Sage Reference Manual: Cell complexes and their homology

Release 6.3

The Sage Development Team

August 11, 2014

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.

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*, ***kwds*)
> 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 $[r_0, d_0, r_1, d_1, r_2, d_2, ...]$, 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 $[d_0, d_1, d_2, ...]$ 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
{0: [1], 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 coerce 0 (<type
'sage.rings.finite_rings.element_givaro.FiniteField_givaroElement'>) to 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}
```

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

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([1,2]))
sage: C.grading_group()
Additive abelian group isomorphic to Z/3 + Z
sage: C.degree_of_differential()
(2, 1)
```

homology (*deg=None*, ***kwds*)
    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 `'auto'`); 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'` (the default), then use CHomP if available. CHomP is available at the web page http://chomp.rutgers.edu/. It is also an experimental 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 [DHSW]: 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, CHomP is by far the fastest option, followed by the `'auto'` or `'no_chomp'` setting of using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm [DHSW] for large matrices and Pari for small ones.

> **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)])}
```

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, -1, -1, 0, 0, 0, -1, 0, 1, 0))),
  (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, 0, 1, 0, -1, -1)))],
 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
```

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)
()
```

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:

```
from sage.matrix.constructor import zero_matrix
sage: S = simplicial_complexes.Sphere(1)
sage: S
Simplicial complex with vertex set (0, 1, 2) and facets {(1, 2), (0, 2), (0, 1)}
sage: C = S.chain_complex()
sage: C.differential()
{0: [], 1: [ 1  1  0]
[ 0 -1 -1]
[-1  0  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 morphism from Chain complex with at most 2 nonzero terms over Integer Ring to Chain com
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.structure.sage_object.SageObject

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

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
```

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 w

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 com
sage: x._matrix_dictionary
{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 0 1 0 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 0 1 0 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 0 1 0 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 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], -1: [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 com
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 com
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 comp
> ```

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
> ```

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 nth 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=
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=
```

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'
```

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 [Hat] (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 kth 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*,
**kwds)

    Bases: sage.structure.category_object.CategoryObject,
sage.homology.cell_complex.GenericCellComplex

Define a simplicial complex.

> **Parameters**
>
> > • **maximal_faces** – set of maximal faces
> >
> > • **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
>
> **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'.

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}
```

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
```

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)}
```

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

```
sage: T = SimplicialComplex([range(1,5)]).n_skeleton(1)
sage: T.homology()
{0: 0, 1: Z x Z x Z}
sage: T.add_face([1,2,3])
sage: T.homology()
{0: 0, 1: Z x Z, 2: 0}
```

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)}
```

automorphism_group()

   Returns 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'}
```

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
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 6 facets
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()
{0: set([(4,), (2,), (1,)]), 1: set([(1, 2), (1, 4)]), -1: set([()])}
sage: L = SimplicialComplex([[1,2]])
sage: Y.faces(subcomplex=L)
{0: set([(4,)]), 1: set([(1, 4)]), -1: set([])}
```

chain_complex (***kwds*)

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_diffs** (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 = 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
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 10 facets
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}
```

face_iterator(*increasing=True*)
    An iterator for the faces in this simplicial complex.

    INPUTS:

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

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 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 | e6, e5, e3, e9, e4*e7^-1
```

```
sage: C2.simplified()
Finitely presented group < e0 | e0^2 >
```

This is the same answer given if the argument `simplify` is True (the default):
```
sage: RP2.fundamental_group()
Finitely presented group < e0 | e0^2 >
```

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 < e0 | e0^2 >
```

Some other examples:
```
sage: S1.wedge(S1).fundamental_group()
Finitely presented group < e0, e1 | >
sage: simplicial_complexes.Torus().fundamental_group()
Finitely presented group < e0, e3 | e0*e3^-1*e0^-1*e3 >
sage: simplicial_complexes.MooreSpace(5).fundamental_group()
Finitely presented group < e1 | e1^5 >
```

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
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 2), (1, 2, 3),
```

```
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_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 empy 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(*ncpus=0*)

> Returns True if `self` is Cohen-Macaulay, i.e., if $\tilde{H}_i(\mathrm{lk}_\Delta(F); \mathbf{Z}) = 0$ for all $F \in \Delta$ and $i < \dim \mathrm{lk}_\Delta(F)$. Here, Δ is `self`, and lk denotes the link operator on `self`.
>
> INPUT:
>
> > •`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
```

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*, *certify=False*)
>   Checks whether two simplicial complexes are isomorphic
>
>   INPUT:
>
>   >   •certify - 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, certify=True)
>   (True, {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f'})
>   sage: Z3.is_isomorphic(Z2)
>   False
>   ```

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
```

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: first take the complement (within the vertex set) of each facet, obtaining a set $(f_1, f_2, ...)$ of simplices. Now form the set of all simplices of the form $(v_1, v_2, ...)$ where vertex v_i is in face f_i. This set will contain the minimal nonfaces and may contain some non-minimal nonfaces also, so loop through the set to find the minimal ones. (The last two steps are taken care of by the `_transpose_simplices` routine.)

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)}
```

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.

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)
set([(1, 3, 4), (1, 2, 3), (2, 3, 4), (1, 2, 4)])
sage: K = SimplicialComplex([[1,2,3], [2,3,4]])
sage: Z.n_faces(2, subcomplex=K)
set([(1, 3, 4), (1, 2, 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),
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)}
```

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
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6) and 14 facets
sage: K = simplicial_complexes.KleinBottle(); K
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6, 7) and 16 facets
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)}
```

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 (x
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 p. 434 in the cited article): 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.

REFERENCES:

- Basudeb Datta, "Minimal triangulations of manifolds", J. Indian Inst. Sci. 87 (2007), no. 4, 429-449.

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 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)
```

Note that this actually returns a simplex:
```
sage: type(S.vertices())
<class 'sage.homology.simplicial_complex.Simplex'>
```

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.**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(range(3), range(3), length=2)
[]
sage: lattice_paths(range(3), range(3), length=3)
[[(0, 0), (1, 1), (2, 2)]]
sage: lattice_paths(range(3), 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'
> ```

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 {0: 'L0R0', 1: 'L1R1', 2: 'L2R2', 3: 'L3R3', 4: 'L4R4', 5: 'L5R5'} from S

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 {'L1R1': 1, 'L3R3': 3, 'L2R2': 2, 'L0R0': 0} from Simplicial complex with

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
Simplicial complex with vertex set (0, 1, 2) and facets {(1, 2), (0, 2), (0, 1)}
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 {'L2R(2, 0)': 2, 'L2R(2, 1)': 2, 'L0R(0, 0)': 0, 'L0R(0, 1)': 0, 'L1R(1,
```

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

Bases: `sage.structure.sage_object.SageObject`

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 {0: 0, 1: 1, 2: 2} from Simplicial complex with vertex set (0, 1
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
sage: a._matrix_dictionary
{0: [0 0 0]
[0 1 0]
[0 0 1]
[1 0 0],
 1: [0 0 0]
[0 1 0]
[0 0 0]
[1 0 0]
[0 0 0]
```

```
 [0 0 1],
 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
sage: x.associated_chain_complex_morphism(cochain=True)
Chain complex morphism from Chain complex with at most 3 nonzero terms over Integer Ring to
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
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
```

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

codomain()
    Returns the codomain of the morphism.

    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.codomain()
Simplicial complex with vertex set (0, 1) and facets {(0, 1)}
```

domain()
    Returns the domain of the morphism.

    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.domain()
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(2, 3), (0, 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 {'L1R2': 1, 'L1R1': 1, 'L2R0': 0, 'L0R0': 0}
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)}
```

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)}
```

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
>   Simplicial complex with vertex set (0, 1, 2) and facets {(1, 2), (0, 2), (0, 1)}
>   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 {0: 0, 1: 1, 2: 2, 3: 3} from Simplicial complex with vertex set
>   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
```

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 {0: 0, 1: 1, 2: 3} from Simplicial complex with vertex set (0, 1, 2) and
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 m
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 'hom_category'
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()

Returns 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 {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0} from Simplicial
```

diagonal_morphism(*rename_vertices=True*)

Returns 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 {0: 'L0R0', 1: 'L1R1', 2: 'L2R2', 3: 'L3R3'} from
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 2), (1, 2, 3),
to Simplicial complex with 16 vertices and 96 facets

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 {0: (0, 0), 1: (1, 1)} 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),)}
```

identity()

Returns 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 morphism {0: 0, 1: 1} from
Simplicial complex with vertex set (0, 1) and facets {(0, 1)} to
Simplicial complex with vertex set (0, 1) and facets {(0, 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 comple
sage: from sage.homology.simplicial_complex_homset import is_SimplicialComplexHomset
sage: is_SimplicialComplexHomset(H)
True
```

EXAMPLES OF SIMPLICIAL COMPLEXES

AUTHORS:

- John H. Palmieri (2009-04)

This file constructs some 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 some functions for constructing some other simplicial complexes: the simplicial complex of not-i-connected graphs on n vertices, the matching complex on n vertices, and the chessboard complex for an n by i chessboard. 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. You can get a list by typing `simplicial_complexes.` and hitting the TAB key:

```
simplicial_complexes.BarnetteSphere
simplicial_complexes.BrucknerGrunbaumSphere
simplicial_complexes.ChessboardComplex
simplicial_complexes.ComplexProjectivePlane
simplicial_complexes.K3Surface
simplicial_complexes.KleinBottle
simplicial_complexes.MatchingComplex
simplicial_complexes.MooreSpace
simplicial_complexes.NotIConnectedGraphs
simplicial_complexes.PoincareHomologyThreeSphere
simplicial_complexes.PseudoQuaternionicProjectivePlane
simplicial_complexes.RandomComplex
simplicial_complexes.RealProjectivePlane
simplicial_complexes.RealProjectiveSpace
simplicial_complexes.Simplex
simplicial_complexes.Sphere
simplicial_complexes.SumComplex
simplicial_complexes.SurfaceOfGenus
simplicial_complexes.Torus
```

See the documentation for `simplicial_complexes` and for each particular type of example for full details.

class `sage.homology.examples.`**`SimplicialComplexExamples`**

Some examples of simplicial complexes.

Here are the available examples; you can also type `simplicial_complexes.` and hit tab to get a list:

- BarnetteSphere()
- BrucknerGrunbaumSphere()
- ChessboardComplex()
- ComplexProjectivePlane()
- K3Surface()
- KleinBottle()
- MatchingComplex()
- MooreSpace()
- NotIConnectedGraphs()
- PoincareHomologyThreeSphere()
- PseudoQuaternionicProjectivePlane()
- RandomComplex()
- RealProjectivePlane()
- RealProjectiveSpace()
- Simplex()
- Sphere()
- SumComplex()
- SurfaceOfGenus()
- Torus()

EXAMPLES:
```
sage: S = simplicial_complexes.Sphere(2) # the 2-sphere
sage: S.homology()
{0: 0, 1: 0, 2: Z}
sage: simplicial_complexes.SurfaceOfGenus(3)
Simplicial complex with 15 vertices and 38 facets
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}
```

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 [B1970]. The construction here uses the labeling from De Loera, Rambau and Santos [DLRS2010]. Another reference is chapter III.4 of Ewald [E1996].
>
> EXAMPLES:
> ```
> sage: BS = simplicial_complexes.BarnetteSphere() ; BS
> Simplicial complex with vertex set (1, 2, 3, 4, 5, 6, 7, 8) and 19 facets
> 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 [E1996].:

```
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
```

REFERENCES:

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 [Ku1995].
>
> EXAMPLES:
> ```
> sage: BGS = simplicial_complexes.BrucknerGrunbaumSphere() ; BGS
> Simplicial complex with vertex set (1, 2, 3, 4, 5, 6, 7, 8) and 20 facets
> sage: BGS.f_vector()
> [1, 8, 28, 40, 20]
> ```
>
> REFERENCES:

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 [Wa2003] 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}
> ```

ComplexProjectivePlane()
> A minimal triangulation of the complex projective plane.

This was constructed by Kühnel and Banchoff [KB1983].

REFERENCES:

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
```

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].

REFERENCES:

EXAMPLES:
```
sage: K3=simplicial_complexes.K3Surface() ; K3
Simplicial complex with 16 vertices and 288 facets
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.

KleinBottle()

A minimal triangulation of the Klein bottle, as presented for example in Davide Cervone's thesis [Ce1994].

EXAMPLES:
```
sage: simplicial_complexes.KleinBottle()
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6, 7) and 16 facets
```

REFERENCES:

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 [Wa2003] 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, 201
Z^132
```

REFERENCES:

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)
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 10 facets
sage: simplicial_complexes.MooreSpace(3).homology()[1]
C3
sage: simplicial_complexes.MooreSpace(4).suspension().homology()[2]
C4
sage: simplicial_complexes.MooreSpace(8)
Simplicial complex with 19 vertices and 54 facets
```

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
```

REFERENCES:

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].
>
> REFERENCES:
>
> 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
> ```

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
> ```

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 [BrK92]. 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
```

REFERENCES:

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
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 10 facets
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)
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and facets {(0, 1, 2, 3, 4, 5)}
```

REFERENCES:

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
```

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 [Wa1970]; this function returns a triangulation with 11 vertices, as given by Lutz [Lu2005].

- $\mathbf{R}P^4$: 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 [Da2007], 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 [Ku1987] 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.

REFERENCES:

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
```

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)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 1, 2, 3)}
sage: simplicial_complexes.Simplex(5).euler_characteristic()
1
```

Sphere(n)

A minimal triangulation of the n-dimensional sphere.

INPUT:

    •n – positive integer

EXAMPLES:

```
sage: simplicial_complexes.Sphere(2)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 2), (1, 2, 3),
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]]
```

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 $\mathbf{Z}/(n)$

REFERENCES:

EXAMPLES:

```
sage: S = simplicial_complexes.SumComplex(10, [0,1,2,3,6]); S
Simplicial complex with 10 vertices and 126 facets
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
Simplicial complex with 11 vertices and 45 facets
sage: S.homology(1)
C23
sage: S = simplicial_complexes.SumComplex(11, [0,1,2,3,4,7]); S
Simplicial complex with 11 vertices and 252 facets
sage: S.homology() # 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
Simplicial complex with 13 vertices and 66 facets
sage: S.homology(1)
C159
sage: factor(159)
3 * 53
sage: S = simplicial_complexes.SumComplex(13, [0,1,2,5]); S
Simplicial complex with 13 vertices and 220 facets
sage: S.homology() # 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
Simplicial complex with 13 vertices and 495 facets
sage: S.homology() # 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
Simplicial complex with 17 vertices and 120 facets
sage: S.homology(1)
C140183
sage: factor(140183)
103 * 1361
sage: S = simplicial_complexes.SumComplex(19, [0, 1, 4]); S
```

```
Simplicial complex with 19 vertices and 153 facets
sage: S.homology(1)
C5670599
sage: factor(5670599)
11 * 191 * 2699
sage: S = simplicial_complexes.SumComplex(31, [0, 1, 4]); S
Simplicial complex with 31 vertices and 435 facets
sage: S.homology(1) # long time
C5 x C5 x C5 x C5 x C26951480558170926865
sage: factor(26951480558170926865)
5 * 311 * 683 * 1117 * 11657 * 1948909
```

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)
Simplicial complex with 11 vertices and 26 facets
sage: simplicial_complexes.SurfaceOfGenus(1, orientable=False)
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 10 facets
```

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:

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 [Da2007].

    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])
[set([(1, 3), (2, 4)]), set([(2, 3), (1, 4)])]
sage: matching([0,2], [0])
[set([(0, 0)]), set([(2, 0)])]
```

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 [Hat], especially Section 2.1 and the Appendix on "Simplicial CW Structures". As Hatcher points out, Δ-complexes were first introduced by Eilenberg and Zilber [EZ], 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:

class sage.homology.delta_complex.**DeltaComplex**(*data=None*, ***kwds*)

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)`.
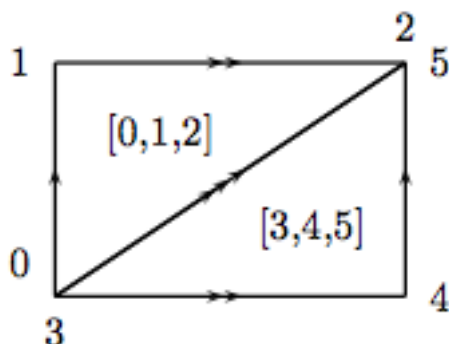
- `data` may be nested lists or tuples. The nth entry in the list is a list of the n-simplices in the complex, and each n-simplex is encoded as a list, the ith entry of which is its ith face. Each face is represented by an integer, giving its index in the list of (n-1)-faces. For example, consider this:

```
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.
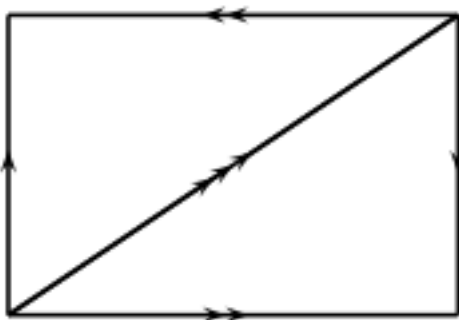

```
sage: P.homology(1)
C2
sage: P.cohomology(2)
C2
```

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()
{0: ((), ()), 1: ((1, 0), (1, 0), (0, 0)), 2: ((1, 0, 2), (0, 1, 2)), -1: ((),)}
```

•`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
{0: ((), ()), 1: ((1, 0), (1, 0), (0, 0)), 2: ((1, 0, 2), (0, 1, 2)), -1: ((),)}
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.

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()
{0: ((), (), ()), 1: ((0, 1), (0, 2), (1, 2)), 2: ((0, 1, 2), (0, 1, 2)), -1: ((),)}
sage: A = S2.subcomplex({1: [0,2]}) # one edge
sage: S2.cells(subcomplex=A)
{0: (None, None, None), 1: (None, (0, 2), None), 2: ((0, 1, 2), (0, 1, 2)), -1: (None,)}
```

chain_complex(***kwds*)

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_diffs** (*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)
```

```
{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 [Lee].

REFERENCES:

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_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 sub-
>   complex.

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()
{0: ((), ()), 1: ((0, 1),), -1: ((),)}
```

```
sage: ends = line.subcomplex({0: (0, 1)})
sage: ends.cells()
{0: ((), ()), -1: ((),)}
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.
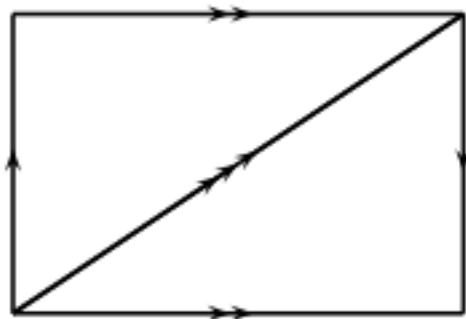
EXAMPLES:
```
sage: delta_complexes.KleinBottle()
Delta complex with 1 vertex and 7 simplices
```

RealProjectivePlane()

A Δ-complex representation of the real projective plane, consisting of two vertices, three edges, and two triangles.
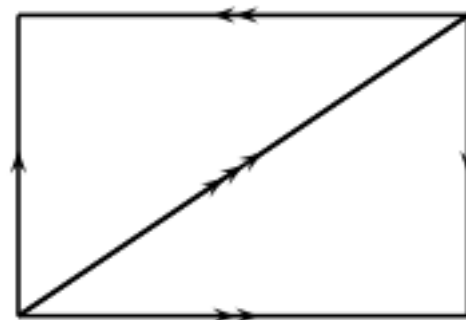
EXAMPLES:
```
sage: P = delta_complexes.RealProjectivePlane()
sage: P.cohomology(1)
0
sage: P.cohomology(2)
C2
sage: P.cohomology(dim=1, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
sage: P.cohomology(dim=2, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of 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.
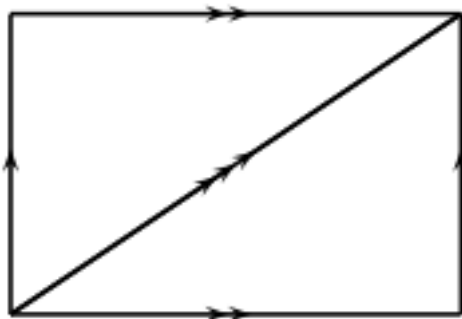
> EXAMPLES:
> ```
> sage: delta_complexes.Torus().homology(1)
> Z x Z
> ```

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 [KMM], 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,
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.

REFERENCES:

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
```

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=[ ]*, ***kwds*)

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],
sage: S1.cells()
{0: set([[1,1] x [2,2], [0,0] x [2,2], [1,1] x [3,3], [0,0] x [3,3]]), 1: set([[0,1] x [3,3], [1
```

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.

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: S2.cells()[2]
set([[1,1] x [0,1] x [0,1], [0,1] x [0,0] x [0,1], [0,1] x [1,1] x [0,1], [0,0] x [0,1] x [0
```

chain_complex(***kwds*)

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_diffs** (*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)
set([[0,1] x [0,1] x [0,1]])
sage: C.n_cubes(2)
set([[1,1] x [0,1] x [0,1], [0,1] x [0,0] x [0,1], [0,1] x [1,1] x [0,1], [0,0] x [0,1] x [0
```

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]
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
```

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()
```

    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
```

chain_complex (***kwds*)

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_diffs – 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
```

cohomology (*dim=None*, ***kwds*)

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** –

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],
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
```

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 simplicial complex with another one.

>   > **Parameters right** – the other simplicial 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
>   ```

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,), (0,), (0, 1), (1, 2)]

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*, ***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 'auto'*) – 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.

Note: The keyword arguments to this function get passed on to :meth:`chain_complex` and its homology.

ALGORITHM:

If `algorithm` is set to 'auto' (the default), then use CHomP if available. (CHomP is available at the web page http://chomp.rutgers.edu/. It is also an experimental 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 `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: 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, CHomP is by far the fastest option, followed by the 'auto' or 'no_chomp' setting of using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm for large matrices and Pari for small ones.

EXAMPLES:
```
sage: P = delta_complexes.RealProjectivePlane()
sage: P.homology()
{0: 0, 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)
```

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
```

join(*right*, ***kwds*)

> The join of this cell complex with another one.
>
> > **Parameters   right** – the other simplicial 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
> ```

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: simplicial_complexes.Simplex(2).n_cells(1)
> [(1, 2), (0, 2), (0, 1)]
> 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_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
> ```

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
```

wedge(*right*)

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

> **Parameters right** – the other simplicial 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
```

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
```

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. [DHSW], 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
```

REFERENCES:

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 **Z**) – ring over which to perform computations – must be **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)

- **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
[[[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 **Z**) – ring over which to perform computations – must be **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', '
```

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
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))]]
```

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[B1970] Barnette, "Diagrams and Schlegel diagrams", in Combinatorial Structures and Their Applications, Proc. Calgary Internat. Conference 1969, New York, 1970, Gordon and Breach.

[DLRS2010] De Loera, Rambau and Santos, "Triangulations: Structures for Algorithms and Applications", Algorithms and Computation in Mathematics, Volume 25, Springer, 2011.

[E1996] Ewald, "Combinatorial Convexity and Algebraic Geometry", vol. 168 of Graduate Texts in Mathematics, Springer, 1996

[Br1910] Bruckner, "Uber die Ableitung der allgemeinen Polytope und die nach Isomorphismus verschiedenen Typen der allgemeinen Achtzelle (Oktatope)", Verhand. Konik. Akad. Wetenschap, Erste Sectie, 10 (1910)

[GrS1967] Grunbaum and Sreedharan, "An enumeration of simplicial 4-polytopes with 8 vertices", J. Comb. Th. 2, 437-465 (1967)

[Ku1995] Kuhnel, "Tight Polyhedral Submanifolds and Tight Triangulations" Lecture Notes in Mathematics Volume 1612, 1995

[KB1983] W. Kühnel and T. F. Banchoff, "The 9-vertex complex projective plane", Math. Intelligencer 5 (1983), no. 3, 11-22.

[CK2001] M. Casella and W. Kühnel, "A triangulated K3 surface with the minimum number of vertices", Topology 40 (2001), 753–772.

[SK2011] J. Spreer and W. Kühnel, "Combinatorial properties of the K3 surface: Simplicial blowups and slicings", Experimental Mathematics, Volume 20, Issue 2, 2011.

[Ce1994] D. P. Cervone, "Vertex-minimal simplicial immersions of the Klein bottle in three-space", Geom. Ded. 50 (1994) 117-141, http://www.math.union.edu/~dpvc/papers/1993-03.kb/vmkb.pdf.

[Wa2003] Wachs, "Topology of Matching, Chessboard and General Bounded Degree Graph Complexes" (Algebra Universalis Special Issue in Memory of Gian-Carlo Rota, Algebra Universalis, 49 (2003) 345-385)

[BBLSW1999] Babson, Bjorner, Linusson, Shareshian, and Welker, "Complexes of not i-connected graphs," Topology 38 (1999), 271-299

[DHSW2003] Dumas, Heckenbach, Saunders, Welker, "Computing simplicial homology based on efficient Smith normal form algorithms," in "Algebra, geometry, and software systems" (2003), 177-206.

[BL2000] Anders Björner and Frank H. Lutz, "Simplicial manifolds, bistellar flips and a 16-vertex triangulation of the Poincaré homology 3-sphere", Experiment. Math. 9 (2000), no. 2, 275-289.

[BrK92] Brehm U., Kuhnel W., "15-vertex triangulations of an 8-manifold", Math. Annalen 294 (1992), no. 1, 167-193.

[MW2009] Meshulam and Wallach, "Homological connectivity of random k-dimensional complexes", preprint, math.CO/0609773.

[Da2007] Basudeb Datta, "Minimal triangulations of manifolds", J. Indian Inst. Sci. 87 (2007), no. 4, 429-449.

[Ku1987] W. Kühnel, "Minimal triangulations of Kummer varieties", Abh. Math. Sem. Univ. Hamburg 57 (1987), 7-20.

[Lu2005] Frank H. Lutz, "Triangulated Manifolds with Few Vertices: Combinatorial Manifolds", preprint (2005), arXiv:math/0506372.

[Wa1970] David W. Walkup, "The lower bound conjecture for 3- and 4-manifolds", Acta Math. 125 (1970), 75-107.

[LMR2010] N. Linial, R. Meshulam and M. Rosenthal, "Sum complexes – a new family of hypertrees", Discrete & Computational Geometry, 2010, Volume 44, Number 3, Pages 622-636

[LutzCsas] Császár's Torus

[Hat] Allen Hatcher, "Algebraic Topology", Cambridge University Press (2002).

[EZ] S. Eilenberg and J. Zilber, "Semi-Simplicial Complexes and Singular Homology", Ann. Math. (2) 51 (1950), 499-513.

[Lee] John M. Lee, Introduction to Topological Manifolds, Springer-Verlag, GTM volume 202.

[KMM] Tomasz Kaczynski, Konstantin Mischaikow, and Marian Mrozek, "Computational Homology", Springer-Verlag (2004).

[DHSW] Dumas, Heckenbach, Saunders, and Welker. *Computing simplicial homology based on efficient Smith normal form algorithms*. Algebra, geometry, and software systems. (2003) 177-206.

PYTHON MODULE INDEX

h

i

INDEX