplicial complex with 9 vertices and 18 facets
      Defn: 0 |--> L0R0
            1 |--> L1R1
            2 |--> L2R2
```

We can view the matrix form for the homomorphism:

```
sage: h.to_matrix(0) # in degree 0
[1]
sage: h.to_matrix(1) # in degree 1
[1]
[0]
sage: h.to_matrix()  # the entire homomorphism
[1|0]
[-+-]
[0|1]
[0|0]
[-+-]
[0|0]
```

We can evaluate it on (co)homology classes:

```
sage: coh = diag.induced_homology_morphism(QQ, cohomology=True)
sage: coh.to_matrix(1)
[1 0]
```

```
sage: x,y = list(T.cohomology_ring(QQ).basis(1))
sage: coh(x)
h^{1,0}
sage: coh(2*x+3*y)
2*h^{1,0}
```

Note that the complexes must be immutable for this to work. Many, but not all, complexes are immutable when constructed:

```
sage: S.is_immutable()
True
sage: S.barycentric_subdivision().is_immutable()
False
sage: S2 = S.suspension()
sage: S2.is_immutable()
False
sage: h = Hom(S,S2)({0: 0, 1:1, 2:2}).induced_homology_morphism()
Traceback (most recent call last):
...
ValueError: the domain and codomain complexes must be immutable
sage: S2.set_immutable(); S2.is_immutable()
True
sage: h = Hom(S,S2)({0: 0, 1:1, 2:2}).induced_homology_morphism()
```

is_contiguous_to (*other*)

Return `True` if `self` is contiguous to `other` .

Two morphisms $f_0, f_1 : K \to L$ are *contiguous* if for any simplex $\sigma \in K$, the union $f_0(\sigma) \cup f_1(\sigma)$ is a simplex in L. This is not a transitive relation, but it induces an equivalence relation on simplicial maps: f is equivalent to g if there is a finite sequence $f_0 = f$, f_1, ..., $f_n = g$ such that f_i and f_{i+1} are contiguous for each i.

This is related to maps being homotopic: if they are contiguous, then they induce homotopic maps on the geometric realizations. Given two homotopic maps on the geometric realizations, then after barycentrically subdividing n times for some n, the maps have simplicial approximations which are in the same contiguity class. (This last fact is only true if the domain is a *finite* simplicial complex, by the way.)

See Section 3.5 of Spanier [Spa1966] for details.

ALGORITHM:

It is enough to check when σ ranges over the facets.

INPUT:

- `other` – a simplicial complex morphism with the same domain and codomain as `self`

EXAMPLES:

```
sage: K = simplicial_complexes.Simplex(1)
sage: L = simplicial_complexes.Sphere(1)
sage: H = Hom(K, L)
sage: f = H({0: 0, 1: 1})
sage: g = H({0: 0, 1: 0})
sage: f.is_contiguous_to(f)
True
sage: f.is_contiguous_to(g)
True
sage: h = H({0: 1, 1: 2})
```

```
sage: f.is_contiguous_to(h)
False
```

TESTS:

```
sage: one = Hom(K,K).identity()
sage: one.is_contiguous_to(f)
False
sage: one.is_contiguous_to(3) # nonsensical input
False
```

is_identity ( )

If `self` is an identity morphism, returns `True` . Otherwise, `False` .

EXAMPLES:

```
sage: T = simplicial_complexes.Sphere(1)
sage: G = Hom(T,T)
sage: T
Minimal triangulation of the 1-sphere
sage: j = G({0:0,1:1,2:2})
sage: j.is_identity()
True

sage: S = simplicial_complexes.Sphere(2)
sage: T = simplicial_complexes.Sphere(3)
sage: H = Hom(S,T)
sage: f = {0:0,1:1,2:2,3:3}
sage: x = H(f)
sage: x
Simplicial complex morphism:
  From: Minimal triangulation of the 2-sphere
  To:   Minimal triangulation of the 3-sphere
  Defn: 0 |--> 0
        1 |--> 1
        2 |--> 2
        3 |--> 3
sage: x.is_identity()
False
```

is_injective ( )

Returns `True` if and only if `self` is injective.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(1)
sage: T = simplicial_complexes.Sphere(2)
sage: U = simplicial_complexes.Sphere(3)
sage: H = Hom(T,S)
sage: G = Hom(T,U)
sage: f = {0:0,1:1,2:0,3:1}
sage: x = H(f)
sage: g = {0:0,1:1,2:2,3:3}
sage: y = G(g)
sage: x.is_injective()
False
sage: y.is_injective()
True
```

is_surjective ( )

Returns `True` if and only if `self` is surjective.

EXAMPLES:

```
sage: S = SimplicialComplex([(0,1,2)], is_mutable=False)
sage: S
Simplicial complex with vertex set (0, 1, 2) and facets {(0, 1, 2)}
sage: T = SimplicialComplex([(0,1)], is_mutable=False)
sage: T
Simplicial complex with vertex set (0, 1) and facets {(0, 1)}
sage: H = Hom(S,T)
sage: x = H({0:0,1:1,2:1})
sage: x.is_surjective()
True

sage: S = SimplicialComplex([[0,1],[2,3]], is_mutable=False)
sage: T = SimplicialComplex([[0,1]], is_mutable=False)
sage: f = {0:0,1:1,2:0,3:1}
sage: H = Hom(S,T)
sage: x = H(f)
sage: x.is_surjective()
True
```

mapping_torus ( )

The mapping torus of a simplicial complex endomorphism

The mapping torus is the simplicial complex formed by taking the product of the domain of `self` with a 4 point interval $[I_0, I_1, I_2, I_3]$ and identifying vertices of the form (I_0, v) with (I_3, w) where w is the image of v under the given morphism.

See Wikipedia article Mapping torus

EXAMPLES:

```
sage: C = simplicial_complexes.Sphere(1)             # Circle
sage: T = Hom(C,C).identity().mapping_torus() ; T    # Torus
Simplicial complex with 9 vertices and 18 facets
sage: T.homology() == simplicial_complexes.Torus().homology()
True

sage: f = Hom(C,C)({0:0,1:2,2:1})
sage: K = f.mapping_torus() ; K  # Klein Bottle
Simplicial complex with 9 vertices and 18 facets
sage: K.homology() == simplicial_complexes.KleinBottle().homology()
True
```

TESTS:

```
sage: g = Hom(simplicial_complexes.Simplex([1]),C)({1:0})
sage: g.mapping_torus()
Traceback (most recent call last):
...
ValueError: self must have the same domain and codomain.
```

sage.homology.simplicial_complex_morphism. **is_SimplicialComplexMorphism** (*x*)

Returns `True` if and only if `x` is a morphism of simplicial complexes.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex_morphism import is_
↪SimplicialComplexMorphism
sage: S = SimplicialComplex([[0,1],[3,4]], is_mutable=False)
sage: H = Hom(S,S)
sage: f = {0:0,1:1,3:3,4:4}
sage: x = H(f)
sage: is_SimplicialComplexMorphism(x)
True
```

CHAPTER

EIGHT

HOMSETS BETWEEN SIMPLICIAL COMPLEXES

AUTHORS:

- Travis Scrimshaw (2012-08-18): Made all simplicial complexes immutable to work with the homset cache.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(1)
sage: T = simplicial_complexes.Sphere(2)
sage: H = Hom(S,T)
sage: f = {0:0,1:1,2:3}
sage: x = H(f)
sage: x
Simplicial complex morphism:
  From: Minimal triangulation of the 1-sphere
  To: Minimal triangulation of the 2-sphere
  Defn: 0 |--> 0
        1 |--> 1
        2 |--> 3
sage: x.is_injective()
True
sage: x.is_surjective()
False
sage: x.image()
Simplicial complex with vertex set (0, 1, 3) and facets {(1, 3), (0, 3), (0, 1)}
sage: from sage.homology.simplicial_complex import Simplex
sage: s = Simplex([1,2])
sage: x(s)
(1, 3)
```

TESTS:

```
sage: S = simplicial_complexes.Sphere(1)
sage: T = simplicial_complexes.Sphere(2)
sage: H = Hom(S,T)
sage: loads(dumps(H)) == H
True
```

class sage.homology.simplicial_complex_homset. **SimplicialComplexHomset** (*X*, *Y*, *category=None*, *base=None*, *check=True*)

Bases: sage.categories.homset.Homset

TESTS:

```
sage: X = ZZ['x']; X.rename("X")
sage: Y = ZZ['y']; Y.rename("Y")
sage: class MyHomset(Homset):
....:     def my_function(self, x):
....:         return Y(x[0])
....:     def _an_element_(self):
....:         return sage.categories.morphism.SetMorphism(self, self.my_function)
...
sage: import __main__; __main__.MyHomset = MyHomset # fakes MyHomset being␣
↪defined in a Python module
sage: H = MyHomset(X, Y, category=Monoids(), base = ZZ)
sage: H
Set of Morphisms from X to Y in Category of monoids
sage: TestSuite(H).run()

sage: H = MyHomset(X, Y, category=1, base = ZZ)
Traceback (most recent call last):
...
TypeError: category (=1) must be a category

sage: H
Set of Morphisms from X to Y in Category of monoids
sage: TestSuite(H).run()
sage: H = MyHomset(X, Y, category=1, base = ZZ, check = False)
Traceback (most recent call last):
...
AttributeError: 'sage.rings.integer.Integer' object has no attribute 'Homsets'
sage: P.<t> = ZZ[]
sage: f = P.hom([1/2*t])
sage: f.parent().domain()
Univariate Polynomial Ring in t over Integer Ring
sage: f.domain() is f.parent().domain()
True
```

Test that `base_ring` is initialized properly:

```
sage: R = QQ['x']
sage: Hom(R, R).base_ring()
Rational Field
sage: Hom(R, R, category=Sets()).base_ring()
sage: Hom(R, R, category=Modules(QQ)).base_ring()
Rational Field
sage: Hom(QQ^3, QQ^3, category=Modules(QQ)).base_ring()
Rational Field
```

For whatever it's worth, the `base` arguments takes precedence:

```
sage: MyHomset(ZZ^3, ZZ^3, base = QQ).base_ring()
Rational Field
```

an_element ( )

Return a (non-random) element of `self` .

EXAMPLES:

```
sage: S = simplicial_complexes.KleinBottle()
sage: T = simplicial_complexes.Sphere(5)
sage: H = Hom(S,T)
```

```
sage: x = H.an_element()
sage: x
Simplicial complex morphism:
  From: Minimal triangulation of the Klein bottle
  To:   Minimal triangulation of the 5-sphere
  Defn: [0, 1, 2, 3, 4, 5, 6, 7] --> [0, 0, 0, 0, 0, 0, 0, 0]
```

diagonal_morphism (*rename_vertices=True*)

Return the diagonal morphism in $Hom(S, S \times S)$.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(2)
sage: H = Hom(S,S.product(S, is_mutable=False))
sage: d = H.diagonal_morphism()
sage: d
Simplicial complex morphism:
  From: Minimal triangulation of the 2-sphere
  To:   Simplicial complex with 16 vertices and 96 facets
  Defn: 0 |--> L0R0
        1 |--> L1R1
        2 |--> L2R2
        3 |--> L3R3

sage: T = SimplicialComplex([[0], [1]], is_mutable=False)
sage: U = T.product(T,rename_vertices = False, is_mutable=False)
sage: G = Hom(T,U)
sage: e = G.diagonal_morphism(rename_vertices = False)
sage: e
Simplicial complex morphism:
  From: Simplicial complex with vertex set (0, 1) and facets {(0,), (1,)}
  To:   Simplicial complex with 4 vertices and facets {((1, 1),), ((1, 0),),␣
↪((0, 0),), ((0, 1),)}
  Defn: 0 |--> (0, 0)
        1 |--> (1, 1)
```

identity ( )

Return the identity morphism of $Hom(S, S)$.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(2)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: i.is_identity()
True

sage: T = SimplicialComplex([[0,1]], is_mutable=False)
sage: G = Hom(T,T)
sage: G.identity()
Simplicial complex endomorphism of Simplicial complex with vertex set (0, 1)␣
↪and facets {(0, 1)}
  Defn: 0 |--> 0
        1 |--> 1
```

sage.homology.simplicial_complex_homset. **is_SimplicialComplexHomset** (*x*)

Return `True` if and only if `x` is a simplicial complex homspace.

EXAMPLES:

```
sage: S = SimplicialComplex(is_mutable=False)
sage: T = SimplicialComplex(is_mutable=False)
sage: H = Hom(S, T)
sage: H
Set of Morphisms from Simplicial complex with vertex set () and facets {()}
 to Simplicial complex with vertex set () and facets {()}
 in Category of finite simplicial complexes
sage: from sage.homology.simplicial_complex_homset import is_
 ↪SimplicialComplexHomset
sage: is_SimplicialComplexHomset(H)
True
```

CHAPTER

NINE

EXAMPLES OF SIMPLICIAL COMPLEXES

There are two main types: manifolds and examples related to graph theory.

For manifolds, there are functions defining the n-sphere for any n, the torus, n-dimensional real projective space for any n, the complex projective plane, surfaces of arbitrary genus, and some other manifolds, all as simplicial complexes.

Aside from surfaces, this file also provides functions for constructing some other simplicial complexes: the simplicial complex of not-i-connected graphs on n vertices, the matching complex on n vertices, the chessboard complex for an n by i chessboard, and others. These provide examples of large simplicial complexes; for example, `simplicial_complexes.NotIConnectedGraphs(7,2)` has over a million simplices.

All of these examples are accessible by typing `simplicial_complexes.NAME` , where `NAME` is the name of the example.

- *BarnetteSphere()*
- *BrucknerGrunbaumSphere()*
- *ChessboardComplex()*
- *ComplexProjectivePlane()*
- *DunceHat()*
- *K3Surface()*
- *KleinBottle()*
- *MatchingComplex()*
- *MooreSpace()*
- *NotIConnectedGraphs()*
- *PoincareHomologyThreeSphere()*
- *PseudoQuaternionicProjectivePlane()*
- *RandomComplex()*
- *RandomTwoSphere()*
- *RealProjectivePlane()*
- *RealProjectiveSpace()*
- *RudinBall()*
- *ShiftedComplex()*
- *Simplex()*
- *Sphere()*

- *SumComplex()*
- *SurfaceOfGenus()*
- *Torus()*
- *ZieglerBall()*

You can also get a list by typing `simplicial_complexes.` and hitting the TAB key.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(2) # the 2-sphere
sage: S.homology()
{0: 0, 1: 0, 2: Z}
sage: simplicial_complexes.SurfaceOfGenus(3)
Triangulation of an orientable surface of genus 3
sage: M4 = simplicial_complexes.MooreSpace(4)
sage: M4.homology()
{0: 0, 1: C4, 2: 0}
sage: simplicial_complexes.MatchingComplex(6).homology()
{0: 0, 1: Z^16, 2: 0}
```

sage.homology.examples. **BarnetteSphere** ( )

Returns Barnette's triangulation of the 3-sphere.

This is a pure simplicial complex of dimension 3 with 8 vertices and 19 facets, which is a non-polytopal triangulation of the 3-sphere. It was constructed by Barnette in [Bar1970]. The construction here uses the labeling from De Loera, Rambau and Santos [DLRS2010]. Another reference is chapter III.4 of Ewald [Ewa1996].

EXAMPLES:

```
sage: BS = simplicial_complexes.BarnetteSphere() ; BS
Barnette's triangulation of the 3-sphere
sage: BS.f_vector()
[1, 8, 27, 38, 19]
```

TESTS:

Checks that this is indeed the same Barnette Sphere as the one given on page 87 of [Ewa1996].:

```
sage: BS2 = SimplicialComplex([[1,2,3,4],[3,4,5,6],[1,2,5,6],
....:                          [1,2,4,7],[1,3,4,7],[3,4,6,7],
....:                          [3,5,6,7],[1,2,5,7],[2,5,6,7],
....:                          [2,4,6,7],[1,2,3,8],[2,3,4,8],
....:                          [3,4,5,8],[4,5,6,8],[1,2,6,8],
....:                          [1,5,6,8],[1,3,5,8],[2,4,6,8],
....:                          [1,3,5,7]])
sage: BS.is_isomorphic(BS2)
True
```

sage.homology.examples. **BrucknerGrunbaumSphere** ( )

Returns Bruckner and Grunbaum's triangulation of the 3-sphere.

This is a pure simplicial complex of dimension 3 with 8 vertices and 20 facets, which is a non-polytopal triangulation of the 3-sphere. It appeared first in [Br1910] and was studied in [GrS1967].

It is defined here as the link of any vertex in the unique minimal triangulation of the complex projective plane, see chapter 4 of [Kuh1995].

EXAMPLES:

```
sage: BGS = simplicial_complexes.BrucknerGrunbaumSphere() ; BGS
Bruckner and Grunbaum's triangulation of the 3-sphere
sage: BGS.f_vector()
[1, 8, 28, 40, 20]
```

sage.homology.examples. **ChessboardComplex** (*n*, *i*)

The chessboard complex for an $n \times i$ chessboard.

Fix integers $n, i > 0$ and consider sets V of n vertices and W of i vertices. A 'partial matching' between V and W is a graph formed by edges (v, w) with $v \in V$ and $w \in W$ so that each vertex is in at most one edge. If G is a partial matching, then so is any graph obtained by deleting edges from G. Thus the set of all partial matchings on V and W, viewed as a set of subsets of the $n + i$ choose 2 possible edges, is closed under taking subsets, and thus forms a simplicial complex called the 'chessboard complex'. This function produces that simplicial complex. (It is called the chessboard complex because such graphs also correspond to ways of placing rooks on an n by i chessboard so that none of them are attacking each other.)

INPUT:

- `n, i` – positive integers.

See Dumas et al. [DHSW2003] for information on computing its homology by computer, and see Wachs [Wac2003] for an expository article about the theory.

EXAMPLES:

```
sage: C = simplicial_complexes.ChessboardComplex(5,5)
sage: C.f_vector()
[1, 25, 200, 600, 600, 120]
sage: simplicial_complexes.ChessboardComplex(3,3).homology()
{0: 0, 1: Z x Z x Z x Z, 2: 0}
```

sage.homology.examples. **ComplexProjectivePlane** ( )

A minimal triangulation of the complex projective plane.

This was constructed by Kühnel and Banchoff [KB1983].

EXAMPLES:

```
sage: C = simplicial_complexes.ComplexProjectivePlane()
sage: C.f_vector()
[1, 9, 36, 84, 90, 36]
sage: C.homology(2)
Z
sage: C.homology(4)
Z
```

sage.homology.examples. **DunceHat** ( )

Return the minimal triangulation of the dunce hat given by Hachimori [Hac2016].

This is a standard example of a space that is contractible but not collapsible.

EXAMPLES:

```
sage: D = simplicial_complexes.DunceHat(); D
Minimal triangulation of the dunce hat
sage: D.f_vector()
[1, 8, 24, 17]
sage: D.homology()
{0: 0, 1: 0, 2: 0}
```

```
sage: D.is_cohen_macaulay()
True
```

sage.homology.examples. **K3Surface** ( )

Returns a minimal triangulation of the K3 surface.

This is a pure simplicial complex of dimension 4 with 16 vertices and 288 facets. It was constructed by Casella and Kühnel in [CK2001]. The construction here uses the labeling from Spreer and Kühnel [SK2011].

EXAMPLES:

```
sage: K3=simplicial_complexes.K3Surface() ; K3
Minimal triangulation of the K3 surface
sage: K3.f_vector()
[1, 16, 120, 560, 720, 288]
```

This simplicial complex is implemented just by listing all 288 facets. The list of facets can be computed by the function *facets_for_K3()* , but running the function takes a few seconds.

sage.homology.examples. **KleinBottle** ( )

A minimal triangulation of the Klein bottle, as presented for example in Davide Cervone's thesis [Cer1994].

EXAMPLES:

```
sage: simplicial_complexes.KleinBottle()
Minimal triangulation of the Klein bottle
```

sage.homology.examples. **MatchingComplex** (n)

The matching complex of graphs on n vertices.

Fix an integer $n > 0$ and consider a set V of n vertices. A 'partial matching' on V is a graph formed by edges so that each vertex is in at most one edge. If G is a partial matching, then so is any graph obtained by deleting edges from G. Thus the set of all partial matchings on n vertices, viewed as a set of subsets of the n choose 2 possible edges, is closed under taking subsets, and thus forms a simplicial complex called the 'matching complex'. This function produces that simplicial complex.

INPUT:

- n – positive integer.

See Dumas et al. [DHSW2003] for information on computing its homology by computer, and see Wachs [Wac2003] for an expository article about the theory. For example, the homology of these complexes seems to have only mod 3 torsion, and this has been proved for the bottom non-vanishing homology group for the matching complex M_n.

EXAMPLES:

```
sage: M = simplicial_complexes.MatchingComplex(7)
sage: H = M.homology()
sage: H
{0: 0, 1: C3, 2: Z^20}
sage: H[2].ngens()
20
sage: simplicial_complexes.MatchingComplex(8).homology(2)  # long time (6s on␣
↪sage.math, 2012)
Z^132
```

sage.homology.examples. **MooreSpace** (q)

Triangulation of the mod q Moore space.

INPUT:

- `q` -0 integer, at least 2

This is a simplicial complex with simplices of dimension 0, 1, and 2, such that its reduced homology is isomorphic to $\mathbf{Z}/q\mathbf{Z}$ in dimension 1, zero otherwise.

If $q = 2$, this is the real projective plane. If $q > 2$, then construct it as follows: start with a triangle with vertices 1, 2, 3. We take a $3q$-gon forming a q-fold cover of the triangle, and we form the resulting complex as an identification space of the $3q$-gon. To triangulate this identification space, put q vertices A_0, ..., A_{q-1}, in the interior, each of which is connected to 1, 2, 3 (two facets each: $[1, 2, A_i]$, $[2, 3, A_i]$). Put q more vertices in the interior: B_0, ..., B_{q-1}, with facets $[3, 1, B_i]$, $[3, B_i, A_i]$, $[1, B_i, A_{i+1}]$, $[B_i, A_i, A_{i+1}]$. Then triangulate the interior polygon with vertices A_0, A_1, ..., A_{q-1}.

EXAMPLES:

```
sage: simplicial_complexes.MooreSpace(2)
Minimal triangulation of the real projective plane
sage: simplicial_complexes.MooreSpace(3).homology()[1]
C3
sage: simplicial_complexes.MooreSpace(4).suspension().homology()[2]
C4
sage: simplicial_complexes.MooreSpace(8)
Triangulation of the mod 8 Moore space
```

sage.homology.examples. **NotIConnectedGraphs** (*n*, *i*)

The simplicial complex of all graphs on n vertices which are not i-connected.

Fix an integer $n > 0$ and consider the set of graphs on n vertices. View each graph as its set of edges, so it is a subset of a set of size n choose 2. A graph is i-connected if, for any $j < i$, if any j vertices are removed along with the edges emanating from them, then the graph remains connected. Now fix i: it is clear that if G is not i-connected, then the same is true for any graph obtained from G by deleting edges. Thus the set of all graphs which are not i-connected, viewed as a set of subsets of the n choose 2 possible edges, is closed under taking subsets, and thus forms a simplicial complex. This function produces that simplicial complex.

INPUT:

- `n` , `i` – non-negative integers with i at most n

See Dumas et al. [DHSW2003] for information on computing its homology by computer, and see Babson et al. [BBLSW1999] for theory. For example, Babson et al. show that when $i = 2$, the reduced homology of this complex is nonzero only in dimension $2n - 5$, where it is free abelian of rank $(n - 2)!$.

EXAMPLES:

```
sage: simplicial_complexes.NotIConnectedGraphs(5,2).f_vector()
[1, 10, 45, 120, 210, 240, 140, 20]
sage: simplicial_complexes.NotIConnectedGraphs(5,2).homology(5).ngens()
6
```

sage.homology.examples. **PoincareHomologyThreeSphere** ( )

A triangulation of the Poincare homology 3-sphere.

This is a manifold whose integral homology is identical to the ordinary 3-sphere, but it is not simply connected. In particular, its fundamental group is the binary icosahedral group, which has order 120. The triangulation given here has 16 vertices and is due to Björner and Lutz [BL2000].

EXAMPLES:

```
sage: S3 = simplicial_complexes.Sphere(3)
sage: Sigma3 = simplicial_complexes.PoincareHomologyThreeSphere()
```

```
sage: S3.homology() == Sigma3.homology()
True
sage: Sigma3.fundamental_group().cardinality() # long time
120
```

sage.homology.examples. **ProjectivePlane** ( )

A minimal triangulation of the real projective plane.

EXAMPLES:

```
sage: P = simplicial_complexes.RealProjectivePlane()
sage: Q = simplicial_complexes.ProjectivePlane()
sage: P == Q
True
sage: P.cohomology(1)
0
sage: P.cohomology(2)
C2
sage: P.cohomology(1, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
sage: P.cohomology(2, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
```

sage.homology.examples. **PseudoQuaternionicProjectivePlane** ( )

Returns a pure simplicial complex of dimension 8 with 490 facets.

Warning: This is expected to be a triangulation of the projective plane HP^2 over the ring of quaternions, but this has not been proved yet.

This simplicial complex has the same homology as HP^2. Its automorphism group is isomorphic to the alternating group A_5 and acts transitively on vertices.

This is defined here using the description in [BK1992]. This article deals with three different triangulations. This procedure returns the only one which has a transitive group of automorphisms.

EXAMPLES:

```
sage: HP2 = simplicial_complexes.PseudoQuaternionicProjectivePlane() ; HP2
Simplicial complex with 15 vertices and 490 facets
sage: HP2.f_vector()
[1, 15, 105, 455, 1365, 3003, 4515, 4230, 2205, 490]
```

Checking its automorphism group:

```
sage: HP2.automorphism_group().is_isomorphic(AlternatingGroup(5))
True
```

sage.homology.examples. **RandomComplex** (*n*, *d*, *p=0.5*)

A random `d` -dimensional simplicial complex on `n` vertices.

INPUT:

- `n` – number of vertices
- `d` – dimension of the complex
- `p` – floating point number between 0 and 1 (optional, default 0.5)

A random d-dimensional simplicial complex on n vertices, as defined for example by Meshulam and Wallach [MW2009], is constructed as follows: take n vertices and include all of the simplices of dimension strictly less than d, and then for each possible simplex of dimension d, include it with probability p.

EXAMPLES:

```
sage: X = simplicial_complexes.RandomComplex(6, 2); X
Random 2-dimensional simplicial complex on 6 vertices
sage: len(list(X.vertices()))
6
```

If d is too large (if $d + 1 > n$, so that there are no d-dimensional simplices), then return the simplicial complex with a single $(n + 1)$-dimensional simplex:

```
sage: simplicial_complexes.RandomComplex(6, 12)
The 5-simplex
```

sage.homology.examples. **RandomTwoSphere** (*n*)

Return a random triangulation of the 2-dimensional sphere with n vertices.

INPUT:

n – an integer

OUTPUT:

A random triangulation of the sphere chosen uniformly among the *rooted* triangulations on n vertices. Because some triangulations have nontrivial automorphism groups, this may not be equal to the uniform distribution among unrooted triangulations.

ALGORITHM:

The algorithm is taken from [PS2006], section 2.1.

Starting from a planar tree (represented by its contour as a sequence of vertices), one first performs local closures, until no one is possible. A local closure amounts to replace in the cyclic contour word a sequence `in1,in2,in3,lf,in3` by `in1,in3` . After all local closures are done, one has reached the partial closure, as in [PS2006], figure 5 (a).

Then one has to perform complete closure by adding two more vertices, in order to reach the situation of [PS2006], figure 5 (b). For this, it is necessary to find inside the final contour one of the two subsequences `lf,in,lf` .

At every step of the algorithm, newly created triangles are added in a simplicial complex.

This algorithm is implemented in `RandomTriangulation()` , which creates an embedded graph. The triangles of the simplicial complex are recovered from this embedded graph.

EXAMPLES:

```
sage: G = simplicial_complexes.RandomTwoSphere(6); G
Simplicial complex with vertex set (0, 1, 2, 3, 'a', 'b')
and 8 facets
sage: G.homology()
{0: 0, 1: 0, 2: Z}
sage: G.is_pure()
True
sage: fg = G.flip_graph(); fg
Graph on 8 vertices
sage: fg.is_planar() and fg.is_regular(3)
True
```

sage.homology.examples. **RealProjectivePlane** ( )

A minimal triangulation of the real projective plane.

EXAMPLES:

```
sage: P = simplicial_complexes.RealProjectivePlane()
sage: Q = simplicial_complexes.ProjectivePlane()
sage: P == Q
True
sage: P.cohomology(1)
0
sage: P.cohomology(2)
C2
sage: P.cohomology(1, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
sage: P.cohomology(2, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
```

sage.homology.examples. **RealProjectiveSpace** (*n*)

A triangulation of $\mathbf{R}P^n$ for any $n \geq 0$.

INPUT:

- n – integer, the dimension of the real projective space to construct

The first few cases are pretty trivial:

- $\mathbf{R}P^0$ is a point.
- $\mathbf{R}P^1$ is a circle, triangulated as the boundary of a single 2-simplex.
- $\mathbf{R}P^2$ is the real projective plane, here given its minimal triangulation with 6 vertices, 15 edges, and 10 triangles.
- $\mathbf{R}P^3$: any triangulation has at least 11 vertices by a result of Walkup [Wal1970]; this function returns a triangulation with 11 vertices, as given by Lutz [Lut2005].
- $\mathbf{R}P^4$: any triangulation has at least 16 vertices by a result of Walkup; this function returns a triangulation with 16 vertices as given by Lutz; see also Datta [Dat2007], Example 3.12.
- $\mathbf{R}P^n$: Lutz has found a triangulation of $\mathbf{R}P^5$ with 24 vertices, but it does not seem to have been published. Kühnel [Kuh1987] has described a triangulation of $\mathbf{R}P^n$, in general, with $2^{n+1}-1$ vertices; see also Datta, Example 3.21. This triangulation is presumably not minimal, but it seems to be the best in the published literature as of this writing. So this function returns it when $n > 4$.

ALGORITHM: For $n < 4$, these are constructed explicitly by listing the facets. For $n = 4$, this is constructed by specifying 16 vertices, two facets, and a certain subgroup G of the symmetric group S_{16}. Then the set of all facets is the G-orbit of the two given facets. This is implemented here by explicitly listing all of the facets; the facets can be computed by the function *facets_for_RP4()* , but running the function takes a few seconds.

For $n > 4$, the construction is as follows: let S denote the simplicial complex structure on the n-sphere given by the first barycentric subdivision of the boundary of an $(n+1)$-simplex. This has a simplicial antipodal action: if V denotes the vertices in the boundary of the simplex, then the vertices in its barycentric subdivision S correspond to nonempty proper subsets U of V, and the antipodal action sends any subset U to its complement. One can show that modding out by this action results in a triangulation for $\mathbf{R}P^n$. To find the facets in this triangulation, find the facets in S. These are indentified in pairs to form $\mathbf{R}P^n$, so choose a representative from each pair: for each facet in S, replace any vertex in S containing 0 with its complement.

Of course these complexes increase in size pretty quickly as n increases.

EXAMPLES:

```
sage: P3 = simplicial_complexes.RealProjectiveSpace(3)
sage: P3.f_vector()
[1, 11, 51, 80, 40]
sage: P3.homology()
{0: 0, 1: C2, 2: 0, 3: Z}
sage: P4 = simplicial_complexes.RealProjectiveSpace(4)
sage: P4.f_vector()
[1, 16, 120, 330, 375, 150]
sage: P4.homology() # long time
{0: 0, 1: C2, 2: 0, 3: C2, 4: 0}
sage: P5 = simplicial_complexes.RealProjectiveSpace(5)  # long time (44s on sage.
↪math, 2012)
sage: P5.f_vector()  # long time
[1, 63, 903, 4200, 8400, 7560, 2520]
```

The following computation can take a long time – over half an hour – with Sage's default computation of homology groups, but if you have CHomP installed, Sage will use that and the computation should only take a second or two. (You can download CHomP from http://chomp.rutgers.edu/, or you can install it as a Sage package using `sage -i chomp`).

```
sage: P5.homology()  # long time # optional - CHomP
{0: 0, 1: C2, 2: 0, 3: C2, 4: 0, 5: Z}
sage: simplicial_complexes.RealProjectiveSpace(2).dimension()
2
sage: P3.dimension()
3
sage: P4.dimension() # long time
4
sage: P5.dimension() # long time
5
```

sage.homology.examples. **RudinBall** ( )

Return the non-shellable ball constructed by Rudin.

This complex is a non-shellable triangulation of the 3-ball with 14 vertices and 41 facets, constructed by Rudin in [Rud1958].

EXAMPLES:

```
sage: R = simplicial_complexes.RudinBall(); R
Rudin ball
sage: R.f_vector()
[1, 14, 66, 94, 41]
sage: R.homology()
{0: 0, 1: 0, 2: 0, 3: 0}
sage: R.is_cohen_macaulay()
True
```

sage.homology.examples. **ShiftedComplex** (*generators*)

Return the smallest shifted simplicial complex containing `generators` as faces.

Let V be a set of vertices equipped with a total order. The 'componentwise partial ordering' on k-subsets of V is defined as follows: if $A = \{a_1 < \cdots < a_k\}$ and $B = \{b_1 < \cdots < b_k\}$, then $A \leq_C B$ iff $a_i \leq b_i$ for all i. A simplicial complex X on vertex set $[n]$ is *shifted* if its faces form an order ideal under the componentwise partial ordering, i.e., if $B \in X$ and $A \leq_C B$ then $A \in X$. Shifted complexes of dimension 1 are also known as threshold graphs.

Note: This method assumes that V consists of positive integers with the natural ordering.

INPUT:

- `generators` – a list of generators of the order ideal, which may be lists, tuples or simplices

EXAMPLES:

```
sage: X = simplicial_complexes.ShiftedComplex([ Simplex([1,6]), (2,4), [8] ])
sage: X.facets()
{(2, 4), (7,), (1, 2), (1, 5), (1, 4), (8,), (2, 3), (1, 6), (1, 3)}
sage: X = simplicial_complexes.ShiftedComplex([ [2,3,5] ])
sage: X.facets()
{(1, 3, 4), (1, 3, 5), (2, 3, 5), (1, 2, 3), (2, 3, 4), (1, 2, 5), (1, 2, 4)}
sage: X = simplicial_complexes.ShiftedComplex([ [1,3,5], [2,6] ])
sage: X.facets()
{(1, 3, 4), (1, 3, 5), (1, 6), (2, 6), (1, 2, 3), (1, 2, 5), (1, 2, 4)}
```

sage.homology.examples. **Simplex** (*n*)

An n-dimensional simplex, as a simplicial complex.

INPUT:

- `n` – a non-negative integer

OUTPUT: the simplicial complex consisting of the n-simplex on vertices $(0, 1, ..., n)$ and all of its faces.

EXAMPLES:

```
sage: simplicial_complexes.Simplex(3)
The 3-simplex
sage: simplicial_complexes.Simplex(5).euler_characteristic()
1
```

sage.homology.examples. **Sphere** (*n*)

A minimal triangulation of the n-dimensional sphere.

INPUT:

- `n` – positive integer

EXAMPLES:

```
sage: simplicial_complexes.Sphere(2)
Minimal triangulation of the 2-sphere
sage: simplicial_complexes.Sphere(5).homology()
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: Z}
sage: [simplicial_complexes.Sphere(n).euler_characteristic() for n in range(6)]
[2, 0, 2, 0, 2, 0]
sage: [simplicial_complexes.Sphere(n).f_vector() for n in range(6)]
[[1, 2],
 [1, 3, 3],
 [1, 4, 6, 4],
 [1, 5, 10, 10, 5],
 [1, 6, 15, 20, 15, 6],
 [1, 7, 21, 35, 35, 21, 7]]
```

sage.homology.examples. **SumComplex** (*n*, *A*)

The sum complexes of Linial, Meshulam, and Rosenthal [LMR2010].

If $k+1$ is the cardinality of A, then this returns a k-dimensional simplicial complex X_A with vertices $\mathbf{Z}/(n)$, and facets given by all $k+1$-tuples $(x_0, x_1, ..., x_k)$ such that the sum $\sum x_i$ is in A. See the paper by Linial, Meshulam, and Rosenthal [LMR2010], in which they prove various results about these complexes; for example, if n is prime, then X_A is rationally acyclic, and if in addition A forms an arithmetic progression in $\mathbf{Z}/(n)$, then X_A is $\mathbf{Z}$-acyclic. Throughout their paper, they assume that n and k are relatively prime, but the construction makes sense in general.

In addition to the results from the cited paper, these complexes can have large torsion, given the number of vertices; for example, if $n = 10$, and $A = \{0, 1, 2, 3, 6\}$, then $H_3(X_A)$ is cyclic of order 2728, and there is a 4-dimensional complex on 13 vertices with H_3 having a cyclic summand of order

$$706565607945 = 3 \cdot 5 \cdot 53 \cdot 79 \cdot 131 \cdot 157 \cdot 547.$$

See the examples.

INPUT:

- `n` – a positive integer
- `A` – a subset of $\mathbf{Z}/(n)$

EXAMPLES:

```
sage: S = simplicial_complexes.SumComplex(10, [0,1,2,3,6]); S
Sum complex on vertices Z/10Z associated to {0, 1, 2, 3, 6}
sage: S.homology()
{0: 0, 1: 0, 2: 0, 3: C2728, 4: 0}
sage: factor(2728)
2^3 * 11 * 31

sage: S = simplicial_complexes.SumComplex(11, [0, 1, 3]); S
Sum complex on vertices Z/11Z associated to {0, 1, 3}
sage: S.homology(1)
C23
sage: S = simplicial_complexes.SumComplex(11, [0,1,2,3,4,7]); S
Sum complex on vertices Z/11Z associated to {0, 1, 2, 3, 4, 7}
sage: S.homology(algorithm='no_chomp') # long time
{0: 0, 1: 0, 2: 0, 3: 0, 4: C645679, 5: 0}
sage: factor(645679)
23 * 67 * 419

sage: S = simplicial_complexes.SumComplex(13, [0, 1, 3]); S
Sum complex on vertices Z/13Z associated to {0, 1, 3}
sage: S.homology(1)
C159
sage: factor(159)
3 * 53
sage: S = simplicial_complexes.SumComplex(13, [0,1,2,5]); S
Sum complex on vertices Z/13Z associated to {0, 1, 2, 5}
sage: S.homology(algorithm='no_chomp') # long time
{0: 0, 1: 0, 2: C146989209, 3: 0}
sage: factor(1648910295)
3^2 * 5 * 53 * 521 * 1327
sage: S = simplicial_complexes.SumComplex(13, [0,1,2,3,5]); S
Sum complex on vertices Z/13Z associated to {0, 1, 2, 3, 5}
sage: S.homology(algorithm='no_chomp') # long time
{0: 0, 1: 0, 2: 0, 3: C3 x C237 x C706565607945, 4: 0}
sage: factor(706565607945)
3 * 5 * 53 * 79 * 131 * 157 * 547
```

```
sage: S = simplicial_complexes.SumComplex(17, [0, 1, 4]); S
Sum complex on vertices Z/17Z associated to {0, 1, 4}
sage: S.homology(1, algorithm='no_chomp')
C140183
sage: factor(140183)
103 * 1361
sage: S = simplicial_complexes.SumComplex(19, [0, 1, 4]); S
Sum complex on vertices Z/19Z associated to {0, 1, 4}
sage: S.homology(1,algorithm='no_chomp')
C5670599
sage: factor(5670599)
11 * 191 * 2699
sage: S = simplicial_complexes.SumComplex(31, [0, 1, 4]); S
Sum complex on vertices Z/31Z associated to {0, 1, 4}
sage: S.homology(1,algorithm='no_chomp') # long time
C5 x C5 x C5 x C5 x C26951480558170926865
sage: factor(26951480558170926865)
5 * 311 * 683 * 1117 * 11657 * 1948909
```

sage.homology.examples. **SurfaceOfGenus** (*g*, *orientable=True*)

A surface of genus g.

INPUT:

- `g` – a non-negative integer. The desired genus
- `orientable` – boolean (optional, default `True`). If `True` , return an orientable surface, and if `False` , return a non-orientable surface.

In the orientable case, return a sphere if g is zero, and otherwise return a g-fold connected sum of a torus with itself.

In the non-orientable case, raise an error if g is zero. If g is positive, return a g-fold connected sum of a real projective plane with itself.

EXAMPLES:

```
sage: simplicial_complexes.SurfaceOfGenus(2)
Triangulation of an orientable surface of genus 2
sage: simplicial_complexes.SurfaceOfGenus(1, orientable=False)
Triangulation of a non-orientable surface of genus 1
```

sage.homology.examples. **Torus** ( )

A minimal triangulation of the torus.

This is a simplicial complex with 7 vertices, 21 edges and 14 faces. It is the unique triangulation of the torus with 7 vertices, and has been found by Möbius in 1861.

This is also the combinatorial structure of the Császár polyhedron (see Wikipedia article Császár_polyhedron).

EXAMPLES:

```
sage: T = simplicial_complexes.Torus(); T.homology(1)
Z x Z
sage: T.f_vector()
[1, 7, 21, 14]
```

TESTS:

```
sage: T.flip_graph().is_isomorphic(graphs.HeawoodGraph())
True
```

REFERENCES:

- [Lut2002]

class sage.homology.examples. **UniqueSimplicialComplex** (*maximal_faces=None*, *name=None*, ***kwds*)

Bases: *sage.homology.simplicial_complex.SimplicialComplex* , sage.structure.unique_representation.UniqueRepresentation

This combines *SimplicialComplex* and UniqueRepresentation . It is intended to be used to make standard examples of simplicial complexes unique. See trac ticket #13566.

INPUTS:

- the inputs are the same as for a *SimplicialComplex* , with one addition and two exceptions. The exceptions are that is_mutable and is_immutable are ignored: all instances of this class are immutable. The addition:
- name – string (optional), the string representation for this complex.

EXAMPLES:

```
sage: from sage.homology.examples import UniqueSimplicialComplex
sage: SimplicialComplex([[0,1]]) is SimplicialComplex([[0,1]])
False
sage: UniqueSimplicialComplex([[0,1]]) is UniqueSimplicialComplex([[0,1]])
True
sage: UniqueSimplicialComplex([[0,1]])
Simplicial complex with vertex set (0, 1) and facets {(0, 1)}
sage: UniqueSimplicialComplex([[0,1]], name='The 1-simplex')
The 1-simplex
```

sage.homology.examples. **ZieglerBall** ( )

Return the non-shellable ball constructed by Ziegler.

This complex is a non-shellable triangulation of the 3-ball with 10 vertices and 21 facets, constructed by Ziegler in [Zie1998] and the smallest such complex known.

EXAMPLES:

```
sage: Z = simplicial_complexes.ZieglerBall(); Z
Ziegler ball
sage: Z.f_vector()
[1, 10, 38, 50, 21]
sage: Z.homology()
{0: 0, 1: 0, 2: 0, 3: 0}
sage: Z.is_cohen_macaulay()
True
```

sage.homology.examples. **facets_for_K3** ( )

Returns the facets for a minimal triangulation of the K3 surface.

This is a pure simplicial complex of dimension 4 with 16 vertices and 288 facets. The facets are obtained by constructing a few facets and a permutation group G, and then computing the G-orbit of those facets.

See Casella and Kühnel in [CK2001] and Spreer and Kühnel [SK2011]; the construction here uses the labeling from Spreer and Kühnel.

EXAMPLES:

```
sage: from sage.homology.examples import facets_for_K3
sage: A = facets_for_K3()    # long time (a few seconds)
sage: SimplicialComplex(A) == simplicial_complexes.K3Surface()  # long time
True
```

sage.homology.examples. **facets_for_RP4** ( )

Return the list of facets for a minimal triangulation of 4-dimensional real projective space.

We use vertices numbered 1 through 16, define two facets, and define a certain subgroup G of the symmetric group S_{16}. Then the set of all facets is the G-orbit of the two given facets.

See the description in Example 3.12 in Datta [Dat2007].

EXAMPLES:

```
sage: from sage.homology.examples import facets_for_RP4
sage: A = facets_for_RP4()    # long time (1 or 2 seconds)
sage: SimplicialComplex(A) == simplicial_complexes.RealProjectiveSpace(4) # long␣
↪time
True
```

sage.homology.examples. **matching** (*A*, *B*)

List of maximal matchings between the sets `A` and `B` .

A matching is a set of pairs $(a, b) \in A \times B$ where each a and b appears in at most one pair. A maximal matching is one which is maximal with respect to inclusion of subsets of $A \times B$.

INPUT:

- `A` , `B` – list, tuple, or indeed anything which can be converted to a set.

EXAMPLES:

```
sage: from sage.homology.examples import matching
sage: matching([1,2], [3,4])
[{(1, 3), (2, 4)}, {(1, 4), (2, 3)}]
sage: matching([0,2], [0])
[{(0, 0)}, {(2, 0)}]
```

CHAPTER

TEN

FINITE DELTA-COMPLEXES

AUTHORS:

- John H. Palmieri (2009-08)

This module implements the basic structure of finite Δ-complexes. For full mathematical details, see Hatcher [Hat2002], especially Section 2.1 and the Appendix on "Simplicial CW Structures". As Hatcher points out, Δ-complexes were first introduced by Eilenberg and Zilber [EZ1950], although they called them "semi-simplicial complexes".

A Δ-complex is a generalization of a *simplicial complex* ; a Δ-complex X consists of sets X_n for each non-negative integer n, the elements of which are called *n-simplices*, along with *face maps* between these sets of simplices: for each n and for all $0 \leq i \leq n$, there are functions d_i from X_n to X_{n-1}, with $d_i(s)$ equal to the i-th face of s for each simplex $s \in X_n$. These maps must satisfy the *simplicial identity*

$$d_i d_j = d_{j-1} d_i \text{ for all } i < j.$$

Given a Δ-complex, it has a *geometric realization*: a topological space built by taking one topological n-simplex for each element of X_n, and gluing them together as determined by the face maps.

Δ-complexes are an alternative to simplicial complexes. Every simplicial complex is automatically a Δ-complex; in the other direction, though, it seems in practice that one can often construct Δ-complex representations for spaces with many fewer simplices than in a simplicial complex representation. For example, the minimal triangulation of a torus as a simplicial complex contains 14 triangles, 21 edges, and 7 vertices, while there is a Δ-complex representation of a torus using only 2 triangles, 3 edges, and 1 vertex.

Note: This class derives from *GenericCellComplex* , and so inherits its methods. Some of those methods are not listed here; see the *Generic Cell Complex* page instead.

REFERENCES:

- [Hat2002]
- [EZ1950]

class sage.homology.delta_complex. **DeltaComplex** (*data=None*, *check_validity=True*)
Bases: *sage.homology.cell_complex.GenericCellComplex*

Define a Δ-complex.

Parameters

- **data** – see below for a description of the options

- **check_validity** (*boolean; optional, default True*) – If True, check that the simplicial identities hold.

Returns a Δ-complex

Use `data` to define a Δ-complex. It may be in any of three forms:

- `data` may be a dictionary indexed by simplices. The value associated to a d-simplex S can be any of:
  - a list or tuple of (d-1)-simplices, where the ith entry is the ith face of S, given as a simplex,
  - another d-simplex T, in which case the ith face of S is declared to be the same as the ith face of T: S and T are glued along their entire boundary,
  - None or True or False or anything other than the previous two options, in which case the faces are just the ordinary faces of S.

For example, consider the following:

```
sage: n = 5
sage: S5 = DeltaComplex({Simplex(n):True, Simplex(range(1,n+2)): Simplex(n)})
sage: S5
Delta complex with 6 vertices and 65 simplices
```

The first entry in dictionary forming the argument to `DeltaComplex` says that there is an n-dimensional simplex with its ordinary boundary. The second entry says that there is another simplex whose boundary is glued to that of the first one. The resulting Δ-complex is, of course, homeomorphic to an n-sphere, or actually a 5-sphere, since we defined n to be 5. (Note that the second simplex here can be any n-dimensional simplex, as long as it is distinct from `Simplex(n)` .)

Let's compute its homology, and also compare it to the simplicial version:

```
sage: S5.homology()
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: Z}
sage: S5.f_vector()  # number of simplices in each dimension
[1, 6, 15, 20, 15, 6, 2]
sage: simplicial_complexes.Sphere(5).f_vector()
[1, 7, 21, 35, 35, 21, 7]
```

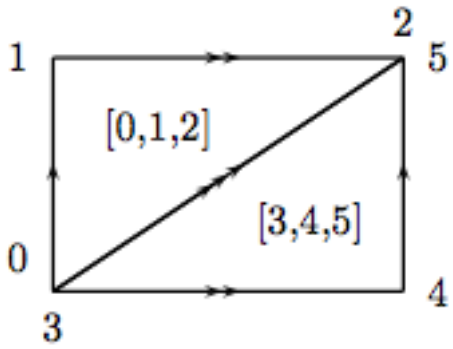
Both contain a single (-1)-simplex, the empty simplex; other than that, the Δ-complex version contains fewer simplices than the simplicial one in each dimension.

To construct a torus, use:

```
sage: torus_dict = {Simplex([0,1,2]): True,
....:               Simplex([3,4,5]): (Simplex([0,1]), Simplex([0,2]),␣
↪Simplex([1,2])),
....:               Simplex([0,1]): (Simplex(0), Simplex(0)),
....:               Simplex([0,2]): (Simplex(0), Simplex(0)),
....:               Simplex([1,2]): (Simplex(0), Simplex(0)),
....:               Simplex(0): ()}
sage: T = DeltaComplex(torus_dict); T
Delta complex with 1 vertex and 7 simplices
sage: T.cohomology(base_ring=QQ)
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

This Δ-complex consists of two triangles (given by `Simplex([0,1,2])` and `Simplex([3,4,5])`); the boundary of the first is just its usual boundary: the 0th face is obtained by omitting the lowest numbered vertex, etc., and so the boundary consists of the edges `[1,2]` , `[0,2]` , and `[0,1]` , in that

order. The boundary of the second is, on the one hand, computed the same way: the nth face is obtained by omitting the nth vertex. On the other hand, the boundary is explicitly declared to be edges `[0,1]`, `[0,2]`, and `[1,2]`, in that order. This glues the second triangle to the first in the prescribed way. The three edges each start and end at the single vertex, `Simplex(0)`.



- `data` may be nested lists or tuples. The nth entry in the list is a list of the n-simplices in the complex, and each n-simplex is encoded as a list, the ith entry of which is its ith face. Each face is represented by an integer, giving its index in the list of (n-1)-faces. For example, consider this:

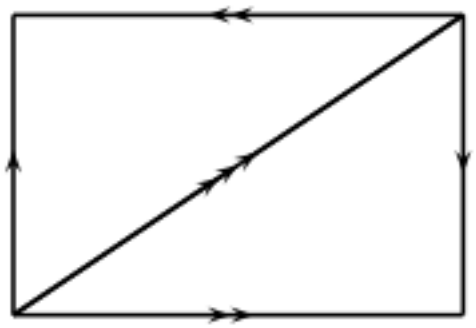
```
sage: P = DeltaComplex( [ [(), ()],  [(1,0), (1,0), (0,0)],
....:                     [(1,0,2), (0, 1, 2)] ])
```

The 0th entry in the list is `[(),()]`: there are two 0-simplices, and their boundaries are empty.

The 1st entry in the list is `[(1,0),(1,0),(0,0)]`: there are three 1-simplices. Two of them have boundary `(1,0)`, which means that their 0th face is vertex 1 (in the list of vertices), and their 1st face is vertex 0. The other edge has boundary `(0,0)`, so it starts and ends at vertex 0.

The 2nd entry in the list is `[(1,0,2),(0,1,2)]`: there are two 2-simplices. The first 2-simplex has boundary `(1,0,2)`, meaning that its 0th face is edge 1 (in the list above), its 1st face is edge 0, and its 2nd face is edge 2; similarly for the 2nd 2-simplex.

If one draws two triangles and identifies them according to this description, the result is the real projective plane.



```
sage: P.homology(1)
C2
sage: P.cohomology(2)
C2
```

Closely related to this form for `data` is `X.cells()` for a Δ-complex `X`: this is a dictionary, indexed by dimension `d`, whose `d`-th entry is a list of the `d`-simplices, as a list:

```
sage: P.cells()
{-1: ((),),
 0: ((), ()),
 1: ((1, 0), (1, 0), (0, 0)),
 2: ((1, 0, 2), (0, 1, 2))}
```

- `data` may be a dictionary indexed by integers. For each integer n, the entry with key n is the list of n-simplices: this is the same format as is output by the *`cells()`* method.

```
sage: P = DeltaComplex( [ [(), ()],  [(1,0), (1,0), (0,0)],
....:                     [(1,0,2), (0, 1, 2)] ])
sage: cells_dict = P.cells()
sage: cells_dict
{-1: ((),),
 0: ((), ()),
 1: ((1, 0), (1, 0), (0, 0)),
 2: ((1, 0, 2), (0, 1, 2))}
sage: DeltaComplex(cells_dict)
Delta complex with 2 vertices and 8 simplices
sage: P == DeltaComplex(cells_dict)
True
```

Since Δ-complexes are generalizations of simplicial complexes, any simplicial complex may be viewed as a Δ-complex:

```
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: RP2_delta = RP2.delta_complex()
sage: RP2.f_vector()
[1, 6, 15, 10]
sage: RP2_delta.f_vector()
[1, 6, 15, 10]
```

Finally, Δ-complex constructions for several familiar spaces are available as follows:

```
sage: delta_complexes.Sphere(4)  # the 4-sphere
Delta complex with 5 vertices and 33 simplices
sage: delta_complexes.KleinBottle()
Delta complex with 1 vertex and 7 simplices
sage: delta_complexes.RealProjectivePlane()
Delta complex with 2 vertices and 8 simplices
```

Type `delta_complexes.` and then hit the TAB key to get the full list.

alexander_whitney (*cell*, *dim_left*)

Subdivide `cell` in this Δ-complex into a pair of simplices.

For an abstract simplex with vertices v_0, v_1, ..., v_n, then subdivide it into simplices $(v_0, v_1, ..., v_{dim_left})$ and $(v_{dim_left}, v_{dim_left+1}, ..., v_n)$. In a Δ-complex, instead take iterated faces: take top faces to get the left factor, take bottom faces to get the right factor.

INPUT:

- `cell` – a simplex in this complex, given as a pair `(idx,tuple)` , where `idx` is its index in the list of cells in the given dimension, and `tuple` is the tuple of its faces
- `dim_left` – integer between 0 and one more than the dimension of this simplex

OUTPUT: a list containing just the triple `(1,left,right)` , where `left` and `right` are the two cells described above, each given as pairs `(idx,tuple)` .

EXAMPLES:

```
sage: X = delta_complexes.Torus()
sage: X.n_cells(2)
[(1, 2, 0), (0, 2, 1)]
sage: X.alexander_whitney((0, (1, 2, 0)), 1)
[(1, (0, (0, 0)), (1, (0, 0)))]
sage: X.alexander_whitney((0, (1, 2, 0)), 0)
[(1, (0, ()), (0, (1, 2, 0)))]
sage: X.alexander_whitney((1, (0, 2, 1)), 2)
[(1, (1, (0, 2, 1)), (0, ()))]
```

algebraic_topological_model (*base_ring=None*)

Algebraic topological model for this Δ-complex with coefficients in `base_ring` .

The term "algebraic topological model" is defined by Pilarczyk and Réal [PR2015].

INPUT:

- `base_ring` - coefficient ring (optional, default `QQ`). Must be a field.

Denote by C the chain complex associated to this Δ-complex. The algebraic topological model is a chain complex M with zero differential, with the same homology as C, along with chain maps $\pi : C \to M$ and $\iota : M \to C$ satisfying $\iota\pi = 1_M$ and $\pi\iota$ chain homotopic to 1_C. The chain homotopy ϕ must satisfy

- $\phi\phi = 0$,
- $\pi\phi = 0$,
- $\phi\iota = 0$.

Such a chain homotopy is called a *chain contraction*.

OUTPUT: a pair consisting of

- chain contraction `phi` associated to C, M, π, and ι
- the chain complex M

Note that from the chain contraction `phi` , one can recover the chain maps π and ι via `phi.pi()` and `phi.iota()` . Then one can recover C and M from, for example, `phi.pi().domain()` and `phi.pi().codomain()` , respectively.

EXAMPLES:

```
sage: RP2 = delta_complexes.RealProjectivePlane()
sage: phi, M = RP2.algebraic_topological_model(GF(2))
sage: M.homology()
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: T = delta_complexes.Torus()
sage: phi, M = T.algebraic_topological_model(QQ)
sage: M.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

barycentric_subdivision ( )

Not implemented.

EXAMPLES:

```
sage: K = delta_complexes.KleinBottle()
sage: K.barycentric_subdivision()
Traceback (most recent call last):
...
NotImplementedError: Barycentric subdivisions are not implemented for Delta␣
↪complexes.
```

cells (*subcomplex=None*)

The cells of this Δ-complex.

Parameters **subcomplex** (*optional, default None*) – a subcomplex of this complex

The cells of this Δ-complex, in the form of a dictionary: the keys are integers, representing dimension, and the value associated to an integer d is the list of d-cells. Each d-cell is further represented by a list, the ith entry of which gives the index of its ith face in the list of (d-1)-cells.

If the optional argument `subcomplex` is present, then "return only the faces which are *not* in the subcomplex". To preserve the indexing, which is necessary to compute the relative chain complex, this actually replaces the faces in `subcomplex` with `None` .

EXAMPLES:

```
sage: S2 = delta_complexes.Sphere(2)
sage: S2.cells()
{-1: ((),),
 0: ((), (), ()),
 1: ((0, 1), (0, 2), (1, 2)),
 2: ((0, 1, 2), (0, 1, 2))}
sage: A = S2.subcomplex({1: [0,2]}) # one edge
sage: S2.cells(subcomplex=A)
{-1: (None,),
 0: (None, None, None),
 1: (None, (0, 2), None),
 2: ((0, 1, 2), (0, 1, 2))}
```

chain_complex (*subcomplex=None*, *augmented=False*, *verbose=False*, *check=False*, *dimensions=None*, *base_ring=Integer Ring*, *cochain=False*)

The chain complex associated to this Δ-complex.

Parameters

- **dimensions** – if None, compute the chain complex in all dimensions. If a list or tuple of integers, compute the chain complex in those dimensions, setting the chain groups in all other dimensions to zero. NOT IMPLEMENTED YET: this function always returns the entire chain complex
- **base_ring** (*optional, default ZZ*) – commutative ring
- **subcomplex** (*optional, default empty*) – a subcomplex of this simplicial complex. Compute the chain complex relative to this subcomplex.
- **augmented** (*boolean; optional, default False*) – If True, return the augmented chain complex (that is, include a class in dimension -1 corresponding to the empty cell). This is ignored if `dimensions` is specified or if `subcomplex` is nonempty.
- **cochain** (*boolean; optional, default False*) – If True, return the cochain complex (that is, the dual of the chain complex).
- **verbose** (*boolean; optional, default False*) – If True, print some messages as the chain complex is computed.

- **check** (*boolean; optional, default False*) – If True, make sure that the chain complex is actually a chain complex: the differentials are composable and their product is zero.

Note: If subcomplex is nonempty, then the argument `augmented` has no effect: the chain complex relative to a nonempty subcomplex is zero in dimension -1.

EXAMPLES:

```
sage: circle = delta_complexes.Sphere(1)
sage: circle.chain_complex()
Chain complex with at most 2 nonzero terms over Integer Ring
sage: circle.chain_complex()._latex_()
'\\Bold{Z}^{1} \\xrightarrow{d_{1}} \\Bold{Z}^{1}'
sage: circle.chain_complex(base_ring=QQ, augmented=True)
Chain complex with at most 3 nonzero terms over Rational Field
sage: circle.homology(dim=1)
Z
sage: circle.cohomology(dim=1)
Z
sage: T = delta_complexes.Torus()
sage: T.chain_complex(subcomplex=T)
Trivial chain complex over Integer Ring
sage: T.homology(subcomplex=T, algorithm='no_chomp')
{0: 0, 1: 0, 2: 0}
sage: A = T.subcomplex({2: [1]})  # one of the two triangles forming T
sage: T.chain_complex(subcomplex=A)
Chain complex with at most 1 nonzero terms over Integer Ring
sage: T.homology(subcomplex=A)
{0: 0, 1: 0, 2: Z}
```

cone ( )

The cone on this Δ-complex.

The cone is the complex formed by adding a new vertex C and simplices of the form $[C, v_0, ..., v_k]$ for every simplex $[v_0, ..., v_k]$ in the original complex. That is, the cone is the join of the original complex with a one-point complex.

EXAMPLES:

```
sage: K = delta_complexes.KleinBottle()
sage: K.cone()
Delta complex with 2 vertices and 14 simplices
sage: K.cone().homology()
{0: 0, 1: 0, 2: 0, 3: 0}
```

connected_sum (*other*)

Return the connected sum of self with other.

Parameters **other** – another Δ-complex

Returns the connected sum `self # other`

Warning: This does not check that self and other are manifolds. It doesn't even check that their facets all have the same dimension. It just chooses top-dimensional simplices from each complex, checks that they have the same dimension, removes them, and glues the remaining pieces together. Since a (more

or less) random facet is chosen from each complex, this method may return random results if applied to non-manifolds, depending on which facet is chosen.

ALGORITHM:

Pick a top-dimensional simplex from each complex. Check to see if there are any identifications on either simplex, using the `_is_glued()` method. If there are no identifications, remove the simplices and glue the remaining parts of complexes along their boundary. If there are identifications on a simplex, subdivide it repeatedly (using *`elementary_subdivision()`*) until some piece has no identifications.

EXAMPLES:

```
sage: T = delta_complexes.Torus()
sage: S2 = delta_complexes.Sphere(2)
sage: T.connected_sum(S2).cohomology() == T.cohomology()
True
sage: RP2 = delta_complexes.RealProjectivePlane()
sage: T.connected_sum(RP2).homology(1)
Z x Z x C2
sage: T.connected_sum(RP2).homology(2)
0
sage: RP2.connected_sum(RP2).connected_sum(RP2).homology(1)
Z x Z x C2
```

`disjoint_union` (*right*)

The disjoint union of this Δ-complex with another one.

Parameters **`right`** – the other Δ-complex (the right-hand factor)

EXAMPLES:

```
sage: S1 = delta_complexes.Sphere(1)
sage: S2 = delta_complexes.Sphere(2)
sage: S1.disjoint_union(S2).homology()
{0: Z, 1: Z, 2: Z}
```

`elementary_subdivision` (*idx=-1*)

Perform an "elementary subdivision" on a top-dimensional simplex in this Δ-complex. If the optional argument `idx` is present, it specifies the index (in the list of top-dimensional simplices) of the simplex to subdivide. If not present, subdivide the last entry in this list.

Parameters **`idx`** (*`integer; optional, default -1`*) – index specifying which simplex to subdivide

Returns Δ-complex with one simplex subdivided.

Elementary subdivision of a simplex means replacing that simplex with the cone on its boundary. That is, given a Δ-complex containing an d-simplex S with vertices v_0, ..., v_d, form a new Δ-complex by

- removing S
- adding a vertex w (thought of as being in the interior of S)
- adding all simplices with vertices v_{i_0}, ..., v_{i_k}, w, preserving any identifications present along the boundary of S

The algorithm for achieving this uses `_epi_from_standard_simplex()` to keep track of simplices (with multiplicity) and what their faces are: this method defines a surjection π from the standard d-simplex to S. So first remove S and add a new vertex w, say at the end of the old list of vertices. Then for each

vertex v in the standard d-simplex, add an edge from $\pi(v)$ to w; for each edge (v_0, v_1) in the standard d-simplex, add a triangle $(\pi(v_0), \pi(v_1), w)$, etc.

Note that given an n-simplex $(v_0, v_1, ..., v_n)$ in the standard d-simplex, the faces of the new $(n+1)$-simplex are given by removing vertices, one at a time, from $(\pi(v_0), ..., \pi(v_n), w)$. These are either the image of the old n-simplex (if w is removed) or the various new n-simplices added in the previous dimension. So keep track of what's added in dimension n for use in computing the faces in dimension $n + 1$.

In contrast with barycentric subdivision, note that only the interior of S has been changed; this allows for subdivision of a single top-dimensional simplex without subdividing every simplex in the complex.

The term "elementary subdivison" is taken from p. 112 in John M. Lee's book [Lee2011].

EXAMPLES:

```
sage: T = delta_complexes.Torus()
sage: T.n_cells(2)
[(1, 2, 0), (0, 2, 1)]
sage: T.elementary_subdivision(0)  # subdivide first triangle
Delta complex with 2 vertices and 13 simplices
sage: X = T.elementary_subdivision(); X  # subdivide last triangle
Delta complex with 2 vertices and 13 simplices
sage: X.elementary_subdivision()
Delta complex with 3 vertices and 19 simplices
sage: X.homology() == T.homology()
True
```

face_poset ( )

The face poset of this Δ-complex, the poset of nonempty cells, ordered by inclusion.

EXAMPLES:

```
sage: T = delta_complexes.Torus()
sage: T.face_poset()
Finite poset containing 6 elements
```

graph ( )

The 1-skeleton of this Δ-complex as a graph.

EXAMPLES:

```
sage: T = delta_complexes.Torus()
sage: T.graph()
Looped multi-graph on 1 vertex
sage: S = delta_complexes.Sphere(2)
sage: S.graph()
Graph on 3 vertices
sage: delta_complexes.Simplex(4).graph() == graphs.CompleteGraph(5)
True
```

join (*other*)

The join of this Δ-complex with another one.

Parameters **other** – another Δ-complex (the right-hand factor)

Returns the join `self * other`

The join of two Δ-complexes S and T is the Δ-complex $S * T$ with simplices of the form $[v_0, ..., v_k, w_0, ..., w_n]$ for all simplices $[v_0, ..., v_k]$ in S and $[w_0, ..., w_n]$ in T. The faces are computed accordingly: the ith face of such a simplex is either $(d_i S) * T$ if $i \leq k$, or $S * (d_{i-k-1} T)$ if $i > k$.

EXAMPLES:

```
sage: T = delta_complexes.Torus()
sage: S0 = delta_complexes.Sphere(0)
sage: T.join(S0)  # the suspension of T
Delta complex with 3 vertices and 21 simplices
```

Compare to simplicial complexes:

```
sage: K = delta_complexes.KleinBottle()
sage: T_simp = simplicial_complexes.Torus()
sage: K_simp = simplicial_complexes.KleinBottle()
sage: T.join(K).homology()[3] == T_simp.join(K_simp).homology()[3] # long␣
↪time (3 seconds)
True
```

The notation '*' may be used, as well:

```
sage: S1 = delta_complexes.Sphere(1)
sage: X = S1 * S1    # X is a 3-sphere
sage: X.homology()
{0: 0, 1: 0, 2: 0, 3: Z}
```

n_chains (*n*, *base_ring=None*, *cochains=False*)

Return the free module of chains in degree `n` over `base_ring`.

INPUT:

- `n` – integer
- `base_ring` – ring (optional, default **Z**)
- `cochains` – boolean (optional, default `False`); if `True`, return cochains instead

Since the list of n-cells for a Δ-complex may have some ambiguity – for example, the list of edges may look like `[(0,0),(0,0),(0,0)]` if each edge starts and ends at vertex 0 – we record the indices of the cells along with their tuples. So the basis of chains in such a case would look like `[(0,(0,0)),(1,(0,0)),(2,(0,0))]`.

The only difference between chains and cochains is notation: the dual cochain to the chain basis element `b` is written as `\chi_b`.

EXAMPLES:

```
sage: T = delta_complexes.Torus()
sage: T.n_chains(1, QQ)
Free module generated by {(0, (0, 0)), (1, (0, 0)), (2, (0, 0))} over␣
↪Rational Field
sage: list(T.n_chains(1, QQ, cochains=False).basis())
[(0, (0, 0)), (1, (0, 0)), (2, (0, 0))]
sage: list(T.n_chains(1, QQ, cochains=True).basis())
[\chi_(0, (0, 0)), \chi_(1, (0, 0)), \chi_(2, (0, 0))]
```

n_skeleton (*n*)

The n-skeleton of this Δ-complex.

Parameters **n** (*non-negative integer*) – dimension

EXAMPLES:

```
sage: S3 = delta_complexes.Sphere(3)
sage: S3.n_skeleton(1) # 1-skeleton of a tetrahedron
Delta complex with 4 vertices and 11 simplices
sage: S3.n_skeleton(1).dimension()
1
sage: S3.n_skeleton(1).homology()
{0: 0, 1: Z x Z x Z}
```

product (*other*)

The product of this Δ-complex with another one.

Parameters **other** – another Δ-complex (the right-hand factor)

Returns the product `self x other`

Warning: If `X` and `Y` are Δ-complexes, then `X*Y` returns their join, not their product.

EXAMPLES:

```
sage: K = delta_complexes.KleinBottle()
sage: X = K.product(K)
sage: X.homology(1)
Z x Z x C2 x C2
sage: X.homology(2)
Z x C2 x C2 x C2
sage: X.homology(3)
C2
sage: X.homology(4)
0
sage: X.homology(base_ring=GF(2))
{0: Vector space of dimension 0 over Finite Field of size 2,
 1: Vector space of dimension 4 over Finite Field of size 2,
 2: Vector space of dimension 6 over Finite Field of size 2,
 3: Vector space of dimension 4 over Finite Field of size 2,
 4: Vector space of dimension 1 over Finite Field of size 2}
sage: S1 = delta_complexes.Sphere(1)
sage: K.product(S1).homology() == S1.product(K).homology()
True
sage: S1.product(S1) == delta_complexes.Torus()
True
```

subcomplex (*data*)

Create a subcomplex.

Parameters **data** – a dictionary indexed by dimension or a list (or tuple); in either case, data[n] should be the list (or tuple or set) of the indices of the simplices to be included in the subcomplex.

This automatically includes all faces of the simplices in `data` , so you only have to specify the simplices which are maximal with respect to inclusion.

EXAMPLES:

```
sage: X = delta_complexes.Torus()
sage: A = X.subcomplex({2: [0]})  # one of the triangles of X
sage: X.homology(subcomplex=A)
{0: 0, 1: 0, 2: Z}
```

In the following, `line` is a line segment and `ends` is the complex consisting of its two endpoints, so the relative homology of the two is isomorphic to the homology of a circle:

```
sage: line = delta_complexes.Simplex(1) # an edge
sage: line.cells()
{-1: ((),), 0: ((), ()), 1: ((0, 1),)}
sage: ends = line.subcomplex({0: (0, 1)})
sage: ends.cells()
{-1: ((),), 0: ((), ())}
sage: line.homology(subcomplex=ends)
{0: 0, 1: Z}
```

suspension (*n=1*)

The suspension of this Δ-complex.

Parameters **n** (*positive integer; optional, default 1*) – suspend this many times.

The suspension is the complex formed by adding two new vertices S_0 and S_1 and simplices of the form $[S_0, v_0, ..., v_k]$ and $[S_1, v_0, ..., v_k]$ for every simplex $[v_0, ..., v_k]$ in the original complex. That is, the suspension is the join of the original complex with a two-point complex (the 0-sphere).

EXAMPLES:

```
sage: S = delta_complexes.Sphere(0)
sage: S3 = S.suspension(3)  # the 3-sphere
sage: S3.homology()
{0: 0, 1: 0, 2: 0, 3: Z}
```

wedge (*right*)

The wedge (one-point union) of this Δ-complex with another one.

Parameters **right** – the other Δ-complex (the right-hand factor)

Note: This operation is not well-defined if `self` or `other` is not path-connected.

EXAMPLES:

```
sage: S1 = delta_complexes.Sphere(1)
sage: S2 = delta_complexes.Sphere(2)
sage: S1.wedge(S2).homology()
{0: 0, 1: Z, 2: Z}
```

class sage.homology.delta_complex. **DeltaComplexExamples**

Some examples of Δ-complexes.

Here are the available examples; you can also type `delta_complexes.` and hit TAB to get a list:

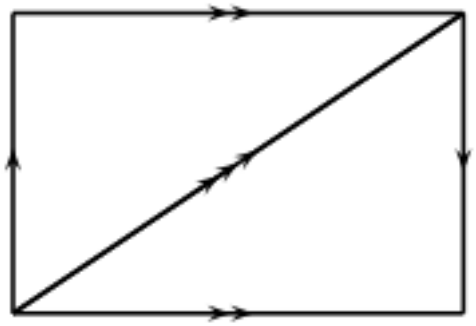
```
Sphere
Torus
RealProjectivePlane
KleinBottle
Simplex
SurfaceOfGenus
```

EXAMPLES:

```
sage: S = delta_complexes.Sphere(6) # the 6-sphere
sage: S.dimension()
6
sage: S.cohomology(6)
Z
sage: delta_complexes.Torus() == delta_complexes.Sphere(3)
False
```

KleinBottle ( )

A Δ-complex representation of the Klein bottle, consisting of one vertex, three edges, and two triangles.

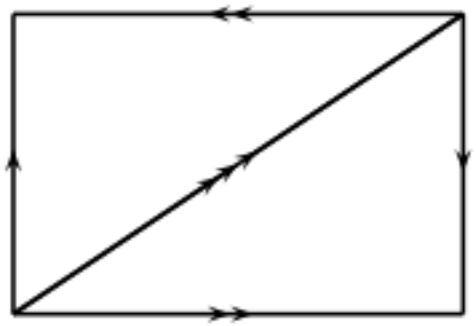


EXAMPLES:

```
sage: delta_complexes.KleinBottle()
Delta complex with 1 vertex and 7 simplices
```

RealProjectivePlane ( )

A Δ-complex representation of the real projective plane, consisting of two vertices, three edges, and two triangles.



EXAMPLES:

```
sage: P = delta_complexes.RealProjectivePlane()
sage: P.cohomology(1)
0
sage: P.cohomology(2)
C2
sage: P.cohomology(dim=1, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
sage: P.cohomology(dim=2, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
```

Simplex (n)

A Δ-complex representation of an n-simplex, consisting of a single n-simplex and its

faces. (This is the same as the simplicial complex representation available by using `simplicial_complexes.Simplex(n)`.)

EXAMPLES:

```
sage: delta_complexes.Simplex(3)
Delta complex with 4 vertices and 16 simplices
```

Sphere (*n*)

A Δ-complex representation of the n-dimensional sphere, formed by gluing two n-simplices along their boundary, except in dimension 1, in which case it is a single 1-simplex starting and ending at the same vertex.

Parameters **n** – dimension of the sphere

EXAMPLES:

```
sage: delta_complexes.Sphere(4).cohomology(4, base_ring=GF(3))
Vector space of dimension 1 over Finite Field of size 3
```

SurfaceOfGenus (*g*, *orientable=True*)

A surface of genus g as a Δ-complex.

Parameters

- **g** (*non-negative integer*) – the genus
- **orientable** (*bool, optional, default True*) – whether the surface should be orientable

In the orientable case, return a sphere if g is zero, and otherwise return a g-fold connected sum of a torus with itself.

In the non-orientable case, raise an error if g is zero. If g is positive, return a g-fold connected sum of a real projective plane with itself.

EXAMPLES:

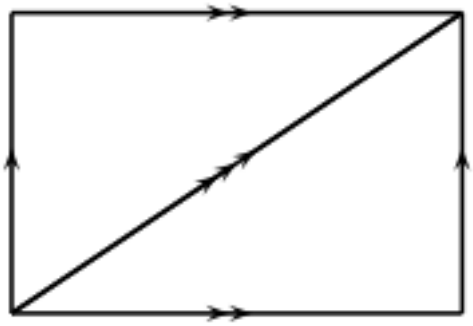
```
sage: delta_complexes.SurfaceOfGenus(1, orientable=False)
Delta complex with 2 vertices and 8 simplices
sage: delta_complexes.SurfaceOfGenus(3, orientable=False).homology(1)
Z x Z x C2
sage: delta_complexes.SurfaceOfGenus(3, orientable=False).homology(2)
0
```

Compare to simplicial complexes:

```
sage: delta_g4 = delta_complexes.SurfaceOfGenus(4)
sage: delta_g4.f_vector()
[1, 5, 33, 22]
sage: simpl_g4 = simplicial_complexes.SurfaceOfGenus(4)
sage: simpl_g4.f_vector()
[1, 19, 75, 50]
sage: delta_g4.homology() == simpl_g4.homology()
True
```

Torus ( )

A Δ-complex representation of the torus, consisting of one vertex, three edges, and two triangles.



EXAMPLES:

```
sage: delta_complexes.Torus().homology(1)
Z x Z
```

CHAPTER

ELEVEN

FINITE CUBICAL COMPLEXES

AUTHORS:

- John H. Palmieri (2009-08)

This module implements the basic structure of finite cubical complexes. For full mathematical details, see Kaczynski, Mischaikow, and Mrozek [KMM2004], for example.

Cubical complexes are topological spaces built from gluing together cubes of various dimensions; the collection of cubes must be closed under taking faces, just as with a simplicial complex. In this context, a "cube" means a product of intervals of length 1 or length 0 (degenerate intervals), with integer endpoints, and its faces are obtained by using the nondegenerate intervals: if C is a cube – a product of degenerate and nondegenerate intervals – and if $[i, i+1]$ is the k-th nondegenerate factor, then C has two faces indexed by k: the cubes obtained by replacing $[i, i+1]$ with $[i, i]$ or $[i+1, i+1]$.

So to construct a space homeomorphic to a circle as a cubical complex, we could take for example the four line segments in the plane from $(0, 2)$ to $(0, 3)$ to $(1, 3)$ to $(1, 2)$ to $(0, 2)$. In Sage, this is done with the following command:

```
sage: S1 = CubicalComplex([([0,0], [2,3]), ([0,1], [3,3]), ([0,1], [2,2]), ([1,1],␣
↪[2,3])]); S1
Cubical complex with 4 vertices and 8 cubes
```

The argument to `CubicalComplex` is a list of the maximal "cubes" in the complex. Each "cube" can be an instance of the class `Cube` or a list (or tuple) of "intervals", and an "interval" is a pair of integers, of one of the two forms $[i, i]$ or $[i, i+1]$. So the cubical complex `S1` above has four maximal cubes:

```
sage: S1.maximal_cells()
{[0,0] x [2,3], [1,1] x [2,3], [0,1] x [3,3], [0,1] x [2,2]}
```

The first of these, for instance, is the product of the degenerate interval $[0, 0]$ with the unit interval $[2, 3]$: this is the line segment in the plane from $(0, 2)$ to $(0, 3)$. We could form a topologically equivalent space by inserting some degenerate simplices:

```
sage: S1.homology()
{0: 0, 1: Z}
sage: X = CubicalComplex([([0,0], [2,3], [2]), ([0,1], [3,3], [2]), ([0,1], [2,2],␣
↪[2]), ([1,1], [2,3], [2])])
sage: X.homology()
{0: 0, 1: Z}
```

Topologically, the cubical complex `X` consists of four edges of a square in $\mathbf{R}^3$: the same unit square as `S1`, but embedded in $\mathbf{R}^3$ with z-coordinate equal to 2. Thus `X` is homeomorphic to `S1` (in fact, they're "cubically equivalent"), and this is reflected in the fact that they have isomorphic homology groups.

Note: This class derives from *GenericCellComplex* , and so inherits its methods. Some of those methods are not listed here; see the *Generic Cell Complex* page instead.

class sage.homology.cubical_complex. **Cube** (*data*)
Bases: sage.structure.sage_object.SageObject

Define a cube for use in constructing a cubical complex.

"Elementary cubes" are products of intervals with integer endpoints, each of which is either a unit interval or a degenerate (length 0) interval; for example,

$$[0, 1] \times [3, 4] \times [2, 2] \times [1, 2]$$

is a 3-dimensional cube (since one of the intervals is degenerate) embedded in $\mathbf{R}^4$.

Parameters **data** – list or tuple of terms of the form `(i,i+1)` or `(i,i)` or `(i,)` – the last two are degenerate intervals.

Returns an elementary cube

Each cube is stored in a standard form: a tuple of tuples, with a nondegenerate interval `[j,j]` represented by `(j,j)` , not `(j,)` . (This is so that for any interval `I` , `I[1]` will produce a value, not an `IndexError` .)

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C = Cube([[1,2], [5,], [6,7], [-1, 0]]); C
[1,2] x [5,5] x [6,7] x [-1,0]
sage: C.dimension() # number of nondegenerate intervals
3
sage: C.nondegenerate_intervals()  # indices of these intervals
[0, 2, 3]
sage: C.face(1, upper=False)
[1,2] x [5,5] x [6,6] x [-1,0]
sage: C.face(1, upper=True)
[1,2] x [5,5] x [7,7] x [-1,0]
sage: Cube(()).dimension()  # empty cube has dimension -1
-1
```

alexander_whitney (*dim*)
Subdivide this cube into pairs of cubes.

This provides a cubical approximation for the diagonal map $K \to K \times K$.

INPUT:

- `dim` – integer between 0 and one more than the dimension of this cube

OUTPUT:

- a list containing triples `(coeff,left,right)`

This uses the algorithm described by Pilarczyk and Réal [PR2015] on p. 267; the formula is originally due to Serre. Calling this method `alexander_whitney` is an abuse of notation, since the actual Alexander-Whitney map goes from $C(K \times L) \to C(K) \otimes C(L)$, where $C(-)$ denotes the associated chain complex, but this subdivision of cubes is at the heart of it.

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C1 = Cube([[0,1], [3,4]])
sage: C1.alexander_whitney(0)
[(1, [0,0] x [3,3], [0,1] x [3,4])]
sage: C1.alexander_whitney(1)
[(1, [0,1] x [3,3], [1,1] x [3,4]), (-1, [0,0] x [3,4], [0,1] x [4,4])]
sage: C1.alexander_whitney(2)
[(1, [0,1] x [3,4], [1,1] x [4,4])]
```

dimension ( )

The dimension of this cube: the number of its nondegenerate intervals.

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C = Cube([[1,2], [5,], [6,7], [-1, 0]])
sage: C.dimension()
3
sage: C = Cube([[1,], [5,], [6,], [-1,]])
sage: C.dimension()
0
sage: Cube([]).dimension()  # empty cube has dimension -1
-1
```

face (*n*, *upper=True*)

The nth primary face of this cube.

Parameters

- **n** – an integer between 0 and one less than the dimension of this cube
- **upper** (*boolean; optional, default=True*) – if True, return the "upper" nth primary face; otherwise, return the "lower" nth primary face.

Returns the cube obtained by replacing the nth non-degenrate interval with either its upper or lower endpoint.

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C = Cube([[1,2], [5,], [6,7], [-1, 0]]); C
[1,2] x [5,5] x [6,7] x [-1,0]
sage: C.face(0)
[2,2] x [5,5] x [6,7] x [-1,0]
sage: C.face(0, upper=False)
[1,1] x [5,5] x [6,7] x [-1,0]
sage: C.face(1)
[1,2] x [5,5] x [7,7] x [-1,0]
sage: C.face(2, upper=False)
[1,2] x [5,5] x [6,7] x [-1,-1]
sage: C.face(3)
Traceback (most recent call last):
...
ValueError: Can only compute the nth face if 0 <= n < dim.
```

faces ( )

The list of faces (of codimension 1) of this cube.

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C = Cube([[1,2], [3,4]])
sage: C.faces()
[[2,2] x [3,4], [1,2] x [4,4], [1,1] x [3,4], [1,2] x [3,3]]
```

faces_as_pairs ( )

The list of faces (of codimension 1) of this cube, as pairs (upper, lower).

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C = Cube([[1,2], [3,4]])
sage: C.faces_as_pairs()
[([2,2] x [3,4], [1,1] x [3,4]), ([1,2] x [4,4], [1,2] x [3,3])]
```

is_face (*other*)

Return True iff this cube is a face of other.

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C1 = Cube([[1,2], [5,], [6,7], [-1, 0]])
sage: C2 = Cube([[1,2], [5,], [6,], [-1, 0]])
sage: C1.is_face(C2)
False
sage: C1.is_face(C1)
True
sage: C2.is_face(C1)
True
```

nondegenerate_intervals ( )

The list of indices of nondegenerate intervals of this cube.

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C = Cube([[1,2], [5,], [6,7], [-1, 0]])
sage: C.nondegenerate_intervals()
[0, 2, 3]
sage: C = Cube([[1,], [5,], [6,], [-1,]])
sage: C.nondegenerate_intervals()
[]
```

product (*other*)

Cube obtained by concatenating the underlying tuples of the two arguments.

Parameters **other** – another cube

Returns the product of `self` and `other` , as a Cube

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C = Cube([[1,2], [3,]])
sage: D = Cube([[4], [0,1]])
sage: C.product(D)
[1,2] x [3,3] x [4,4] x [0,1]
```

You can also use `__add__` or `+` or `__mul__` or `*` :

```
sage: D * C
[4,4] x [0,1] x [1,2] x [3,3]
sage: D + C * C
[4,4] x [0,1] x [1,2] x [3,3] x [1,2] x [3,3]
```

tuple ( )

The tuple attached to this cube.

EXAMPLES:

```
sage: from sage.homology.cubical_complex import Cube
sage: C = Cube([[1,2], [5,], [6,7], [-1, 0]])
sage: C.tuple()
((1, 2), (5, 5), (6, 7), (-1, 0))
```

class sage.homology.cubical_complex. **CubicalComplex** (*maximal_faces=[]*, *maximality_check=True*)

Bases: *sage.homology.cell_complex.GenericCellComplex*

Define a cubical complex.

Parameters

- **maximal_faces** – set of maximal faces
- **maximality_check** (*boolean; optional, default True*) – see below

Returns a cubical complex

maximal_faces should be a list or tuple or set (or anything which may be converted to a set) of "cubes": instances of the class *Cube* , or lists or tuples suitable for conversion to cubes. These cubes are the maximal cubes in the complex.

In addition, maximal_faces may be a cubical complex, in which case that complex is returned. Also, maximal_faces may instead be any object which has a _cubical_ method (e.g., a simplicial complex); then that method is used to convert the object to a cubical complex.

If maximality_check is True, check that each maximal face is, in fact, maximal. In this case, when producing the internal representation of the cubical complex, omit those that are not. It is highly recommended that this be True; various methods for this class may fail if faces which are claimed to be maximal are in fact not.

EXAMPLES:

The empty complex, consisting of one cube, the empty cube:

```
sage: CubicalComplex()
Cubical complex with 0 vertices and 1 cube
```

A "circle" (four edges connecting the vertices (0,2), (0,3), (1,2), and (1,3)):

```
sage: S1 = CubicalComplex([([0,0], [2,3]), ([0,1], [3,3]), ([0,1], [2,2]),␣
↪([1,1], [2,3])])
sage: S1
Cubical complex with 4 vertices and 8 cubes
sage: S1.homology()
{0: 0, 1: Z}
```

A set of five points and its product with S1 :

```
sage: pts = CubicalComplex([([0],), ([3],), ([6],), ([-12],), ([5],)])
sage: pts
Cubical complex with 5 vertices and 5 cubes
sage: pts.homology()
{0: Z x Z x Z x Z}
sage: X = S1.product(pts); X
Cubical complex with 20 vertices and 40 cubes
sage: X.homology()
{0: Z x Z x Z x Z, 1: Z^5}
```

Converting a simplicial complex to a cubical complex:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: C2 = CubicalComplex(S2)
sage: all([C2.homology(n) == S2.homology(n) for n in range(3)])
True
```

You can get the set of maximal cells or a dictionary of all cells:

```
sage: X.maximal_cells()
{[0,0] x [2,3] x [-12,-12], [0,1] x [3,3] x [5,5], [0,1] x [2,2] x [3,3], [0,1] x␣
↪[2,2] x [0,0], [0,1] x [3,3] x [6,6], [1,1] x [2,3] x [0,0], [0,1] x [2,2] x [-
↪12,-12], [0,0] x [2,3] x [6,6], [1,1] x [2,3] x [-12,-12], [1,1] x [2,3] x␣
↪[5,5], [0,1] x [2,2] x [5,5], [0,1] x [3,3] x [3,3], [1,1] x [2,3] x [3,3],␣
↪[0,0] x [2,3] x [5,5], [0,1] x [3,3] x [0,0], [1,1] x [2,3] x [6,6], [0,1] x␣
↪[2,2] x [6,6], [0,0] x [2,3] x [0,0], [0,0] x [2,3] x [3,3], [0,1] x [3,3] x [-
↪12,-12]}
sage: S1.cells()
{-1: set(),
 0: {[0,0] x [2,2], [0,0] x [3,3], [1,1] x [2,2], [1,1] x [3,3]},
 1: {[0,0] x [2,3], [0,1] x [2,2], [0,1] x [3,3], [1,1] x [2,3]}}
```

Chain complexes, homology, and cohomology:

```
sage: T = S1.product(S1); T
Cubical complex with 16 vertices and 64 cubes
sage: T.chain_complex()
Chain complex with at most 3 nonzero terms over Integer Ring
sage: T.homology(base_ring=QQ)
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
sage: RP2 = cubical_complexes.RealProjectivePlane()
sage: RP2.cohomology(dim=[1, 2], base_ring=GF(2))
{1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
```

Joins are not implemented:

```
sage: S1.join(S1)
Traceback (most recent call last):
...
NotImplementedError: Joins are not implemented for cubical complexes.
```

Therefore, neither are cones or suspensions.

alexander_whitney (*cube*, *dim_left*)

Subdivide `cube` in this cubical complex into pairs of cubes.

See *Cube.alexander_whitney()* for more details. This method just calls that one.

INPUT:

- `cube` – a cube in this cubical complex
- `dim` – integer between 0 and one more than the dimension of this cube

OUTPUT: a list containing triples `(coeff,left,right)`

EXAMPLES:

```
sage: C = cubical_complexes.Cube(3)
sage: c = list(C.n_cubes(3))[0]; c
[0,1] x [0,1] x [0,1]
sage: C.alexander_whitney(c, 1)
[(1, [0,1] x [0,0] x [0,0], [1,1] x [0,1] x [0,1]),
 (-1, [0,0] x [0,1] x [0,0], [0,1] x [1,1] x [0,1]),
 (1, [0,0] x [0,0] x [0,1], [0,1] x [0,1] x [1,1])]
```

algebraic_topological_model (*base_ring=None*)

Algebraic topological model for this cubical complex with coefficients in `base_ring` .

The term "algebraic topological model" is defined by Pilarczyk and Réal [PR2015].

INPUT:

- `base_ring` - coefficient ring (optional, default `QQ`). Must be a field.

Denote by C the chain complex associated to this cubical complex. The algebraic topological model is a chain complex M with zero differential, with the same homology as C, along with chain maps $\pi : C \to M$ and $\iota : M \to C$ satisfying $\iota\pi = 1_M$ and $\pi\iota$ chain homotopic to 1_C. The chain homotopy ϕ must satisfy

- $\phi\phi = 0$,
- $\pi\phi = 0$,
- $\phi\iota = 0$.

Such a chain homotopy is called a *chain contraction*.

OUTPUT: a pair consisting of

- chain contraction `phi` associated to C, M, π, and ι
- the chain complex M

Note that from the chain contraction `phi` , one can recover the chain maps π and ι via `phi.pi()` and `phi.iota()` . Then one can recover C and M from, for example, `phi.pi().domain()` and `phi.pi().codomain()` , respectively.

EXAMPLES:

```
sage: RP2 = cubical_complexes.RealProjectivePlane()
sage: phi, M = RP2.algebraic_topological_model(GF(2))
sage: M.homology()
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: T = cubical_complexes.Torus()
sage: phi, M = T.algebraic_topological_model(QQ)
sage: M.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

cells (*subcomplex=None*)

The cells of this cubical complex, in the form of a dictionary: the keys are integers, representing dimension, and the value associated to an integer d is the list of d-cells.

If the optional argument `subcomplex` is present, then return only the faces which are *not* in the subcomplex.

Parameters **subcomplex** (*a cubical complex; optional, default None*) – a subcomplex of this cubical complex

Returns cells of this complex not contained in `subcomplex`

Return type dictionary

EXAMPLES:

```
sage: S2 = cubical_complexes.Sphere(2)
sage: sorted(S2.cells()[2])
[[0,0] x [0,1] x [0,1],
 [0,1] x [0,0] x [0,1],
 [0,1] x [0,1] x [0,0],
 [0,1] x [0,1] x [1,1],
 [0,1] x [1,1] x [0,1],
 [1,1] x [0,1] x [0,1]]
```

chain_complex (*subcomplex=None*, *augmented=False*, *verbose=False*, *check=False*, *dimensions=None*, *base_ring=Integer Ring*, *cochain=False*)

The chain complex associated to this cubical complex.

Parameters

- **dimensions** – if None, compute the chain complex in all dimensions. If a list or tuple of integers, compute the chain complex in those dimensions, setting the chain groups in all other dimensions to zero. NOT IMPLEMENTED YET: this function always returns the entire chain complex
- **base_ring** (*optional, default ZZ*) – commutative ring
- **subcomplex** (*optional, default empty*) – a subcomplex of this cubical complex. Compute the chain complex relative to this subcomplex.
- **augmented** (*boolean; optional, default False*) – If True, return the augmented chain complex (that is, include a class in dimension -1 corresponding to the empty cell). This is ignored if `dimensions` is specified.
- **cochain** (*boolean; optional, default False*) – If True, return the cochain complex (that is, the dual of the chain complex).
- **verbose** (*boolean; optional, default False*) – If True, print some messages as the chain complex is computed.
- **check** (*boolean; optional, default False*) – If True, make sure that the chain complex is actually a chain complex: the differentials are composable and their product is zero.

Note: If subcomplex is nonempty, then the argument `augmented` has no effect: the chain complex relative to a nonempty subcomplex is zero in dimension -1.

EXAMPLES:

```
sage: S2 = cubical_complexes.Sphere(2)
sage: S2.chain_complex()
Chain complex with at most 3 nonzero terms over Integer Ring
sage: Prod = S2.product(S2); Prod
Cubical complex with 64 vertices and 676 cubes
sage: Prod.chain_complex()
Chain complex with at most 5 nonzero terms over Integer Ring
sage: Prod.chain_complex(base_ring=QQ)
Chain complex with at most 5 nonzero terms over Rational Field
sage: C1 = cubical_complexes.Cube(1)
sage: S0 = cubical_complexes.Sphere(0)
sage: C1.chain_complex(subcomplex=S0)
Chain complex with at most 1 nonzero terms over Integer Ring
sage: C1.homology(subcomplex=S0)
{0: 0, 1: Z}
```

cone ( )

The cone on this cubical complex.

NOT IMPLEMENTED

The cone is the complex formed by taking the join of the original complex with a one-point complex (that is, a 0-dimensional cube). Since joins are not implemented for cubical complexes, neither are cones.

EXAMPLES:

```
sage: C1 = cubical_complexes.Cube(1)
sage: C1.cone()
Traceback (most recent call last):
...
NotImplementedError: Cones are not implemented for cubical complexes.
```

connected_sum (*other*)

Return the connected sum of self with other.

Parameters **other** – another cubical complex

Returns the connected sum `self # other`

Warning: This does not check that self and other are manifolds, only that their facets all have the same dimension. Since a (more or less) random facet is chosen from each complex and then glued together, this method may return random results if applied to non-manifolds, depending on which facet is chosen.

EXAMPLES:

```
sage: T = cubical_complexes.Torus()
sage: S2 = cubical_complexes.Sphere(2)
sage: T.connected_sum(S2).cohomology() == T.cohomology()
True
sage: RP2 = cubical_complexes.RealProjectivePlane()
sage: T.connected_sum(RP2).homology(1)
Z x Z x C2
sage: RP2.connected_sum(RP2).connected_sum(RP2).homology(1)
Z x Z x C2
```

disjoint_union (*other*)

The disjoint union of this cubical complex with another one.

Parameters **right** – the other cubical complex (the right-hand factor)

Algorithm: first embed both complexes in d-dimensional Euclidean space. Then embed in (1+d)-dimensional space, calling the new axis x, and putting the first complex at $x = 0$, the second at $x = 1$.

EXAMPLES:

```
sage: S1 = cubical_complexes.Sphere(1)
sage: S2 = cubical_complexes.Sphere(2)
sage: S1.disjoint_union(S2).homology()
{0: Z, 1: Z, 2: Z}
```

graph ( )

The 1-skeleton of this cubical complex, as a graph.

EXAMPLES:

```
sage: cubical_complexes.Sphere(2).graph()
Graph on 8 vertices
```

is_pure ( )

True iff this cubical complex is pure: that is, all of its maximal faces have the same dimension.

Warning: This may give the wrong answer if the cubical complex was constructed with `maximality_check` set to False.

EXAMPLES:

```
sage: S4 = cubical_complexes.Sphere(4)
sage: S4.is_pure()
True
sage: C = CubicalComplex([([0,0], [3,3]), ([1,2], [4,5])])
sage: C.is_pure()
False
```

is_subcomplex (*other*)

Return True if `self` is a subcomplex of `other` .

Parameters **other** – a cubical complex

Each maximal cube of `self` must be a face of a maximal cube of `other` for this to be True.

EXAMPLES:

```
sage: S1 = cubical_complexes.Sphere(1)
sage: C0 = cubical_complexes.Cube(0)
sage: C1 = cubical_complexes.Cube(1)
sage: cyl = S1.product(C1)
sage: end = S1.product(C0)
sage: end.is_subcomplex(cyl)
True
sage: cyl.is_subcomplex(end)
False
```

The embedding of the cubical complex is important here:

```
sage: C2 = cubical_complexes.Cube(2)
sage: C1.is_subcomplex(C2)
```

```
False
sage: C1.product(C0).is_subcomplex(C2)
True
```

`C1` is not a subcomplex of `C2` because it's not embedded in $\mathbf{R}^2$. On the other hand, `C1 x C0` is a face of `C2`. Look at their maximal cells:

```
sage: C1.maximal_cells()
{[0,1]}
sage: C2.maximal_cells()
{[0,1] x [0,1]}
sage: C1.product(C0).maximal_cells()
{[0,1] x [0,0]}
```

join (*other*)

The join of this cubical complex with another one.

NOT IMPLEMENTED.

Parameters **other** – another cubical complex

EXAMPLES:

```
sage: C1 = cubical_complexes.Cube(1)
sage: C1.join(C1)
Traceback (most recent call last):
...
NotImplementedError: Joins are not implemented for cubical complexes.
```

maximal_cells ( )

The set of maximal cells (with respect to inclusion) of this cubical complex.

Returns Set of maximal cells

This just returns the set of cubes used in defining the cubical complex, so if the complex was defined with no maximality checking, none is done here, either.

EXAMPLES:

```
sage: interval = cubical_complexes.Cube(1)
sage: interval
Cubical complex with 2 vertices and 3 cubes
sage: interval.maximal_cells()
{[0,1]}
sage: interval.product(interval).maximal_cells()
{[0,1] x [0,1]}
```

n_cubes (*n*, *subcomplex=None*)

The set of cubes of dimension n of this cubical complex. If the optional argument `subcomplex` is present, then return the `n`-dimensional cubes which are *not* in the subcomplex.

Parameters

- **n** (*integer*) – dimension
- **subcomplex** (*a cubical complex; optional, default None*) – a subcomplex of this cubical complex

Returns cells in dimension `n`

Return type *set*

EXAMPLES:

```
sage: C = cubical_complexes.Cube(3)
sage: C.n_cubes(3)
{[0,1] x [0,1] x [0,1]}
sage: sorted(C.n_cubes(2))
[[0,0] x [0,1] x [0,1],
 [0,1] x [0,0] x [0,1],
 [0,1] x [0,1] x [0,0],
 [0,1] x [0,1] x [1,1],
 [0,1] x [1,1] x [0,1],
 [1,1] x [0,1] x [0,1]]
```

n_skeleton (*n*)

The n-skeleton of this cubical complex.

Parameters **n** (*non-negative integer*) – dimension

Returns cubical complex

EXAMPLES:

```
sage: S2 = cubical_complexes.Sphere(2)
sage: C3 = cubical_complexes.Cube(3)
sage: S2 == C3.n_skeleton(2)
True
```

product (*other*)

The product of this cubical complex with another one.

Parameters **other** – another cubical complex

EXAMPLES:

```
sage: RP2 = cubical_complexes.RealProjectivePlane()
sage: S1 = cubical_complexes.Sphere(1)
sage: RP2.product(S1).homology()[1] # long time: 5 seconds
Z x C2
```

suspension (*n=1*)

The suspension of this cubical complex.

NOT IMPLEMENTED

Parameters **n** (*positive integer; optional, default 1*) – suspend this many times

The suspension is the complex formed by taking the join of the original complex with a two-point complex (the 0-sphere). Since joins are not implemented for cubical complexes, neither are suspensions.

EXAMPLES:

```
sage: C1 = cubical_complexes.Cube(1)
sage: C1.suspension()
Traceback (most recent call last):
...
NotImplementedError: Suspensions are not implemented for cubical complexes.
```

wedge (*other*)

The wedge (one-point union) of this cubical complex with another one.

Parameters **right** – the other cubical complex (the right-hand factor)

Algorithm: if `self` is embedded in d dimensions and `other` in n dimensions, embed them in $d + n$ dimensions: `self` using the first d coordinates, `other` using the last n, translating them so that they have the origin as a common vertex.

Note: This operation is not well-defined if `self` or `other` is not path-connected.

EXAMPLES:

```
sage: S1 = cubical_complexes.Sphere(1)
sage: S2 = cubical_complexes.Sphere(2)
sage: S1.wedge(S2).homology()
{0: 0, 1: Z, 2: Z}
```

class sage.homology.cubical_complex. **CubicalComplexExamples**

Some examples of cubical complexes.

Here are the available examples; you can also type "cubical_complexes." and hit TAB to get a list:

```
Sphere
Torus
RealProjectivePlane
KleinBottle
SurfaceOfGenus
Cube
```

EXAMPLES:

```
sage: cubical_complexes.Torus()  # indirect doctest
Cubical complex with 16 vertices and 64 cubes
sage: cubical_complexes.Cube(7)
Cubical complex with 128 vertices and 2187 cubes
sage: cubical_complexes.Sphere(7)
Cubical complex with 256 vertices and 6560 cubes
```

Cube (*n*)

A cubical complex representation of an n-dimensional cube.

Parameters **n** (*non-negative integer*) – the dimension

EXAMPLES:

```
sage: cubical_complexes.Cube(0)
Cubical complex with 1 vertex and 1 cube
sage: cubical_complexes.Cube(3)
Cubical complex with 8 vertices and 27 cubes
```

KleinBottle ( )

A cubical complex representation of the Klein bottle, formed by taking the connected sum of the real projective plane with itself.

EXAMPLES:

```
sage: cubical_complexes.KleinBottle()
Cubical complex with 42 vertices and 168 cubes
```

RealProjectivePlane ( )

A cubical complex representation of the real projective plane. This is taken from the examples from CHomP, the Computational Homology Project: http://chomp.rutgers.edu/.

EXAMPLES:

```
sage: cubical_complexes.RealProjectivePlane()
Cubical complex with 21 vertices and 81 cubes
```

Sphere (*n*)

A cubical complex representation of the n-dimensional sphere, formed by taking the boundary of an $(n+1)$-dimensional cube.

Parameters **n** (*non-negative integer*) – the dimension of the sphere

EXAMPLES:

```
sage: cubical_complexes.Sphere(7)
Cubical complex with 256 vertices and 6560 cubes
```

SurfaceOfGenus (*g*, *orientable=True*)

A surface of genus g as a cubical complex.

Parameters

- **g** (*non-negative integer*) – the genus
- **orientable** (*bool, optional, default True*) – whether the surface should be orientable

In the orientable case, return a sphere if g is zero, and otherwise return a g-fold connected sum of a torus with itself.

In the non-orientable case, raise an error if g is zero. If g is positive, return a g-fold connected sum of a real projective plane with itself.

EXAMPLES:

```
sage: cubical_complexes.SurfaceOfGenus(2)
Cubical complex with 32 vertices and 134 cubes
sage: cubical_complexes.SurfaceOfGenus(1, orientable=False)
Cubical complex with 21 vertices and 81 cubes
```

Torus ( )

A cubical complex representation of the torus, obtained by taking the product of the circle with itself.

EXAMPLES:

```
sage: cubical_complexes.Torus()
Cubical complex with 16 vertices and 64 cubes
```

CHAPTER

TWELVE

GENERIC CELL COMPLEXES

AUTHORS:

- John H. Palmieri (2009-08)

This module defines a class of abstract finite cell complexes. This is meant as a base class from which other classes (like *SimplicialComplex* , *CubicalComplex* , and *DeltaComplex*) should derive. As such, most of its properties are not implemented. It is meant for use by developers producing new classes, not casual users.

Note: Keywords for *chain_complex()* , *homology()* , etc.: any keywords given to the *homology()* method get passed on to the *chain_complex()* method and also to the constructor for chain complexes in *sage.homology.chain_complex.ChainComplex_class* , as well as its associated *homology()* method. This means that those keywords should have consistent meaning in all of those situations. It also means that it is easy to implement new keywords: for example, if you implement a new keyword for the *sage.homology.chain_complex.ChainComplex_class.homology()* method, then it will be automatically accessible through the *homology()* method for cell complexes – just make sure it gets documented.

---

class sage.homology.cell_complex. **GenericCellComplex**

Bases: sage.structure.sage_object.SageObject

Class of abstract cell complexes.

This is meant to be used by developers to produce new classes, not by casual users. Classes which derive from this are *SimplicialComplex* , *DeltaComplex* , and *CubicalComplex* .

Most of the methods here are not implemented, but probably should be implemented in a derived class. Most of the other methods call a non-implemented one; their docstrings contain examples from derived classes in which the various methods have been defined. For example, *homology()* calls *chain_complex()* ; the class *DeltaComplex* implements *chain_complex()* , and so the *homology()* method here is illustrated with examples involving Δ-complexes.

EXAMPLES:

It's hard to give informative examples of the base class, since essentially nothing is implemented.

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
```

alexander_whitney (*cell*, *dim_left*)

The decomposition of `cell` in this complex into left and right factors, suitable for computing cup products. This should provide a cellular approximation for the diagonal map $K \to K \times K$.

This method is not implemented for generic cell complexes, but must be implemented for any derived class to make cup products work in `self.cohomology_ring()` .

INPUT:

- `cell` – a cell in this complex
- `dim_left` – the dimension of the left-hand factors in the decomposition

OUTPUT: a list containing triples `(c,left,right)`. `left` and `right` should be cells in this complex, and `c` an integer. In the cellular approximation of the diagonal map, the chain represented by `cell` should get sent to the sum of terms $c(left \otimes right)$ in the tensor product $C(K) \otimes C(K)$ of the chain complex for this complex with itself.

This gets used in the method *product_on_basis()* for the class of cohomology rings.

For simplicial and cubical complexes, the decomposition can be done at the level of individual cells: see *alexander_whitney()* and *alexander_whitney()*. Then the method for simplicial complexes just calls the method for individual simplices, and similarly for cubical complexes. For Δ-complexes, the method is instead defined at the level of the cell complex.

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.alexander_whitney(None, 2)
Traceback (most recent call last):
...
NotImplementedError: <abstract method alexander_whitney at ...>
```

algebraic_topological_model (*base_ring=Rational Field*)

Algebraic topological model for this cell complex with coefficients in `base_ring`.

The term "algebraic topological model" is defined by Pilarczyk and Réal [PR2015].

This is not implemented for generic cell complexes. For any classes deriving from this one, when this method is implemented, it should essentially just call either *algebraic_topological_model()* or *algebraic_topological_model_delta_complex()*.

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.algebraic_topological_model(QQ)
Traceback (most recent call last):
...
NotImplementedError
```

betti (*dim=None*, *subcomplex=None*)

The Betti numbers of this simplicial complex as a dictionary (or a single Betti number, if only one dimension is given): the ith Betti number is the rank of the ith homology group.

Parameters

- **dim** (integer or list of integers or `None`; optional, default `None`) – If `None`, then return every Betti number, as a dictionary with keys the non-negative integers. If `dim` is an integer or list, return the Betti number for each given dimension. (Actually, if `dim` is a list, return the Betti numbers, as a dictionary, in the range from `min(dim)` to `max(dim)`. If `dim` is a number, return the Betti number in that dimension.)
- **subcomplex** (optional, default `None`) – a subcomplex of this cell complex. Compute the Betti numbers of the homology relative to this subcomplex.

EXAMPLES:

Build the two-sphere as a three-fold join of a two-point space with itself:

```
sage: S = SimplicialComplex([[0], [1]])
sage: (S*S*S).betti()
{0: 1, 1: 0, 2: 1}
sage: (S*S*S).betti([1,2])
{1: 0, 2: 1}
sage: (S*S*S).betti(2)
1
```

Or build the two-sphere as a Δ-complex:

```
sage: S2 = delta_complexes.Sphere(2)
sage: S2.betti([1,2])
{1: 0, 2: 1}
```

Or as a cubical complex:

```
sage: S2c = cubical_complexes.Sphere(2)
sage: S2c.betti(2)
1
```

cells (*subcomplex=None*)

The cells of this cell complex, in the form of a dictionary: the keys are integers, representing dimension, and the value associated to an integer d is the set of d-cells. If the optional argument `subcomplex` is present, then return only the faces which are *not* in the subcomplex.

Parameters **subcomplex** (*optional, default None*) – a subcomplex of this cell complex. Return the cells which are not in this subcomplex.

This is not implemented in general; it should be implemented in any derived class. When implementing, see the warning in the *dimension()* method.

This method is used by various other methods, such as *n_cells()* and *f_vector()* .

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.cells()
Traceback (most recent call last):
...
NotImplementedError: <abstract method cells at ...>
```

chain_complex (*subcomplex=None*, *augmented=False*, *verbose=False*, *check=True*, *dimensions=None*, *base_ring='ZZ'*, *cochain=False*)

This is not implemented for general cell complexes.

Some keywords to possibly implement in a derived class:

- `subcomplex` – a subcomplex: compute the relative chain complex
- `augmented` – a bool: whether to return the augmented complex
- `verbose` – a bool: whether to print informational messages as the chain complex is being computed
- `check` – a bool: whether to check that the each composite of two consecutive differentials is zero
- **`dimensions` – if `None`, compute the chain complex in all** dimensions. If a list or tuple of integers, compute the chain complex in those dimensions, setting the chain groups in all other dimensions to zero.

Definitely implement the following:

- base_ring – commutative ring (optional, default ZZ)
- cochain – a bool: whether to return the cochain complex

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.chain_complex()
Traceback (most recent call last):
...
NotImplementedError: <abstract method chain_complex at ...>
```

cohomology (*dim=None*, *base_ring=Integer Ring*, *subcomplex=None*, *generators=False*, *algorithm='pari'*, *verbose=False*, *reduced=True*)

The reduced cohomology of this cell complex.

The arguments are the same as for the *homology()* method, except that *homology()* accepts a cohomology key word, while this function does not: cohomology is automatically true here. Indeed, this function just calls *homology()* with cohomology set to True .

Parameters

- **dim** –
- **base_ring** –
- **subcomplex** –
- **algorithm** –
- **verbose** –
- **reduced** –

EXAMPLES:

```
sage: circle = SimplicialComplex([[0,1], [1,2], [0, 2]])
sage: circle.cohomology(0)
0
sage: circle.cohomology(1)
Z
sage: P2 = SimplicialComplex([[0,1,2], [0,2,3], [0,1,5], [0,4,5], [0,3,4],␣
↪[1,2,4], [1,3,4], [1,3,5], [2,3,5], [2,4,5]])   # projective plane
sage: P2.cohomology(2)
C2
sage: P2.cohomology(2, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
sage: P2.cohomology(2, base_ring=GF(3))
Vector space of dimension 0 over Finite Field of size 3

sage: cubical_complexes.KleinBottle().cohomology(2)
C2
```

Relative cohomology:

```
sage: T = SimplicialComplex([[0,1]])
sage: U = SimplicialComplex([[0], [1]])
sage: T.cohomology(1, subcomplex=U)
Z
```

A Δ-complex example:

```
sage: s5 = delta_complexes.Sphere(5)
sage: s5.cohomology(base_ring=GF(7))[5]
Vector space of dimension 1 over Finite Field of size 7
```

cohomology_ring (*base_ring=Rational Field*)

Return the unreduced cohomology with coefficients in `base_ring` with a chosen basis.

This is implemented for simplicial, cubical, and Δ-complexes, not for arbitrary generic cell complexes. The resulting elements are suitable for computing cup products. For simplicial complexes, they should be suitable for computing cohomology operations; so far, only mod 2 cohomology operations have been implemented.

INPUT:

- `base_ring` – coefficient ring (optional, default `QQ`); must be a field

The basis elements in dimension `dim` are named 'h^{dim,i}' where i ranges between 0 and $r-1$, if r is the rank of the cohomology group.

Note: For all but the smallest complexes, this is likely to be slower than *cohomology()* (with field coefficients), possibly by several orders of magnitute. This and its companion *homology_with_basis()* carry extra information which allows computation of cup products, for example, but because of speed issues, you may only wish to use these if you need that extra information.

EXAMPLES:

```
sage: K = simplicial_complexes.KleinBottle()
sage: H = K.cohomology_ring(QQ); H
Cohomology ring of Minimal triangulation of the Klein bottle
 over Rational Field
sage: sorted(H.basis(), key=str)
[h^{0,0}, h^{1,0}]
sage: H = K.cohomology_ring(GF(2)); H
Cohomology ring of Minimal triangulation of the Klein bottle
 over Finite Field of size 2
sage: sorted(H.basis(), key=str)
[h^{0,0}, h^{1,0}, h^{1,1}, h^{2,0}]

sage: X = delta_complexes.SurfaceOfGenus(2)
sage: H = X.cohomology_ring(QQ); H
Cohomology ring of Delta complex with 3 vertices and 29 simplices
 over Rational Field
sage: sorted(H.basis(1), key=str)
[h^{1,0}, h^{1,1}, h^{1,2}, h^{1,3}]

sage: H = simplicial_complexes.Torus().cohomology_ring(QQ); H
Cohomology ring of Minimal triangulation of the torus
 over Rational Field
sage: x = H.basis()[1,0]; x
h^{1,0}
sage: y = H.basis()[1,1]; y
h^{1,1}
```

You can compute cup products of cohomology classes:

```
sage: x.cup_product(y)
-h^{2,0}
```

```
sage: x * y # alternate notation
-h^{2,0}
sage: y.cup_product(x)
h^{2,0}
sage: x.cup_product(x)
0
```

Cohomology operations:

```
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: K = RP2.suspension()
sage: K.set_immutable()
sage: y = K.cohomology_ring(GF(2)).basis()[2,0]; y
h^{2,0}
sage: y.Sq(1)
h^{3,0}
```

To compute the cohomology ring, the complex must be "immutable". This is only relevant for simplicial complexes, and most simplicial complexes are immutable, but certain constructions make them mutable. The suspension is one example, and this is the reason for calling `K.set_immutable()` above. Another example:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: T = S1.product(S1)
sage: T.is_immutable()
False
sage: T.cohomology_ring()
Traceback (most recent call last):
...
ValueError: This simplicial complex must be immutable. Call set_immutable().
sage: T.set_immutable()
sage: T.cohomology_ring()
Cohomology ring of Simplicial complex with 9 vertices and
18 facets over Rational Field
```

dimension ( )

The dimension of this cell complex: the maximum dimension of its cells.

Warning: If the *cells()* method calls *dimension()*, then you'll get an infinite loop. So either don't use *dimension()* or override *dimension()*.

EXAMPLES:

```
sage: simplicial_complexes.RandomComplex(d=5, n=8).dimension()
5
sage: delta_complexes.Sphere(3).dimension()
3
sage: T = cubical_complexes.Torus()
sage: T.product(T).dimension()
4
```

disjoint_union (*right*)

The disjoint union of this cell complex with another one.

Parameters **right** – the other cell complex (the right-hand factor)

Disjoint unions are not implemented for general cell complexes.

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex(); B = GenericCellComplex()
sage: A.disjoint_union(B)
Traceback (most recent call last):
...
NotImplementedError: <abstract method disjoint_union at ...>
```

euler_characteristic ( )

The Euler characteristic of this cell complex: the alternating sum over $n \geq 0$ of the number of n-cells.

EXAMPLES:

```
sage: simplicial_complexes.Simplex(5).euler_characteristic()
1
sage: delta_complexes.Sphere(6).euler_characteristic()
2
sage: cubical_complexes.KleinBottle().euler_characteristic()
0
```

f_vector ( )

The f-vector of this cell complex: a list whose n^{th} item is the number of $(n-1)$-cells. Note that, like all lists in Sage, this is indexed starting at 0: the 0th element in this list is the number of (-1)-cells (which is 1: the empty cell is the only (-1)-cell).

EXAMPLES:

```
sage: simplicial_complexes.KleinBottle().f_vector()
[1, 8, 24, 16]
sage: delta_complexes.KleinBottle().f_vector()
[1, 1, 3, 2]
sage: cubical_complexes.KleinBottle().f_vector()
[1, 42, 84, 42]
```

face_poset ( )

The face poset of this cell complex, the poset of nonempty cells, ordered by inclusion.

This uses the *cells()* method, and also assumes that for each cell `f`, all of `f.faces()`, `tuple(f)`, and `f.dimension()` make sense. (If this is not the case in some derived class, as happens with Δ-complexes, then override this method.)

EXAMPLES:

```
sage: P = SimplicialComplex([[0, 1], [1,2], [2,3]]).face_poset(); P
Finite poset containing 7 elements
sage: P.list()
[(3,), (2,), (2, 3), (1,), (1, 2), (0,), (0, 1)]

sage: S2 = cubical_complexes.Sphere(2)
sage: S2.face_poset()
Finite poset containing 26 elements
```

graph ( )

The 1-skeleton of this cell complex, as a graph.

This is not implemented for general cell complexes.

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.graph()
Traceback (most recent call last):
...
NotImplementedError
```

homology (*dim=None*, *base_ring=Integer Ring*, *subcomplex=None*, *generators=False*, *cohomology=False*, *algorithm='pari'*, *verbose=False*, *reduced=True*, ***kwds*)

The (reduced) homology of this cell complex.

Parameters

- **dim** (*integer or list of integers or None; optional, default None*) – If None, then return the homology in every dimension. If `dim` is an integer or list, return the homology in the given dimensions. (Actually, if `dim` is a list, return the homology in the range from `min(dim)` to `max(dim)` .)
- **base_ring** (*optional, default ZZ*) – commutative ring, must be ZZ or a field.
- **subcomplex** (*optional, default empty*) – a subcomplex of this simplicial complex. Compute homology relative to this subcomplex.
- **generators** (*boolean; optional, default False*) – If `True` , return generators for the homology groups along with the groups. NOTE: Since trac ticket #6100, the result may not be what you expect when not using CHomP since its return is in terms of the chain complex.
- **cohomology** (*boolean; optional, default False*) – If True, compute cohomology rather than homology.
- **algorithm** (*string; optional, default 'pari'*) – The options are 'auto', 'dhsw', 'pari' or 'no_chomp'. See below for a description of what they mean.
- **verbose** (*boolean; optional, default False*) – If True, print some messages as the homology is computed.
- **reduced** (boolean; optional, default `True`) – If `True` , return the reduced homology.

ALGORITHM:

If `algorithm` is set to 'auto', then use CHomP if available. (CHomP is available at the web page http://chomp.rutgers.edu/. It is also an optional package for Sage.)

CHomP computes homology, not cohomology, and only works over the integers or finite prime fields. Therefore if any of these conditions fails, or if CHomP is not present, or if `algorithm` is set to 'no_chomp', go to plan B: if this complex has a `_homology` method – each simplicial complex has this, for example – then call that. Such a method implements specialized algorithms for the particular type of cell complex.

Otherwise, move on to plan C: compute the chain complex of this complex and compute its homology groups. To do this: over a field, just compute ranks and nullities, thus obtaining dimensions of the homology groups as vector spaces. Over the integers, compute Smith normal form of the boundary matrices defining the chain complex according to the value of `algorithm` . If `algorithm` is 'auto' or 'no_chomp', then for each relatively small matrix, use the standard Sage method, which calls the Pari package. For any large matrix, reduce it using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm: see *sage.homology.matrix_utils.dhsw_snf()* for details.

Finally, `algorithm` may also be 'pari' or 'dhsw', which forces the named algorithm to be used regardless of the size of the matrices and regardless of whether CHomP is available.

As of this writing, `'pari'` is the fastest standard option. The optional CHomP package may be better still.

EXAMPLES:

```
sage: P = delta_complexes.RealProjectivePlane()
sage: P.homology()
{0: 0, 1: C2, 2: 0}
sage: P.homology(reduced=False)
{0: Z, 1: C2, 2: 0}
sage: P.homology(base_ring=GF(2))
{0: Vector space of dimension 0 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: S7 = delta_complexes.Sphere(7)
sage: S7.homology(7)
Z
sage: cubical_complexes.KleinBottle().homology(1, base_ring=GF(2))
Vector space of dimension 2 over Finite Field of size 2
```

If CHomP is installed, Sage can compute generators of homology groups:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: S2.homology(dim=2, generators=True, base_ring=GF(2))  # optional - CHomP
(Vector space of dimension 1 over Finite Field of size 2, [(0, 1, 2) + (0, 1,␣
↪3) + (0, 2, 3) + (1, 2, 3)])
```

When generators are computed, Sage returns a pair for each dimension: the group and the list of generators. For simplicial complexes, each generator is represented as a linear combination of simplices, as above, and for cubical complexes, each generator is a linear combination of cubes:

```
sage: S2_cub = cubical_complexes.Sphere(2)
sage: S2_cub.homology(dim=2, generators=True)  # optional - CHomP
(Z, [-[[0,1] x [0,1] x [0,0]] + [[0,1] x [0,1] x [1,1]] - [[0,0] x [0,1] x␣
↪[0,1]] - [[0,1] x [1,1] x [0,1]] + [[0,1] x [0,0] x [0,1]] + [[1,1] x [0,1]␣
↪x [0,1]]])
```

homology_with_basis (*base_ring=Rational Field*, *cohomology=False*)

Return the unreduced homology of this complex with coefficients in `base_ring` with a chosen basis.

This is implemented for simplicial, cubical, and Δ-complexes, not for arbitrary generic cell complexes.

INPUT:

- `base_ring` – coefficient ring (optional, default `QQ`); must be a field
- `cohomology` – boolean (optional, default `False`); if `True`, return cohomology instead of homology

Homology basis elements are named 'h_{dim,i}' where i ranges between 0 and $r-1$, if r is the rank of the homology group. Cohomology basis elements are denoted $h^{dim,i}$ instead.

See also:

If `cohomology` is `True`, this returns the cohomology as a graded module. For the ring structure, use *cohomology_ring()*.

EXAMPLES:

```
sage: K = simplicial_complexes.KleinBottle()
sage: H = K.homology_with_basis(QQ); H
```

```
Homology module of Minimal triangulation of the Klein bottle
 over Rational Field
sage: sorted(H.basis(), key=str)
[h_{0,0}, h_{1,0}]
sage: H = K.homology_with_basis(GF(2)); H
Homology module of Minimal triangulation of the Klein bottle
 over Finite Field of size 2
sage: sorted(H.basis(), key=str)
[h_{0,0}, h_{1,0}, h_{1,1}, h_{2,0}]
```

The homology is constructed as a graded object, so for example, you can ask for the basis in a single degree:

```
sage: H.basis(1)
Finite family {(1, 0): h_{1,0}, (1, 1): h_{1,1}}
sage: S3 = delta_complexes.Sphere(3)
sage: H = S3.homology_with_basis(QQ, cohomology=True)
sage: list(H.basis(3))
[h^{3,0}]
```

is_acyclic (*base_ring=Integer Ring*)

True if the reduced homology with coefficients in `base_ring` of this cell complex is zero.

INPUT:

- `base_ring` – optional, default `ZZ` . Compute homology with coefficients in this ring.

EXAMPLES:

```
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: RP2.is_acyclic()
False
sage: RP2.is_acyclic(QQ)
True
```

This first computes the Euler characteristic: if it is not 1, the complex cannot be acyclic. So this should return `False` reasonably quickly on complexes with Euler characteristic not equal to 1:

```
sage: K = cubical_complexes.KleinBottle()
sage: C = cubical_complexes.Cube(2)
sage: P = K.product(C)
sage: P
Cubical complex with 168 vertices and 1512 cubes
sage: P.euler_characteristic()
0
sage: P.is_acyclic()
False
```

join (*right*)

The join of this cell complex with another one.

Parameters **right** – the other cell complex (the right-hand factor)

Joins are not implemented for general cell complexes. They may be implemented in some derived classes (like simplicial complexes).

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex(); B = GenericCellComplex()
```

```
sage: A.join(B)
Traceback (most recent call last):
...
NotImplementedError: <abstract method join at ...>
```

n_cells (*n*, *subcomplex=None*)

List of cells of dimension `n` of this cell complex. If the optional argument `subcomplex` is present, then return the `n` -dimensional faces which are *not* in the subcomplex.

Parameters

- **n** (*non-negative integer*) – the dimension
- **subcomplex** (optional, default `None`) – a subcomplex of this cell complex. Return the cells which are not in this subcomplex.

EXAMPLES:

```
sage: delta_complexes.Torus().n_cells(1)
[(0, 0), (0, 0), (0, 0)]
sage: cubical_complexes.Cube(1).n_cells(0)
[[1,1], [0,0]]
```

n_chains (*n*, *base_ring=Integer Ring*, *cochains=False*)

Return the free module of chains in degree `n` over `base_ring` .

INPUT:

- `n` – integer
- `base_ring` – ring (optional, default **Z**)
- `cochains` – boolean (optional, default `False`); if `True` , return cochains instead

The only difference between chains and cochains is notation. In a simplicial complex, for example, a simplex `(0,1,2)` is written as "(0,1,2)" in the group of chains but as "chi_(0,1,2)" in the group of cochains.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: S2.n_chains(1, QQ)
Free module generated by {(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}␣
↪over Rational Field
sage: list(simplicial_complexes.Sphere(2).n_chains(1, QQ, cochains=False).
↪basis())
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: list(simplicial_complexes.Sphere(2).n_chains(1, QQ, cochains=True).
↪basis())
[\chi_(0, 1), \chi_(0, 2), \chi_(0, 3), \chi_(1, 2), \chi_(1, 3), \chi_(2, 3)]
```

n_skeleton (*n*)

The n-skeleton of this cell complex: the cell complex obtained by discarding all of the simplices in dimensions larger than n.

Parameters **n** – non-negative integer

This is not implemented for general cell complexes.

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.n_skeleton(3)
Traceback (most recent call last):
...
NotImplementedError: <abstract method n_skeleton at ...>
```

product (*right*, *rename_vertices=True*)

The (Cartesian) product of this cell complex with another one.

Products are not implemented for general cell complexes. They may be implemented in some derived classes (like simplicial complexes).

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex(); B = GenericCellComplex()
sage: A.product(B)
Traceback (most recent call last):
...
NotImplementedError: <abstract method product at ...>
```

wedge (*right*)

The wedge (one-point union) of this cell complex with another one.

Parameters **right** – the other cell complex (the right-hand factor)

Wedges are not implemented for general cell complexes.

EXAMPLES:

```
sage: from sage.homology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex(); B = GenericCellComplex()
sage: A.wedge(B)
Traceback (most recent call last):
...
NotImplementedError: <abstract method wedge at ...>
```

CHAPTER

THIRTEEN

KOSZUL COMPLEXES

class sage.homology.koszul_complex. **KoszulComplex** (*R*, *elements*)

Bases: *sage.homology.chain_complex.ChainComplex_class* , sage.structure.unique_representation.UniqueRepresentation

A Koszul complex.

Let R be a ring and consider $x_1, x_2, \ldots, x_n \in R$. The *Koszul complex* $K_*(x_1, \ldots, x_n)$ is given by defining a chain complex structure on the exterior algebra $\bigwedge^n R$ with the basis $e_{i_1} \wedge \cdots \wedge e_{i_a}$. The differential is given by

$$\partial(e_{i_1} \wedge \cdots \wedge e_{i_a}) = \sum_{r=1}^{a} (-1)^{r-1} x_{i_r} e_{i_1} \wedge \cdots \wedge \hat{e}_{i_r} \wedge \cdots \wedge e_{i_a},$$

where $\hat{e}_{i_r}$ denotes the omitted factor.

Alternatively we can describe the Koszul complex by considering the basic complex K_{x_i}

$$0 \to R \xrightarrow{x_i} R \to 0.$$

Then the Koszul complex is given by $K_*(x_1, \ldots, x_n) = \bigotimes_i K_{x_i}$.

INPUT:

- R – the base ring
- elements – a tuple of elements of R

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: K = KoszulComplex(R, [x,y])
sage: ascii_art(K)
                        [-y]
            [x y]       [ x]
 0 <-- C_0 <------ C_1 <----- C_2 <-- 0
sage: K = KoszulComplex(R, [x,y,z])
sage: ascii_art(K)
                          [-y -z  0]       [ z]
                          [ x  0 -z]       [-y]
            [x y z]       [ 0  x  y]       [ x]
 0 <-- C_0 <-------- C_1 <----------- C_2 <----- C_3 <-- 0
sage: K = KoszulComplex(R, [x+y*z,x+y-z])
sage: ascii_art(K)
                                        [-x - y + z]
            [  y*z + x x + y - z]       [   y*z + x]
 0 <-- C_0 <---------------------- C_1 <-------------- C_2 <-- 0
```

REFERENCES:

- Wikipedia article Koszul_complex

CHAPTER

FOURTEEN

HOCHSCHILD COMPLEXES

class sage.homology.hochschild_complex. **HochschildComplex** (*A*, *M*)

Bases: sage.structure.unique_representation.UniqueRepresentation , sage.structure.category_object.CategoryObject

The Hochschild complex.

Let A be an algebra over a commutative ring R such that A a projective R-module, and M an A-bimodule. The *Hochschild complex* is the chain complex given by

$$C_n(A, M) := M \otimes A^{\otimes n}$$

with the boundary operators given as follows. For fixed n, define the face maps

$$f_{n,i}(m \otimes a_1 \otimes \cdots \otimes a_n) = \begin{cases} ma_1 \otimes \cdots \otimes a_n & \text{if } i = 0, \\ a_n m \otimes a_1 \otimes \cdots \otimes a_{n-1} & \text{if } i = n, \\ m \otimes a_1 \otimes \cdots \otimes a_i a_{i+1} \otimes \cdots \otimes a_n & \text{otherwise.} \end{cases}$$

We define the boundary operators as

$$d_n = \sum_{i=0}^{n} (-1)^i f_{n,i}.$$

The *Hochschild homology* of A is the homology of this complex. Alternatively, the Hochschild homology can be described by $HH_n(A, M) = \operatorname{Tor}_n^{A^e}(A, M)$, where $A^e = A \otimes A^o$ (A^o is the opposite algebra of A) is the enveloping algebra of A.

Hochschild cohomology is the homology of the dual complex and can be described by $HH^n(A, M) = \operatorname{Ext}_{A^e}^n(A, M)$.

Another perspective on Hochschild homology is that $f_{n,i}$ make the family $C_n(A, M)$ a simplicial object in the category of R-modules, and the degeneracy maps are

$$s_i(a_0 \otimes \cdots \otimes a_n) = a_0 \otimes \cdots \otimes a_i \otimes 1 \otimes a_{i+1} \otimes \cdots \otimes a_n$$

The Hochschild homology can also be constructed as the homology of this simplicial module.

REFERENCES:

- Wikipedia article Hochschild_homology
- https://ncatlab.org/nlab/show/Hochschild+cohomology
- [Red2001]

algebra ( )

Return the defining algebra of `self` .

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.algebra()
Symmetric group algebra of order 3 over Rational Field
```

boundary (*d*)

Return the boundary operator in degree `d` .

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: d1 = H.boundary(1)
sage: z = d1.domain().an_element(); z
2*1 # 1 + 2*1 # x + 3*1 # y
sage: d1(z)
0
sage: d1.matrix()
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  2  0  0 -2  0  0  0  0  0  0]

sage: s = SymmetricFunctions(QQ).s()
sage: H = s.hochschild_complex(s)
sage: d1 = H.boundary(1)
sage: x = d1.domain().an_element(); x
2*s[] # s[] + 2*s[] # s[1] + 3*s[] # s[2]
sage: d1(x)
0
sage: y = tensor([s.an_element(), s.an_element()])
sage: d1(y)
0
sage: z = tensor([s[2,1] + s[3], s.an_element()])
sage: d1(z)
0
```

TESTS:

```
sage: def test_complex(H, n):
....:     phi = H.boundary(n)
....:     psi = H.boundary(n+1)
....:     comp = phi * psi
....:     zero = H.free_module(n-1).zero()
....:     return all(comp(b) == zero for b in H.free_module(n+1).basis())

sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: H = SGA.hochschild_complex(SGA)
sage: test_complex(H, 1)
True
sage: test_complex(H, 2)
True
sage: test_complex(H, 3) # long time
True

sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
```

```
sage: test_complex(H, 1)
True
sage: test_complex(H, 2)
True
sage: test_complex(H, 3)
True
```

coboundary (*d*)

Return the coboundary morphism of degree `d` .

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: del1 = H.coboundary(1)
sage: z = del1.domain().an_element(); z
2 + 2*x + 3*y
sage: del1(z)
0
sage: del1.matrix()
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  2]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0 -2]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
```

TESTS:

```
sage: def test_complex(H, n):
....:     phi = H.coboundary(n)
....:     psi = H.coboundary(n+1)
....:     comp = psi * phi
....:     zero = H.free_module(n+1).zero()
....:     return all(comp(b) == zero for b in H.free_module(n-1).basis())

sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: H = SGA.hochschild_complex(SGA)
sage: test_complex(H, 1)
True
sage: test_complex(H, 2)
True

sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: test_complex(H, 1)
True
sage: test_complex(H, 2)
```

```
True
sage: test_complex(H, 3)
True
```

coefficients ( )

Return the coefficients of `self` .

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.coefficients()
Trivial representation of Standard permutations of 3 over Rational Field
```

cohomology (*d*)

Return the `d` -th cohomology group.

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: H.cohomology(0)
Vector space of dimension 3 over Rational Field
sage: H.cohomology(1)
Vector space of dimension 4 over Rational Field
sage: H.cohomology(2)
Vector space of dimension 6 over Rational Field

sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.cohomology(0)
Vector space of dimension 1 over Rational Field
sage: H.cohomology(1)
Vector space of dimension 0 over Rational Field
sage: H.cohomology(2)
Vector space of dimension 0 over Rational Field
```

When working over general rings (except $\mathbf{Z}$) and we can construct a unitriangular basis for the image quotient, we fallback to a slower implementation using (combinatorial) free modules:

```
sage: R.<x,y> = QQ[]
sage: SGA = SymmetricGroupAlgebra(R, 2)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.cohomology(1)
Free module generated by {} over Multivariate Polynomial Ring in x, y over␣
↪Rational Field
```

free_module (*d*)

Return the free module in degree `d` .

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.free_module(0)
```

```
Trivial representation of Standard permutations of 3 over Rational Field
sage: H.free_module(1)
Trivial representation of Standard permutations of 3 over Rational Field
 # Symmetric group algebra of order 3 over Rational Field
sage: H.free_module(2)
Trivial representation of Standard permutations of 3 over Rational Field
 # Symmetric group algebra of order 3 over Rational Field
 # Symmetric group algebra of order 3 over Rational Field
```

homology (*d*)

Return the d -th homology group.

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: H.homology(0)
Vector space of dimension 3 over Rational Field
sage: H.homology(1)
Vector space of dimension 4 over Rational Field
sage: H.homology(2)
Vector space of dimension 6 over Rational Field

sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.homology(0)
Vector space of dimension 1 over Rational Field
sage: H.homology(1)
Vector space of dimension 0 over Rational Field
sage: H.homology(2)
Vector space of dimension 0 over Rational Field
```

When working over general rings (except $\mathbf{Z}$) and we can construct a unitriangular basis for the image quotient, we fallback to a slower implementation using (combinatorial) free modules:

```
sage: R.<x,y> = QQ[]
sage: SGA = SymmetricGroupAlgebra(R, 2)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.homology(1)
Free module generated by {} over Multivariate Polynomial Ring in x, y over␣
 ↪Rational Field
```

trivial_module ( )

Return the trivial module of self .

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: H.trivial_module()
Free module generated by {} over Rational Field
```

CHAPTER

FIFTEEN

HOMOLOGY GROUPS

This module defines a *HomologyGroup()* class which is an abelian group that prints itself in a way that is suitable for homology groups.

sage.homology.homology_group. **HomologyGroup** (*n*, *base_ring*, *invfac=None*)

Abelian group on n generators which represents a homology group in a fixed degree.

INPUT:

- n – integer; the number of generators
- base_ring – ring; the base ring over which the homology is computed
- inv_fac – list of integers; the invariant factors – ignored if the base ring is a field

OUTPUT:

A class that can represent the homology group in a fixed homological degree.

EXAMPLES:

```
sage: from sage.homology.homology_group import HomologyGroup
sage: G = AbelianGroup(5, [5,5,7,8,9]); G
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
sage: H = HomologyGroup(5, ZZ, [5,5,7,8,9]); H
C5 x C5 x C7 x C8 x C9
sage: AbelianGroup(4)
Multiplicative Abelian group isomorphic to Z x Z x Z x Z
sage: HomologyGroup(4, ZZ)
Z x Z x Z x Z
sage: HomologyGroup(100, ZZ)
Z^100
```

class sage.homology.homology_group. **HomologyGroup_class** (*n*, *invfac*)

Bases: sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_fixed_ge

Discrete Abelian group on n generators. This class inherits from AdditiveAbelianGroup_fixed_gens ; see sage.groups.additive_abelian.additive_abelian_group for more documentation. The main difference between the classes is in the print representation.

EXAMPLES:

```
sage: from sage.homology.homology_group import HomologyGroup
sage: G = AbelianGroup(5, [5,5,7,8,9]); G
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
sage: H = HomologyGroup(5, ZZ, [5,5,7,8,9]); H
C5 x C5 x C7 x C8 x C9
sage: G == loads(dumps(G))
```

```
True
sage: AbelianGroup(4)
Multiplicative Abelian group isomorphic to Z x Z x Z x Z
sage: HomologyGroup(4, ZZ)
Z x Z x Z x Z
sage: HomologyGroup(100, ZZ)
Z^100
```

CHAPTER

SIXTEEN

HOMOLOGY AND COHOMOLOGY WITH A BASIS

This module provides homology and cohomology vector spaces suitable for computing cup products and cohomology operations.

REFERENCES:

- [GDR2003]
- [GDR1999]

AUTHORS:

- John H. Palmieri, Travis Scrimshaw (2015-09)

class sage.homology.homology_vector_space_with_basis. **CohomologyRing** (*base_ring*, *cell_complex*)

Bases: *sage.homology.homology_vector_space_with_basis.HomologyVectorSpaceWithBasis*

The cohomology ring.

Note: This is not intended to be created directly by the user, but instead via the *cohomology ring* of a *cell complex*.

INPUT:

- base_ring – must be a field
- cell_complex – the cell complex whose homology we are computing

EXAMPLES:

```
sage: CP2 = simplicial_complexes.ComplexProjectivePlane()
sage: H = CP2.cohomology_ring(QQ)
sage: H.basis(2)
Finite family {(2, 0): h^{2,0}}
sage: x = H.basis(2)[2,0]
```

The product structure is the cup product:

```
sage: x.cup_product(x)
-h^{4,0}
sage: x * x
-h^{4,0}
```

There are mod 2 cohomology operations defined, also:

```
sage: Hmod2 = CP2.cohomology_ring(GF(2))
sage: y = Hmod2.basis(2)[2,0]
sage: y.Sq(2)
h^{4,0}
```

class **Element** (*M*, *x*)

Bases: *sage.homology.homology_vector_space_with_basis.HomologyVectorSpaceWithBasis.El*

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

Sq (*i*)

Return the result of applying Sq^i to this element.

INPUT:

- `i` – nonnegative integer

Warning: This is only implemented for simplicial complexes.

This cohomology operation is only defined in characteristic 2.

Algorithm: see González-Díaz and Réal [GDR1999], Corollary 3.2.

EXAMPLES:

```
sage: RP2 = simplicial_complexes.RealProjectiveSpace(2)
sage: x = RP2.cohomology_ring(GF(2)).basis()[1,0]
sage: x.Sq(1)
h^{2,0}

sage: K = RP2.suspension()
sage: K.set_immutable()
sage: y = K.cohomology_ring(GF(2)).basis()[2,0]
sage: y.Sq(1)
h^{3,0}

sage: RP4 = simplicial_complexes.RealProjectiveSpace(4)
sage: H = RP4.cohomology_ring(GF(2))
sage: x = H.basis()[1,0]
sage: y = H.basis()[2,0]
sage: z = H.basis()[3,0]
sage: x.Sq(1) == y
True
sage: z.Sq(1)  # long time
h^{4,0}
```

TESTS:

```
sage: T = cubical_complexes.Torus()
sage: x = T.cohomology_ring(GF(2)).basis()[1,0]
sage: x.Sq(1)
Traceback (most recent call last):
...
NotImplementedError: Steenrod squares are only implemented for simplicial␣
↪complexes
sage: S2 = simplicial_complexes.Sphere(2)
sage: x = S2.cohomology_ring(GF(7)).basis()[2,0]
sage: x.Sq(1)
Traceback (most recent call last):
...
ValueError: Steenrod squares are only defined in characteristic 2
```

cup_product (*other*)

Return the cup product of this element and `other` .

Algorithm: see González-Díaz and Réal [GDR2003], p. 88. Given two cohomology classes, lift them to cocycle representatives via the chain contraction for this complex, using *to_cycle()* . In the sum of their dimensions, look at all of the homology classes γ: lift each of those to a cycle representative, apply the Alexander-Whitney diagonal map to each cell in the cycle, evaluate the two cocycles on these factors, and multiply. The result is the value of the cup product cocycle on this homology class. After this has been done for all homology classes, since homology and cohomology are dual, one can tell which cohomology class corresponds to the cup product.

See also:

CohomologyRing.product_on_basis()

EXAMPLES:

```
sage: RP3 = simplicial_complexes.RealProjectiveSpace(3)
sage: H = RP3.cohomology_ring(GF(2))
sage: c = H.basis()[1,0]
sage: c.cup_product(c)
h^{2,0}
sage: c * c * c
h^{3,0}
```

We can also take powers:

```
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: a = RP2.cohomology_ring(GF(2)).basis()[1,0]
sage: a**0
h^{0,0}
sage: a**1
h^{1,0}
sage: a**2
h^{2,0}
sage: a**3
0
```

A non-connected example:

```
sage: K = cubical_complexes.Torus().disjoint_union(cubical_complexes.
↪Sphere(2))
sage: a,b = K.cohomology_ring(QQ).basis(2)
sage: a**0
h^{0,0} + h^{0,1}
```

CohomologyRing. **one** ( )

The multiplicative identity element.

EXAMPLES:

```
sage: H = simplicial_complexes.Torus().cohomology_ring(QQ)
sage: H.one()
h^{0,0}
sage: all(H.one() * x == x == x * H.one() for x in H.basis())
True
```

CohomologyRing. **product_on_basis** (*li*, *ri*)

The cup product of the basis elements indexed by `li` and `ri` in this cohomology ring.

INPUT:

- `li` , `ri` – index of a cohomology class

See also:

CohomologyRing.Element.cup_product() – the documentation for this method describes the algorithm.

EXAMPLES:

```
sage: RP3 = simplicial_complexes.RealProjectiveSpace(3)
sage: H = RP3.cohomology_ring(GF(2))
sage: c = H.basis()[1,0]
sage: c.cup_product(c).cup_product(c) # indirect doctest
h^{3,0}

sage: T = simplicial_complexes.Torus()
sage: x,y = T.cohomology_ring(QQ).basis(1)
sage: x.cup_product(y)
-h^{2,0}
sage: x.cup_product(x)
0

sage: one = T.cohomology_ring(QQ).basis()[0,0]
sage: x.cup_product(one)
h^{1,0}
sage: one.cup_product(y) == y
True
sage: one.cup_product(one)
h^{0,0}
sage: x.cup_product(y) + y.cup_product(x)
0
```

This also works with cubical complexes:

```
sage: T = cubical_complexes.Torus()
sage: x,y = T.cohomology_ring(QQ).basis(1)
sage: x.cup_product(y)
-h^{2,0}
sage: x.cup_product(x)
0
```

and Δ-complexes:

```
sage: T_d = delta_complexes.Torus()
sage: a,b = T_d.cohomology_ring(QQ).basis(1)
sage: a.cup_product(b)
h^{2,0}
sage: b.cup_product(a)
-h^{2,0}
sage: RP2 = delta_complexes.RealProjectivePlane()
sage: w = RP2.cohomology_ring(GF(2)).basis()[1,0]
sage: w.cup_product(w)
h^{2,0}
```

A non-connected example:

```
sage: K = cubical_complexes.Torus().disjoint_union(cubical_complexes.Torus())
sage: a,b,c,d = K.cohomology_ring(QQ).basis(1)
sage: x,y = K.cohomology_ring(QQ).basis(0)
sage: a.cup_product(x) == a
True
sage: a.cup_product(y)
0
```

class sage.homology.homology_vector_space_with_basis. **HomologyVectorSpaceWithBasis** (*base_ring*, *cell_complex*, *cohomology=False*, *category=None*)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Homology (or cohomology) vector space.

This provides enough structure to allow the computation of cup products and cohomology operations. See the class *CohomologyRing* (which derives from this) for examples.

It also requires field coefficients (hence the "VectorSpace" in the name of the class).

Note: This is not intended to be created directly by the user, but instead via the methods *homology_with_basis()* and *cohomology_ring()* for the class of *cell complexes*.

INPUT:

- base_ring – must be a field
- cell_complex – the cell complex whose homology we are computing
- cohomology – (default: False) if True , return the cohomology as a module
- category – (optional) a subcategory of modules with basis

EXAMPLES:

Homology classes are denoted by h_{d,i} where d is the degree of the homology class and i is their index in the list of basis elements in that degree. Cohomology classes are denoted h^{1,0} :

```
sage: RP2 = cubical_complexes.RealProjectivePlane()
sage: RP2.homology_with_basis(GF(2))
```

```
Homology module of Cubical complex with 21 vertices and 81 cubes
 over Finite Field of size 2
sage: RP2.cohomology_ring(GF(2))
Cohomology ring of Cubical complex with 21 vertices and 81 cubes
 over Finite Field of size 2
sage: simplicial_complexes.Torus().homology_with_basis(QQ)
Homology module of Minimal triangulation of the torus
 over Rational Field
```

To access a basis element, use its degree and index (0 or 1 in the 1st cohomology group of a torus):

```
sage: H = simplicial_complexes.Torus().cohomology_ring(QQ)
sage: H.basis(1)
Finite family {(1, 0): h^{1,0}, (1, 1): h^{1,1}}
sage: x = H.basis()[1,0]; x
h^{1,0}
sage: y = H.basis()[1,1]; y
h^{1,1}
sage: 2*x-3*y
2*h^{1,0} - 3*h^{1,1}
```

You can compute cup products of cohomology classes:

```
sage: x.cup_product(y)
-h^{2,0}
sage: y.cup_product(x)
h^{2,0}
sage: x.cup_product(x)
0
```

This works with simplicial, cubical, and Δ-complexes:

```
sage: Klein_c = cubical_complexes.KleinBottle()
sage: H = Klein_c.cohomology_ring(GF(2))
sage: x,y = H.basis(1)
sage: x.cup_product(x)
h^{2,0}
sage: x.cup_product(y)
0
sage: y.cup_product(y)
h^{2,0}

sage: Klein_d = delta_complexes.KleinBottle()
sage: H = Klein_d.cohomology_ring(GF(2))
sage: u,v = H.basis(1)
sage: u.cup_product(u)
h^{2,0}
sage: u.cup_product(v)
0
sage: v.cup_product(v)
h^{2,0}
```

The basis elements in the simplicial complex case have been chosen differently; apply the change of basis $x \mapsto a + b$, $y \mapsto b$ to see the same product structure.

```
sage: Klein_s = simplicial_complexes.KleinBottle()
sage: H = Klein_s.cohomology_ring(GF(2))
sage: a,b = H.basis(1)
```

```
sage: a.cup_product(a)
0
sage: a.cup_product(b)
h^{2,0}
sage: (a+b).cup_product(a+b)
h^{2,0}
sage: b.cup_product(b)
h^{2,0}
```

class **Element** (*M*, *x*)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's __call__() method.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

to_cycle ( )

(Co)cycle representative of this homogeneous (co)homology class.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: H = S2.homology_with_basis(QQ)
sage: h20 = H.basis()[2,0]; h20
h_{2,0}
sage: h20.to_cycle()
-(0, 1, 2) + (0, 1, 3) - (0, 2, 3) + (1, 2, 3)
```

Chains are written as linear combinations of simplices σ. Cochains are written as linear combinations of characteristic functions χ_σ for those simplices:

```
sage: S2.cohomology_ring(QQ).basis()[2,0].to_cycle()
\chi_(1, 2, 3)
sage: S2.cohomology_ring(QQ).basis()[0,0].to_cycle()
\chi_(0,) + \chi_(1,) + \chi_(2,) + \chi_(3,)
```

HomologyVectorSpaceWithBasis. **basis** (*d=None*)

Return (the degree d homogeneous component of) the basis of this graded vector space.

INPUT:

- d – (optional) the degree

EXAMPLES:

```
sage: RP2 = simplicial_complexes.ProjectivePlane()
sage: H = RP2.homology_with_basis(QQ)
sage: H.basis()
Finite family {(0, 0): h_{0,0}}
sage: H.basis(0)
Finite family {(0, 0): h_{0,0}}
sage: H.basis(1)
```

```
Finite family {}
sage: H.basis(2)
Finite family {}
```

HomologyVectorSpaceWithBasis. **complex** ( )

The cell complex whose homology is being computed.

EXAMPLES:

```
sage: H = simplicial_complexes.Simplex(2).homology_with_basis(QQ)
sage: H.complex()
The 2-simplex
```

HomologyVectorSpaceWithBasis. **contraction** ( )

The chain contraction associated to this homology computation.

That is, to work with chain representatives of homology classes, we need the chain complex C associated to the cell complex, the chain complex H of its homology (with trivial differential), chain maps $\pi : C \to H$ and $\iota : H \to C$, and a chain contraction ϕ giving a chain homotopy between 1_C and $\iota \circ \pi$.

OUTPUT: ϕ

See *ChainContraction* for information about chain contractions, and see *algebraic_topological_model()* for the construction of this particular chain contraction ϕ.

EXAMPLES:

```
sage: H = simplicial_complexes.Simplex(2).homology_with_basis(QQ)
sage: H.contraction()
Chain homotopy between:
  Chain complex endomorphism of Chain complex with at most 3 nonzero terms␣
↪over Rational Field
  and Chain complex endomorphism of Chain complex with at most 3 nonzero␣
↪terms over Rational Field
```

From the chain contraction, one can also recover the maps π and ι:

```
sage: phi = H.contraction()
sage: phi.pi()
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Rational Field
  To: Chain complex with at most 1 nonzero terms over Rational Field
sage: phi.iota()
Chain complex morphism:
  From: Chain complex with at most 1 nonzero terms over Rational Field
  To: Chain complex with at most 3 nonzero terms over Rational Field
```

HomologyVectorSpaceWithBasis. **degree_on_basis** (*i*)

Return the degree of the basis element indexed by `i` .

EXAMPLES:

```
sage: H = simplicial_complexes.Torus().homology_with_basis(GF(7))
sage: H.degree_on_basis((2,0))
2
```

sage.homology.homology_vector_space_with_basis. **sum_indices** (*k*, *i_k_plus_one*, *S_k_plus_one*)

This is a recursive function for computing the indices for the nested sums in González-Díaz and Réal [GDR1999], Corollary 3.2.

In the paper, given indices i_n, i_{n-1}, ..., i_{k+1}, given k, and given $S(k+1)$, the number $S(k)$ is defined to be

$$S(k) = -S(k+1) + floor(k/2) + floor((k+1)/2) + i_{k+1},$$

and i_k ranges from $S(k)$ to $i_{k+1} - 1$. There are two special cases: if $k = 0$, then $i_0 = S(0)$. Also, the initial case of $S(k)$ is $S(n)$, which is set in the method `Sq()` before calling this function. For this function, given k, i_{k+1}, and $S(k+1)$, return a list consisting of the allowable possible indices $[i_k, i_{k-1}, ..., i_1, i_0]$ given by the above formula.

INPUT:

- `k` – non-negative integer
- `i_k_plus_one` – the positive integer i_{k+1}
- `S_k_plus_one` – the integer $S(k+1)$

EXAMPLES:

```
sage: from sage.homology.homology_vector_space_with_basis import sum_indices
sage: sum_indices(1, 3, 3)
[[1, 0], [2, 1]]
sage: sum_indices(0, 4, 2)
[[2]]
```

CHAPTER

SEVENTEEN

ALGEBRAIC TOPOLOGICAL MODEL FOR A CELL COMPLEX

This file contains two functions, *algebraic_topological_model()* and *algebraic_topological_model_delta_complex()*. The second works more generally: for all simplicial, cubical, and Δ-complexes. The first only works for simplicial and cubical complexes, but it is faster in those case.

AUTHORS:

- John H. Palmieri (2015-09)

sage.homology.algebraic_topological_model. **algebraic_topological_model** (*K*, *base_ring=None*)

Algebraic topological model for cell complex `K` with coefficients in the field `base_ring`.

INPUT:

- `K` – either a simplicial complex or a cubical complex
- `base_ring` – coefficient ring; must be a field

OUTPUT: a pair `(phi,M)` consisting of

- chain contraction `phi`
- chain complex M

This construction appears in a paper by Pilarczyk and Réal [PR2015]. Given a cell complex K and a field F, there is a chain complex C associated to K with coefficients in F. The *algebraic topological model* for K is a chain complex M with trivial differential, along with chain maps $\pi : C \to M$ and $\iota : M \to C$ such that

- $\pi\iota = 1_M$, and
- there is a chain homotopy ϕ between 1_C and $\iota\pi$.

In particular, π and ι induce isomorphisms on homology, and since M has trivial differential, it is its own homology, and thus also the homology of C. Thus ι lifts homology classes to their cycle representatives.

The chain homotopy ϕ satisfies some additional properties, making it a *chain contraction*:

- $\phi\phi = 0$,
- $\pi\phi = 0$,
- $\phi\iota = 0$.

Given an algebraic topological model for K, it is then easy to compute cup products and cohomology operations on the cohomology of K, as described in [GDR2003] and [PR2015].

Implementation details: the cell complex K must have an *n_cells()* method from which we can extract a list of cells in each dimension. Combining the lists in increasing order of dimension then defines a filtration of the complex: a list of cells in which the boundary of each cell consists of cells earlier in the list. This is required

by Pilarczyk and Réal's algorithm. There must also be a *chain_complex()* method, to construct the chain complex C associated to this chain complex.

In particular, this works for simplicial complexes and cubical complexes. It doesn't work for Δ-complexes, though: the list of their n-cells has the wrong format.

Note that from the chain contraction `phi` , one can recover the chain maps π and ι via `phi.pi()` and `phi.iota()` . Then one can recover C and M from, for example, `phi.pi().domain()` and `phi.pi().codomain()` , respectively.

EXAMPLES:

```
sage: from sage.homology.algebraic_topological_model import algebraic_topological_
↪model
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: phi, M = algebraic_topological_model(RP2, GF(2))
sage: M.homology()
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: T = cubical_complexes.Torus()
sage: phi, M = algebraic_topological_model(T, QQ)
sage: M.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

If you want to work with cohomology rather than homology, just dualize the outputs of this function:

```
sage: M.dual().homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
sage: M.dual().degree_of_differential()
1
sage: phi.dual()
Chain homotopy between:
  Chain complex endomorphism of Chain complex with at most 3 nonzero terms over␣
↪Rational Field
  and Chain complex morphism:
    From: Chain complex with at most 3 nonzero terms over Rational Field
    To:   Chain complex with at most 3 nonzero terms over Rational Field
```

In degree 0, the inclusion of the homology M into the chain complex C sends the homology generator to a single vertex:

```
sage: K = simplicial_complexes.Simplex(2)
sage: phi, M = algebraic_topological_model(K, QQ)
sage: phi.iota().in_degree(0)
[0]
[0]
[1]
```

In cohomology, though, one needs the dual of every degree 0 cell to detect the degree 0 cohomology generator:

```
sage: phi.dual().iota().in_degree(0)
[1]
[1]
[1]
```

TESTS:

```
sage: T = cubical_complexes.Torus()
sage: C = T.chain_complex()
sage: H, M = T.algebraic_topological_model()
sage: C.differential(1) * H.iota().in_degree(1).column(0) == 0
True
sage: C.differential(1) * H.iota().in_degree(1).column(1) == 0
True
sage: coC = T.chain_complex(cochain=True)
sage: coC.differential(1) * H.dual().iota().in_degree(1).column(0) == 0
True
sage: coC.differential(1) * H.dual().iota().in_degree(1).column(1) == 0
True
```

sage.homology.algebraic_topological_model. **algebraic_topological_model_delta_complex** (*K*, *base_ri*

Algebraic topological model for cell complex `K` with coefficients in the field `base_ring`.

This has the same basic functionality as *algebraic_topological_model()*, but it also works for Δ-complexes. For simplicial and cubical complexes it is somewhat slower, though.

INPUT:

- `K` – a simplicial complex, a cubical complex, or a Δ-complex
- `base_ring` – coefficient ring; must be a field

OUTPUT: a pair `(phi,M)` consisting of

- chain contraction `phi`
- chain complex M

See *algebraic_topological_model()* for the main documentation. The difference in implementation between the two: this uses matrix and vector algebra. The other function does more of the computations "by hand" and uses cells (given as simplices or cubes) to index various dictionaries. Since the cells in Δ-complexes are not as nice, the other function does not work for them, while this function relies almost entirely on the structure of the associated chain complex.

EXAMPLES:

```
sage: from sage.homology.algebraic_topological_model import algebraic_topological_
↪model_delta_complex as AT_model
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: phi, M = AT_model(RP2, GF(2))
sage: M.homology()
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: T = delta_complexes.Torus()
sage: phi, M = AT_model(T, QQ)
sage: M.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

If you want to work with cohomology rather than homology, just dualize the outputs of this function:

```
sage: M.dual().homology()
{0: Vector space of dimension 1 over Rational Field,
```

```
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
sage: M.dual().degree_of_differential()
1
sage: phi.dual()
Chain homotopy between:
  Chain complex endomorphism of Chain complex with at most 3 nonzero terms over␣
↪Rational Field
  and Chain complex morphism:
    From: Chain complex with at most 3 nonzero terms over Rational Field
    To:   Chain complex with at most 3 nonzero terms over Rational Field
```

In degree 0, the inclusion of the homology M into the chain complex C sends the homology generator to a single vertex:

```
sage: K = delta_complexes.Simplex(2)
sage: phi, M = AT_model(K, QQ)
sage: phi.iota().in_degree(0)
[0]
[0]
[1]
```

In cohomology, though, one needs the dual of every degree 0 cell to detect the degree 0 cohomology generator:

```
sage: phi.dual().iota().in_degree(0)
[1]
[1]
[1]
```

TESTS:

```
sage: T = cubical_complexes.Torus()
sage: C = T.chain_complex()
sage: H, M = AT_model(T, QQ)
sage: C.differential(1) * H.iota().in_degree(1).column(0) == 0
True
sage: C.differential(1) * H.iota().in_degree(1).column(1) == 0
True
sage: coC = T.chain_complex(cochain=True)
sage: coC.differential(1) * H.dual().iota().in_degree(1).column(0) == 0
True
sage: coC.differential(1) * H.dual().iota().in_degree(1).column(1) == 0
True
```

CHAPTER

EIGHTEEN

INDUCED MORPHISMS ON HOMOLOGY

This module implements morphisms on homology induced by morphisms of simplicial complexes. It requires working with field coefficients.

See *InducedHomologyMorphism* for documentation.

AUTHORS:

- John H. Palmieri (2015.09)

class sage.homology.homology_morphism. **InducedHomologyMorphism** (*map*, *base_ring=None*, *cohomology=False*)

Bases: sage.categories.morphism.Morphism

An element of this class is a morphism of (co)homology groups induced by a map of simplicial complexes. It requires working with field coefficients.

INPUT:

- map – the map of simplicial complexes
- base_ring – a field (optional, default QQ)
- cohomology – boolean (optional, default False). If True , return the induced map in cohomology rather than homology.

Note: This is not intended to be used directly by the user, but instead via the method *induced_homology_morphism()* .

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: H = Hom(S1, S1)
sage: f = H({0:0, 1:2, 2:1})  # f switches two vertices
sage: f_star = f.induced_homology_morphism(QQ, cohomology=True)
sage: f_star
Graded algebra endomorphism of Cohomology ring of Minimal triangulation of the 1-
↪sphere over Rational Field
  Defn: induced by:
    Simplicial complex endomorphism of Minimal triangulation of the 1-sphere
      Defn: 0 |--> 0
            1 |--> 2
            2 |--> 1
sage: f_star.to_matrix(1)
[-1]
```

```
sage: f_star.to_matrix()
[ 1| 0]
[--+--]
[ 0|-1]

sage: T = simplicial_complexes.Torus()
sage: y = T.homology_with_basis(QQ).basis()[(1,1)]
sage: y.to_cycle()
(0, 2) - (0, 5) + (2, 5)
```

Since $(0,2)-(0,5)+(2,5)$ is a cycle representing a homology class in the torus, we can define a map $S^1 \to T$ inducing an inclusion on H_1:

```
sage: Hom(S1, T)({0:0, 1:2, 2:5})
Simplicial complex morphism:
  From: Minimal triangulation of the 1-sphere
  To: Minimal triangulation of the torus
  Defn: 0 |--> 0
        1 |--> 2
        2 |--> 5
sage: g = Hom(S1, T)({0:0, 1:2, 2: 5})
sage: g_star = g.induced_homology_morphism(QQ)
sage: g_star.to_matrix(0)
[1]
sage: g_star.to_matrix(1)
[0]
[1]
sage: g_star.to_matrix()
[1|0]
[-+-]
[0|0]
[0|1]
[-+-]
[0|0]
```

We can evaluate such a map on (co)homology classes:

```
sage: H = S1.homology_with_basis(QQ)
sage: a = H.basis()[(1,0)]
sage: g_star(a)
h_{1,1}

sage: T = S1.product(S1, is_mutable=False)
sage: diag = Hom(S1,T).diagonal_morphism()
sage: b,c = list(T.cohomology_ring().basis(1))
sage: diag_c = diag.induced_homology_morphism(cohomology=True)
sage: diag_c(b)
h^{1,0}
sage: diag_c(c)
0
```

base_ring ( )

The base ring for this map

EXAMPLES:

```
sage: K = simplicial_complexes.Simplex(2)
sage: H = Hom(K,K)
```

```
sage: id = H.identity()
sage: id.induced_homology_morphism(QQ).base_ring()
Rational Field
sage: id.induced_homology_morphism(GF(13)).base_ring()
Finite Field of size 13
```

is_identity ( )

True if this is the identity map on (co)homology.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: H = Hom(S1, S1)
sage: flip = H({0:0, 1:2, 2:1})
sage: flip.induced_homology_morphism(QQ).is_identity()
False
sage: flip.induced_homology_morphism(GF(2)).is_identity()
True
sage: rotate = H({0:1, 1:2, 2:0})
sage: rotate.induced_homology_morphism(QQ).is_identity()
True
```

is_injective ( )

True if this map is injective on (co)homology.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: K = simplicial_complexes.Simplex(2)
sage: H = Hom(S1, K)
sage: f = H({0:0, 1:1, 2:2})
sage: f.induced_homology_morphism().is_injective()
False
sage: f.induced_homology_morphism(cohomology=True).is_injective()
True

sage: T = simplicial_complexes.Torus()
sage: g = Hom(S1, T)({0:0, 1:3, 2: 6})
sage: g_star = g.induced_homology_morphism(QQ)
sage: g.is_injective()
True
```

is_surjective ( )

True if this map is surjective on (co)homology.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: K = simplicial_complexes.Simplex(2)
sage: H = Hom(S1, K)
sage: f = H({0:0, 1:1, 2:2})
sage: f.induced_homology_morphism().is_surjective()
True
sage: f.induced_homology_morphism(cohomology=True).is_surjective()
False
```

to_matrix (*deg=None*)

The matrix for this map.

If degree `deg` is specified, return the matrix just in that degree; otherwise, return the block matrix representing the entire map.

INPUT:

- `deg` – (optional, default `None`) the degree

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: S1_b = S1.barycentric_subdivision()
sage: S1_b.set_immutable()
sage: d = {(0,): 0, (0,1): 1, (1,): 2, (1,2): 0, (2,): 1, (0,2): 2}
sage: f = Hom(S1_b, S1)(d)
sage: h = f.induced_homology_morphism(QQ)
sage: h.to_matrix(1)
[2]
sage: h.to_matrix()
[1|0]
[-+-]
[0|2]
```

CHAPTER

NINETEEN

UTILITY FUNCTIONS FOR MATRICES

The actual computation of homology groups ends up being linear algebra with the differentials thought of as matrices. This module contains some utility functions for this purpose.

sage.homology.matrix_utils. **dhsw_snf** (*mat*, *verbose=False*)

Preprocess a matrix using the "Elimination algorithm" described by Dumas et al. [DHSW2003], and then call `elementary_divisors` on the resulting (smaller) matrix.

Note: 'snf' stands for 'Smith Normal Form'.

INPUT:

- `mat` – an integer matrix, either sparse or dense.

(They use the transpose of the matrix considered here, so they use rows instead of columns.)

ALGORITHM:

Go through `mat` one column at a time. For each column, add multiples of previous columns to it until either

- it's zero, in which case it should be deleted.
- its first nonzero entry is 1 or -1, in which case it should be kept.
- its first nonzero entry is something else, in which case it is deferred until the second pass.

Then do a second pass on the deferred columns.

At this point, the columns with 1 or -1 in the first entry contribute to the rank of the matrix, and these can be counted and then deleted (after using the 1 or -1 entry to clear out its row). Suppose that there were N of these.

The resulting matrix should be much smaller; we then feed it to Sage's `elementary_divisors` function, and prepend N 1's to account for the rows deleted in the previous step.

EXAMPLES:

```
sage: from sage.homology.matrix_utils import dhsw_snf
sage: mat = matrix(ZZ, 3, 4, range(12))
sage: dhsw_snf(mat)
[1, 4, 0]
sage: mat = random_matrix(ZZ, 20, 20, x=-1, y=2)
sage: mat.elementary_divisors() == dhsw_snf(mat)
True
```

CHAPTER

TWENTY

INTERFACE TO CHOMP

CHomP stands for "Computation Homology Program", and is good at computing homology of simplicial complexes, cubical complexes, and chain complexes. It can also compute homomorphisms induced on homology by maps. See the CHomP web page http://chomp.rutgers.edu/ for more information.

AUTHOR:

- John H. Palmieri

class sage.interfaces.chomp. **CHomP**
Interface to the CHomP package.

Parameters

- **program** (*string*) – which CHomP program to use
- **complex** – a simplicial or cubical complex
- **subcomplex** – a subcomplex of `complex` or None (the default)
- **base_ring** (ring; optional, default $\mathbf{Z}$) – ring over which to perform computations – must be $\mathbf{Z}$ or $\mathbf{F}_p$.
- **generators** (*boolean; optional, default False*) – if True, also return list of generators
- **verbose** (*boolean; optional, default False*) – if True, print helpful messages as the computation progresses
- **extra_opts** (*string*) – options passed directly to `program`

Returns homology groups as a dictionary indexed by dimension

The programs `homsimpl` , `homcubes` , and `homchain` are available through this interface. `homsimpl` computes the relative or absolute homology groups of simplicial complexes. `homcubes` computes the relative or absolute homology groups of cubical complexes. `homchain` computes the homology groups of chain complexes. For consistency with Sage's other homology computations, the answers produced by `homsimpl` and `homcubes` in the absolute case are converted to reduced homology.

Note also that CHomP can only compute over the integers or $\mathbf{F}_p$. CHomP is fast enough, though, that if you want rational information, you should consider using CHomP with integer coefficients, or with mod p coefficients for a sufficiently large p, rather than using Sage's built-in homology algorithms.

See also the documentation for the functions *homchain()* , *homcubes()* , and *homsimpl()* for more examples, including illustrations of some of the optional parameters.

EXAMPLES:

```
sage: from sage.interfaces.chomp import CHomP
sage: T = cubical_complexes.Torus()
sage: CHomP()('homcubes', T) # optional - CHomP
{0: 0, 1: Z x Z, 2: Z}
```

Relative homology of a segment relative to its endpoints:

```
sage: edge = simplicial_complexes.Simplex(1)
sage: ends = edge.n_skeleton(0)
sage: CHomP()('homsimpl', edge)  # optional - CHomP
{0: 0}
sage: CHomP()('homsimpl', edge, ends)  # optional - CHomP
{0: 0, 1: Z}
```

Homology of a chain complex:

```
sage: C = ChainComplex({3: 2 * identity_matrix(ZZ, 2)}, degree=-1)
sage: CHomP()('homchain', C)  # optional - CHomP
{2: C2 x C2}
```

help (*program*)

Print a help message for `program` , a program from the CHomP suite.

Parameters **program** (*string*) – which CHomP program to use

Returns nothing – just print a message

EXAMPLES:

```
sage: from sage.interfaces.chomp import CHomP
sage: CHomP().help('homcubes')   # optional - CHomP
HOMCUBES, ver. ... Copyright (C) ... by Pawel Pilarczyk...
```

`sage.interfaces.chomp.` **have_chomp** (*program='homsimpl'*)

Return True if this computer has `program` installed.

The first time it is run, this function caches its result in the variable `_have_chomp` – a dictionary indexed by program name – and any subsequent time, it just checks the value of the variable.

This program is used in the routine CHomP.__call__.

If this computer doesn't have CHomP installed, you may obtain it from http://chomp.rutgers.edu/.

EXAMPLES:

```
sage: from sage.interfaces.chomp import have_chomp
sage: have_chomp() # random -- depends on whether CHomP is installed
True
sage: 'homsimpl' in sage.interfaces.chomp._have_chomp
True
sage: sage.interfaces.chomp._have_chomp['homsimpl'] == have_chomp()
True
```

`sage.interfaces.chomp.` **homchain** (*complex=None*, ***kwds*)

Compute the homology of a chain complex using the CHomP program `homchain` .

Parameters

- **complex** – a chain complex

- **generators** (*boolean; optional, default False*) – if True, also return list of generators
- **verbose** (*boolean; optional, default False*) – if True, print helpful messages as the computation progresses
- **help** (*boolean; optional, default False*) – if True, just print a help message and exit
- **extra_opts** (*string*) – options passed directly to `homchain`

Returns homology groups as a dictionary indexed by dimension

EXAMPLES:

```
sage: from sage.interfaces.chomp import homchain
sage: C = cubical_complexes.Sphere(3).chain_complex()
sage: homchain(C)[3]    # optional - CHomP
Z
```

Generators: these are given as a list after the homology group. Each generator is specified as a cycle, an element in the appropriate free module over the base ring:

```
sage: C2 = delta_complexes.Sphere(2).chain_complex()
sage: homchain(C2, generators=True)[2]  # optional - CHomP
(Z, [(1, -1)])
sage: homchain(C2, generators=True, base_ring=GF(2))[2]  # optional - CHomP
(Vector space of dimension 1 over Finite Field of size 2, [(1, 1)])
```

TESTS:

Chain complexes concentrated in negative dimensions, cochain complexes, etc.:

```
sage: C = ChainComplex({-5: 4 * identity_matrix(ZZ, 2)}, degree=-1)
sage: homchain(C)    # optional - CHomP
{-6: C4 x C4}
sage: C = ChainComplex({-5: 4 * identity_matrix(ZZ, 2)}, degree=1)
sage: homchain(C, generators=True)    # optional - CHomP
{-4: (C4 x C4, [(1, 0), (0, 1)])}
```

`sage.interfaces.chomp.` **homcubes** (*complex=None*, *subcomplex=None*, ***kwds*)

Compute the homology of a cubical complex using the CHomP program `homcubes`. If the argument `subcomplex` is present, compute homology of `complex` relative to `subcomplex`.

Parameters

- **complex** – a cubical complex
- **subcomplex** – a subcomplex of `complex` or None (the default)
- **base_ring** (ring; optional, default $\mathbf{Z}$) – ring over which to perform computations – must be $\mathbf{Z}$ or $\mathbf{F}_p$.
- **generators** (*boolean; optional, default False*) – if True, also return list of generators
- **verbose** (*boolean; optional, default False*) – if True, print helpful messages as the computation progresses
- **help** (*boolean; optional, default False*) – if True, just print a help message and exit
- **extra_opts** (*string*) – options passed directly to `homcubes`

Returns homology groups as a dictionary indexed by dimension

EXAMPLES:

```
sage: from sage.interfaces.chomp import homcubes
sage: S = cubical_complexes.Sphere(3)
sage: homcubes(S)[3]    # optional - CHomP
Z
```

Relative homology:

```
sage: C3 = cubical_complexes.Cube(3)
sage: bdry = C3.n_skeleton(2)
sage: homcubes(C3, bdry)    # optional - CHomP
{0: 0, 1: 0, 2: 0, 3: Z}
```

Generators: these are given as a list after the homology group. Each generator is specified as a linear combination of cubes:

```
sage: homcubes(cubical_complexes.Sphere(1), generators=True, base_
↪ring=GF(2))[1][1]    # optional - CHomP
[[[1,1] x [0,1]] + [[0,1] x [1,1]] + [[0,1] x [0,0]] + [[0,0] x [0,1]]]
```

sage.interfaces.chomp. **homsimpl** (*complex=None*, *subcomplex=None*, ***kwds*)

Compute the homology of a simplicial complex using the CHomP program `homsimpl` . If the argument `subcomplex` is present, compute homology of `complex` relative to `subcomplex` .

Parameters

- **complex** – a simplicial complex
- **subcomplex** – a subcomplex of `complex` or None (the default)
- **base_ring** (ring; optional, default $\mathbf{Z}$) – ring over which to perform computations – must be $\mathbf{Z}$ or $\mathbf{F}_p$.
- **generators** (*boolean; optional, default False*) – if True, also return list of generators
- **verbose** (*boolean; optional, default False*) – if True, print helpful messages as the computation progresses
- **help** (*boolean; optional, default False*) – if True, just print a help message and exit
- **extra_opts** (*string*) – options passed directly to `program`

Returns homology groups as a dictionary indexed by dimension

EXAMPLES:

```
sage: from sage.interfaces.chomp import homsimpl
sage: T = simplicial_complexes.Torus()
sage: M8 = simplicial_complexes.MooreSpace(8)
sage: M4 = simplicial_complexes.MooreSpace(4)
sage: X = T.disjoint_union(T).disjoint_union(T).disjoint_union(M8).disjoint_
↪union(M4)
sage: homsimpl(X)[1]  # optional - CHomP
Z^6 x C4 x C8
```

Relative homology:

```
sage: S = simplicial_complexes.Simplex(3)
sage: bdry = S.n_skeleton(2)
sage: homsimpl(S, bdry)[3]    # optional - CHomP
Z
```

Generators: these are given as a list after the homology group. Each generator is specified as a linear combination of simplices:

```
sage: homsimpl(S, bdry, generators=True)[3]    # optional - CHomP
(Z, [(0, 1, 2, 3)])

sage: homsimpl(simplicial_complexes.Sphere(1), generators=True)    # optional -␣
↪CHomP
{0: 0, 1: (Z, [(0, 1) - (0, 2) + (1, 2)])}
```

TESTS:

Generators for a simplicial complex whose vertices are not integers:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: homsimpl(S1.join(S1), generators=True, base_ring=GF(2))[3][1]  # optional -␣
↪CHomP
[('L0', 'L1', 'R0', 'R1') + ('L0', 'L1', 'R0', 'R2') + ('L0', 'L1', 'R1', 'R2') +␣
↪('L0', 'L2', 'R0', 'R1') + ('L0', 'L2', 'R0', 'R2') + ('L0', 'L2', 'R1', 'R2')␣
↪+ ('L1', 'L2', 'R0', 'R1') + ('L1', 'L2', 'R0', 'R2') + ('L1', 'L2', 'R1', 'R2
↪')]
```

sage.interfaces.chomp. **process_generators_chain** (*gen_string*, *dim*, *base_ring=None*)

Process CHomP generator information for simplicial complexes.

Parameters

- **gen_string** (*string*) – generator output from CHomP
- **dim** (*integer*) – dimension in which to find generators
- **base_ring** (*optional, default ZZ*) – base ring over which to do the computations

Returns list of generators in each dimension, as described below

`gen_string` has the form

```
[H_0]
a1

[H_1]
a2
a3

[H_2]
a1 - a2
```

For each homology group, each line lists a homology generator as a linear combination of generators `ai` of the group of chains in the appropriate dimension. The elements `ai` are indexed starting with $i = 1$. Each generator is converted, using regular expressions, from a string to a vector (an element in the free module over `base_ring`), with `ai` representing the unit vector in coordinate $i - 1$. For example, the string `a1 -a2` gets converted to the vector `(1,-1)`.

Therefore the return value is a list of vectors.

EXAMPLES:

```
sage: from sage.interfaces.chomp import process_generators_chain
sage: s = "[H_0]\na1\n\n[H_1]\na2\na3\n"
sage: process_generators_chain(s, 1)
[(0, 1), (0, 0, 1)]
sage: s = "[H_0]\na1\n\n[H_1]\n5 * a2 - a1\na3\n"
sage: process_generators_chain(s, 1, base_ring=ZZ)
[(-1, 5), (0, 0, 1)]
sage: process_generators_chain(s, 1, base_ring=GF(2))
[(1, 1), (0, 0, 1)]
```

sage.interfaces.chomp. **process_generators_cubical** (*gen_string*, *dim*)

Process CHomP generator information for cubical complexes.

Parameters

- **gen_string** (*string*) – generator output from CHomP
- **dim** (*integer*) – dimension in which to find generators

Returns list of generators in each dimension, as described below

`gen_string` has the form

```
The 2 generators of H_1 follow:
generator 1
-1 * [(0,0,0,0,0)(0,0,0,0,1)]
1 * [(0,0,0,0,0)(0,0,1,0,0)]
...
generator 2
-1 * [(0,1,0,1,1)(1,1,0,1,1)]
-1 * [(0,1,0,0,1)(0,1,0,1,1)]
...
```

Each line consists of a coefficient multiplied by a cube; the cube is specified by its "bottom left" and "upper right" corners.

For technical reasons, we remove the first coordinate of each tuple, and using regular expressions, the remaining parts get converted from a string to a pair `(coefficient,Cube)` , with the cube represented as a product of tuples. For example, the first line in "generator 1" gets turned into

```
(-1, [0,0] x [0,0] x [0,0] x [0,1])
```

representing an element in the free abelian group with basis given by cubes. Each generator is a list of such pairs, representing the sum of such elements. These are reassembled in `CHomP.__call__()` to actual elements in the free module generated by the cubes of the cubical complex in the appropriate dimension.

Therefore the return value is a list of lists of pairs, one list of pairs for each generator.

EXAMPLES:

```
sage: from sage.interfaces.chomp import process_generators_cubical
sage: s = "The 2 generators of H_1 follow:\ngenerator 1:\n-1 *␣
↪[(0,0,0,0,0)(0,0,0,0,1)]\n1 * [(0,0,0,0,0)(0,0,1,0,0)]"
sage: process_generators_cubical(s, 1)
[[(-1, [0,0] x [0,0] x [0,0] x [0,1]), (1, [0,0] x [0,1] x [0,0] x [0,0])]]
sage: len(process_generators_cubical(s, 1))  # only one generator
1
```

sage.interfaces.chomp. **process_generators_simplicial** (*gen_string*, *dim*, *complex*)

Process CHomP generator information for simplicial complexes.

Parameters

- **gen_string** (*string*) – generator output from CHomP
- **dim** (*integer*) – dimension in which to find generators
- **complex** – simplicial complex under consideration

Returns list of generators in each dimension, as described below

`gen_string` has the form

```
The 2 generators of H_1 follow:
generator 1
-1 * (1,6)
1 * (1,4)
...
generator 2
-1 * (1,6)
1 * (1,4)
...
```

where each line contains a coefficient and a simplex. Each line is converted, using regular expressions, from a string to a pair `(coefficient,Simplex)` , like

```
(-1, (1,6))
```

representing an element in the free abelian group with basis given by simplices. Each generator is a list of such pairs, representing the sum of such elements. These are reassembled in `CHomP.__call__()` to actual elements in the free module generated by the simplices of the simplicial complex in the appropriate dimension.

Therefore the return value is a list of lists of pairs, one list of pairs for each generator.

EXAMPLES:

```
sage: from sage.interfaces.chomp import process_generators_simplicial
sage: s = "The 2 generators of H_1 follow:\ngenerator 1:\n-1 * (1,6)\n1 * (1,4)"
sage: process_generators_simplicial(s, 1, simplicial_complexes.Torus())
[[(-1, (1, 6)), (1, (1, 4))]]
```

CHAPTER

TWENTYONE

INDICES AND TABLES

- Index
- Module Index
- Search Page

h

i

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z