# Topology

***Release 10.5.rc0***

**The Sage Development Team**

**Nov 16, 2024**

# CONTENTS


1 Finite simplicial complexes 3

2 Morphisms of simplicial complexes 47

3 Homsets between simplicial complexes 57

4 Examples of simplicial complexes 61

5 Finite Delta-complexes 77

6 Finite cubical complexes 93

7 Simplicial sets 109

8 Methods of constructing simplicial sets 149

9 Examples of simplicial sets. 175

10 Catalog of simplicial sets 183

11 Morphisms and homsets for simplicial sets 185

12 Generic cell complexes 199

13 Finite filtered complexes 213

14 Moment-angle complexes 217

15 Indices and Tables 227

Bibliography 229

Python Module Index 231

Index 233

Sage includes some tools for topology, and in particular cell complexes: (filtered) simplicial complexes, $\Delta$-complexes, cubical complexes, and simplicial sets. A class of generic cell complexes is also available, mainly for developers who want to use it as a base for other types of cell complexes.

CHAPTER

ONE

# 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.
- Jeremy Martin (2016-06-02): added cone_vertices, decone, is_balanced, is_partitionable, intersection methods

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 $K$.

**Note**

In Sage, the elements of the vertex set are determined automatically: $V$ is defined to be the union of the sets in $K$. So in Sage's implementation of simplicial complexes, every vertex is included in some face.

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

For any simplicial complex $K$ and any commutative ring $R$ there is an associated chain complex, with differential of degree $-1$. The $n$-th term is the free $R$-module with basis given by the $n$-simplices of $K$. The differential is determined

by its value on any simplex: on the $n$-simplex with vertices $(v_0, v_1, ..., v_n)$, the differential is the alternating sum with $i$-th summand $(-1)^i$ multiplied by the $(n-1)$-simplex obtained by omitting vertex $v_i$.

In the implementation here, the vertex set must be finite. To define a simplicial complex, specify its *facets*: the maximal subsets (with respect to inclusion) of the vertex set belonging to $K$. Each facet can be specified as a list, a tuple, or a set.

**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 {(1,), (3, 7)}
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 {(0, 1), (0, 3), (1, 2), (2, 3)}
```

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                          #␣
↪needs sage.modules
{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: x0, x1, x2, x3 = X.stanley_reisner_ring().gens()                         #␣
↪needs sage.libs.singular
sage: x0*x2 == x1*x3 == 0                                                      #␣
↪needs sage.libs.singular
True
sage: X.is_pure()
True
```

Mutability (see Issue #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 {(3,), (1, 4), (2, 4)}
```

(continues on next page)

(continued from previous page)

```
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 Issue #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.topology.simplicial_complex.**Simplex**(*X*)

Bases: 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.

INPUT:

- X – set of vertices (integer, list, or other iterable)

OUTPUT: simplex with those vertices

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

**Warning**

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

EXAMPLES:

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

Vertices may be tuples but not lists:

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

**alexander_whitney**(*dim*)

Subdivide this simplex into a pair of simplices.

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

INPUT:

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

OUTPUT:

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

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

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

EXAMPLES:

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

**dimension**()

The dimension of this simplex.

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

EXAMPLES:

```
sage: Simplex(5).dimension() == 5
True
```

(continues on next page)

(continued from previous page)

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

**face**(*n*)

The $n$-th face of this simplex.

INPUT:

- `n` – integer between 0 and the dimension of this simplex

OUTPUT: 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 $[v_0, ..., v_k, w_0, ..., w_n]$.

INPUT:

- `right` – the other simplex (the right-hand factor)

- `rename_vertices` – boolean (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.

INPUT:

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

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

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

**Note**

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

EXAMPLES:

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

**set** ()

The frozenset attached to this simplex.

EXAMPLES:

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

**tuple**()

The tuple attached to this simplex.

EXAMPLES:

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

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

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

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

Bases: Parent, *GenericCellComplex*

Define a simplicial complex.

INPUT:

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

OUTPUT: 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'. maximal_faces can also be a list containing a single nonnegative integer $n$, in which case this constructs the simplicial complex with a single $n$-simplex as the only facet.

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

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

sort_facets: if not set to None, the default, this should be a dictionary, used for sorting the vertices in each facet. The keys must be the vertices for the simplicial complex, and the values should be distinct sortable objects,

for example integers. This should not need to be specified except in very special circumstances; currently the only use in the Sage library is when defining the product of a simplicial complex with itself: in this case, the vertices in the product must be sorted compatibly with the vertices in each factor so that the diagonal map is properly defined.

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,), (0, 2), (0, 3)}
```

Finally, if the first argument 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()                                                          #␣
↪needs sage.modules
{0: 0, 1: Z x Z, 2: Z}
```

In the situation where the first argument is a simplicial complex or another object with a built-in conversion, most of the other arguments are ignored. The only exception is `is_mutable`:

```
sage: S.is_mutable()
True
sage: SimplicialComplex(S, is_mutable=False).is_mutable()
False
```

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

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

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

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

**Warning**

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

**F_triangle**(*S*)

Return the F-triangle of `self` with respect to one maximal simplex `S`.

This is the bivariate generating polynomial of all faces, according to the number of elements in `S` and outside `S`.

OUTPUT: an `F_triangle`

**See also**

Not to be confused with *f_triangle()* .

EXAMPLES:

```
sage: cs = simplicial_complexes.Torus()
sage: cs.F_triangle(cs.facets()[0])                                             #␣
↪needs sage.combinat
F: x^3 + 9*x^2*y + 3*x*y^2 + y^3 + 6*x^2 + 12*x*y
+ 3*y^2 + 4*x + 3*y + 1
```

**add_face**(*face*)

Add a face to this simplicial complex.

INPUT:

- `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 {(0, 1), (1, 2, 3)}
```

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 {(0, 1), (1, 2, 3)}
```

**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.

INPUT:

- `is_mutable` – boolean (default: `True`); determine whether 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 {(0,), (1,), (2,), (3,), (4,)}
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 {(0, 2), (1, 3)}
```

**alexander_whitney**(*simplex*, *dim_left*)

Subdivide this simplex into a pair of simplices.

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

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

INPUT:

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

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

EXAMPLES:

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

**algebraic_topological_model**(*base_ring=None*)

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

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

INPUT:

- `base_ring` – coefficient ring (default: `QQ`); must be a field

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

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

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

OUTPUT: a pair consisting of

- chain contraction `phi` associated to $C$, $M$, $\pi$, and $\iota$
- the chain complex $M$

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

EXAMPLES:

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

**automorphism_group**()

Return the automorphism group of the simplicial complex.

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

**Warning**

Since Issue #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))                #␣
↪needs sage.groups
True

sage: P = simplicial_complexes.RealProjectivePlane()
sage: P.automorphism_group().is_isomorphic(AlternatingGroup(5))              #␣
↪needs sage.groups
True

sage: Z = SimplicialComplex([['1','2'],['2','3','a']])
sage: Z.automorphism_group().is_isomorphic(CyclicPermutationGroup(2))        #␣
↪needs sage.groups
True
sage: group = Z.automorphism_group()                                         #␣
↪needs sage.groups
sage: sorted(group.domain())                                                 #␣
↪needs sage.groups
['1', '2', '3', 'a']
```

Check that Issue #17032 is fixed:

```
sage: s = SimplicialComplex([[(0,1),(2,3)]])
sage: s.automorphism_group().cardinality()                                  #␣
↪needs sage.groups
2
```

**barycentric_subdivision**()

The barycentric subdivision of this simplicial complex.

See Wikipedia article Barycentric_subdivision for a definition.

EXAMPLES:

```
sage: triangle = SimplicialComplex([[0,1], [1,2], [0, 2]])
sage: hexagon = triangle.barycentric_subdivision(); hexagon
Simplicial complex with 6 vertices and 6 facets
sage: hexagon.homology(1) == triangle.homology(1)                           #␣
↪needs sage.modules
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); S4
Minimal triangulation of the 4-sphere
sage: S4.barycentric_subdivision()
Simplicial complex with 62 vertices and 720 facets
```

**bigraded_betti_number**($a$, $b$, *base_ring=Integer Ring*, *verbose=False*)

Return the bigraded Betti number indexed in the form $(-a, 2b)$.

Bigraded Betti number with indices $(-a, 2b)$ is defined as a sum of ranks of $(b-a-1)$-th (co)homologies of full subcomplexes with exactly $b$ vertices.

INPUT:

- `base_ring` – (default: `ZZ`) the base ring used when computing homology
- `verbose` – boolean (default: `False`); if `True`, print messages during the computation, which indicate in which subcomplexes non-trivial homologies appear

**Note**

If `verbose` is `True`, then caching is avoided.

EXAMPLES:

```
sage: # needs sage.modules
sage: X = SimplicialComplex([[0,1],[1,2],[2,0],[1,3]])
sage: X.bigraded_betti_number(-1, 4, base_ring=QQ)
2
sage: X.bigraded_betti_number(-1, 8)
0
sage: X.bigraded_betti_number(-2, 5)
0
sage: X.bigraded_betti_number(0, 0)
1
sage: sorted(X.bigraded_betti_numbers().items(), reverse=True)
```

(continues on next page)

(continued from previous page)

```
[((0, 0), 1), ((-1, 6), 1), ((-1, 4), 2), ((-2, 8), 1), ((-2, 6), 1)]
sage: X.bigraded_betti_number(-1, 4, base_ring=QQ)
2
sage: X.bigraded_betti_number(-1, 8)
0
sage: Y = SimplicialComplex([[1,2,3],[1,2,4],[3,5],[4,5]])
sage: Y.bigraded_betti_number(-1, 4, verbose=True)
Non-trivial homology Z in dimension 0 of the full subcomplex
generated by a set of vertices (1, 5)
Non-trivial homology Z in dimension 0 of the full subcomplex
generated by a set of vertices (2, 5)
Non-trivial homology Z in dimension 0 of the full subcomplex
generated by a set of vertices (3, 4)
3
```

**bigraded_betti_numbers**(*base_ring=Integer Ring*, *verbose=False*)

Return a dictionary of the bigraded Betti numbers of `self`, with keys $(-a, 2b)$.

INPUT:

- `base_ring` – (default: `ZZ`) the base ring used when computing homology
- `verbose` – boolean (default: `False`); if `True`, print messages during the computation, which indicate in which subcomplexes non-trivial homologies appear

**Note**

If `verbose` is `True`, then caching is avoided.

**See also**

See *bigraded_betti_number()* for more information.

EXAMPLES:

```
sage: X = SimplicialComplex([[0,1],[1,2],[1,3],[2,3]])
sage: Y = SimplicialComplex([[1,2,3],[1,2,4],[3,5],[4,5]])
sage: sorted(X.bigraded_betti_numbers(base_ring=QQ).items(), reverse=True)
[((0, 0), 1), ((-1, 6), 1), ((-1, 4), 2), ((-2, 8), 1), ((-2, 6), 1)]
sage: sorted(Y.bigraded_betti_numbers(verbose=True).items(), reverse=True)
(-1, 4): Non-trivial homology Z in dimension 0 of the full
subcomplex generated by a set of vertices (1, 5)
(-1, 4): Non-trivial homology Z in dimension 0 of the full
subcomplex generated by a set of vertices (2, 5)
(-1, 4): Non-trivial homology Z in dimension 0 of the full
subcomplex generated by a set of vertices (3, 4)
(-2, 6): Non-trivial homology Z in dimension 0 of the full
subcomplex generated by a set of vertices (1, 2, 5)
(-2, 8): Non-trivial homology Z in dimension 1 of the full
subcomplex generated by a set of vertices (1, 3, 4, 5)
(-2, 8): Non-trivial homology Z in dimension 1 of the full
subcomplex generated by a set of vertices (2, 3, 4, 5)
(-3, 10): Non-trivial homology Z in dimension 1 of the full
```

(continues on next page)

(continued from previous page)

```
subcomplex generated by a set of vertices (1, 2, 3, 4, 5)
[((0, 0), 1), ((-1, 4), 3), ((-2, 8), 2), ((-2, 6), 1), ((-3, 10), 1)]
```

If we wish to view them in a form of a table, it is simple enough to create a function as such:

```
sage: def print_table(bbns):
....:     max_a = max(-p[0] for p in bbns)
....:     max_b = max(p[1] for p in bbns)
....:     bbn_table = [[bbns.get((-a,b), 0) for a in range(max_a+1)]
....:                                       for b in range(max_b+1)]
....:     width = len(str(max(bbns.values()))) + 1
....:     print(' '*width, end=' ')
....:     for i in range(max_a+1):
....:         print(f'{-i:{width}}', end=' ')
....:     print()
....:     for j in range(len(bbn_table)):
....:         print(f'{j:{width}}', end=' ')
....:         for r in bbn_table[j]:
....:             print(f'{r:{width}}', end=' ')
....:         print()
sage: print_table(X.bigraded_betti_numbers())
    0 -1 -2
 0  1  0  0
 1  0  0  0
 2  0  0  0
 3  0  0  0
 4  0  2  0
 5  0  0  0
 6  0  1  1
 7  0  0  0
 8  0  0  1
```

**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.

INPUT:

- `subcomplex` – a subcomplex of this simplicial complex (default: `None`); return faces which are not in this subcomplex

EXAMPLES:

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

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

The chain complex associated to this simplicial complex.

INPUT:

- `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` – commutative ring (default: `ZZ`)
- `subcomplex` – a subcomplex of this simplicial complex (default: empty); compute the chain complex relative to this subcomplex
- `augmented` – boolean (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 (default: `False`); if `True`, return the cochain complex (that is, the dual of the chain complex)
- `verbose` – boolean (default: `False`); if `True`, print some messages as the chain complex is computed
- `check` – boolean (default: `False`); if `True`, make sure that the chain complex is actually a chain complex: the differentials are composable and their product is zero

**Note**

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

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

EXAMPLES:

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

**cone**(*is_mutable=True*)

The cone on this simplicial complex.

INPUT:

- `is_mutable` – boolean (default: `True`); determines whether 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: CS = S.cone()
sage: sorted(CS.vertices())
['L0', 'L1', 'R0']
sage: len(CS.facets())
2
sage: CS.facets() == set([Simplex(['L0', 'R0']), Simplex(['L1', 'R0'])])
True
```

**cone_vertices**()

Return the list of cone vertices of `self`.

A vertex is a cone vertex if and only if it appears in every facet.

EXAMPLES:

```
sage: SimplicialComplex([[1,2,3]]).cone_vertices()
[1, 2, 3]
sage: SimplicialComplex([[1,2,3], [1,3,4], [1,5,6]]).cone_vertices()
[1]
sage: SimplicialComplex([[1,2,3], [1,3,4], [2,5,6]]).cone_vertices()
[]
```

**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: CL0 = X.connected_component(Simplex(['L0']))
sage: CR0 = X.connected_component(Simplex(['R0']))
sage: CL0 == CR0
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.

INPUT:

- `other` – another simplicial complex
- `is_mutable` – boolean (default: `True`); determine whether the output is mutable

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

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

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

**decone**()

Return the subcomplex of `self` induced by the non-cone vertices.

EXAMPLES:

```
sage: SimplicialComplex([[1,2,3]]).decone()
Simplicial complex with vertex set () and facets {()}
sage: SimplicialComplex([[1,2,3], [1,3,4], [1,5,6]]).decone()
Simplicial complex with vertex set (2, 3, 4, 5, 6)
 and facets {(2, 3), (3, 4), (5, 6)}
sage: X = SimplicialComplex([[1,2,3], [1,3,4], [2,5,6]])
sage: X.decone() == X
True
```

**delta_complex**(*sort_simplices=False*)

Return `self` as a $\Delta$-complex.

The $\Delta$-complex is essentially identical to the simplicial complex: it has same simplices with the same boundaries.

INPUT:

- `sort_simplices` – boolean (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
```

(continues on next page)

(continued from previous page)

```
sage: T.homology() == Td.homology()                                    #␣
↪needs sage.modules
True
```

**disjoint_union**(*right*, *rename_vertices=None*, *is_mutable=True*)

The disjoint union of this simplicial complex with another one.

INPUT:

- `right` – the other simplicial complex (the right-hand factor)
- `rename_vertices` – boolean (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()                                 #␣
↪needs sage.modules
{0: Z, 1: Z, 2: Z}
```

**f_triangle**()

Compute the $f$-triangle of `self`.

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

**See also**

Not to be confused with *F_triangle()* .

EXAMPLES:

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

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

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

**face**(*simplex*, *i*)

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

INPUT:

- `simplex` – a simplex in this simplicial complex
- `i` – integer

EXAMPLES:

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

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

**face_iterator**(*increasing=True*)

An iterator for the faces in this simplicial complex.

INPUT:

- `increasing` – boolean (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.

**Note**

Among the faces of a fixed dimension, there is no sorting.

EXAMPLES:

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

**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.

INPUT:

- `subcomplex` – a subcomplex of this simplicial complex (default: `None`); return faces which are not in this subcomplex

EXAMPLES:

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

**facets**()

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

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

EXAMPLES:

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

`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)])]); fix       #␣
↪needs sage.groups
Simplicial complex with vertex set (0, 2, 3, 4, 5) and 5 facets
sage: fix.is_isomorphic(S3)                                                 #␣
↪needs sage.groups
True
```

Another simple example:

```
sage: T = SimplicialComplex([[1,2,3],[2,3,4]])
sage: G = T.automorphism_group()                                            #␣
↪needs sage.groups
sage: T.fixed_complex([G([(1,4)])])                                         #␣
↪needs sage.groups
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()                                          #␣
↪needs sage.groups
sage: H = G.subgroup([G([(2,3),(5,6),(8,9)])])                              #␣
↪needs sage.groups
sage: CP2.fixed_complex(H).is_isomorphic(RP2)                               #␣
↪needs sage.groups
True
```

**flip_graph**()

If `self` is pure, return the flip graph of `self`, otherwise, return `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(sort=True); G.edges(sort=True, labels=False)
[(0,), (1,)]
[((0,), (1,))]

sage: G = (S0.wedge(S0)).flip_graph()
sage: G.vertices(sort=True); G.edges(sort=True, 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: len(G.vertices(sort=False))
6
sage: len(G.edges(sort=False))
10

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

sage: G = S2.flip_graph()
sage: G.vertices(sort=True); G.edges(sort=True, 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(sort=False)); len(G.edges(sort=False, labels=False))
46
161
```

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

Return the fundamental group of this simplicial complex.

INPUT:

- `base_point` – (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` – boolean (default: `True`); 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()                                                #␣
↪needs sage.groups
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); C2                        #␣
↪needs sage.groups
Finitely presented group < e0, e1, e2, e3, e4, e5, e6, e7, e8, e9 | e0, e3,
e4, e7, e9, e5*e2^-1*e0, e7*e2^-1*e1, e8*e3^-1*e1, e8*e6^-1*e4, e9*e6^-1*e5 >
sage: C2.simplified()                                                       #␣
↪needs sage.groups
Finitely presented group < e1 | e1^2 >
```

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

```
sage: RP2.fundamental_group()                                               #␣
↪needs sage.groups
Finitely presented group < e1 | e1^2 >
```

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

```
sage: # needs sage.groups
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: K.fundamental_group(base_point='L0')
Finitely presented group < e |  >
sage: K.fundamental_group(base_point='R0').order()
2
```

Some other examples:

```
sage: S1.wedge(S1).fundamental_group()                                      #␣
↪needs sage.groups
Finitely presented group < e0, e1 | >
sage: simplicial_complexes.Torus().fundamental_group()                      #␣
↪needs sage.groups
Finitely presented group < e1, e4 | e4^-1*e1^-1*e4*e1 >

sage: # needs sage.groups
sage: G = simplicial_complexes.MooreSpace(5).fundamental_group()
sage: G.ngens()
1
sage: x = G.gen(0)
sage: [(x**n).is_one() for n in range(1,6)]
[False, False, False, False, True]
```

**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: # needs sage.combinat
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*)

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

INPUT:

- `sub_vertex_set` – the sub-vertex set
- `is_mutable` – boolean (default: `True`); determine whether the output is mutable

EXAMPLES:

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

**graph**()

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

**Warning**

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

EXAMPLES:

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

**h_triangle**()

Compute the $h$-triangle of `self`.

The $h$-triangle of a simplicial complex $\Delta$ is given by

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

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

EXAMPLES:

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

**h_vector**()

The $h$-vector of this simplicial complex.

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

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

Alternatively,

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

EXAMPLES:

The $f$- and $h$-vectors of the boundary of an octahedron are computed in Wikipedia article 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]
```

**intersection**(*other*)

Calculate the intersection of two simplicial complexes.

EXAMPLES:

```
sage: X = SimplicialComplex([[1,2,3], [1,2,4]])
sage: Y = SimplicialComplex([[1,2,3], [1,4,5]])
sage: Z = SimplicialComplex([[1,2,3], [1,4], [2,4]])
sage: sorted(X.intersection(Y).facets())
[(1, 2, 3), (1, 4)]
sage: X.intersection(X) == X
True
sage: X.intersection(Z) == X
False
sage: X.intersection(Z) == Z
True
```

**is_balanced**(*check_purity=False*, *certificate=False*)

Determine whether `self` is balanced.

A simplicial complex $X$ of dimension $d-1$ is balanced if and only if its vertices can be colored with $d$ colors such that every face contains at most one vertex of each color. An equivalent condition is that the 1-skeleton

of $X$ is $d$-colorable. In some contexts, it is also required that $X$ be pure (i.e., that all maximal faces of $X$ have the same dimension).

INPUT:

- `check_purity` – boolean (default: `False`); if this is `True`, require that `self` be pure as well as balanced
- `certificate` – boolean (default: `False`); if this is `True` and `self` is balanced, then return a $d$-coloring of the 1-skeleton

EXAMPLES:

A 1-dim simplicial complex is balanced iff it is bipartite:

```
sage: X = SimplicialComplex([[1,2], [1,4], [3,4], [2,5]])
sage: X.is_balanced()
True
sage: sorted(X.is_balanced(certificate=True))
[[1, 3, 5], [2, 4]]
sage: X = SimplicialComplex([[1,2], [1,4], [3,4], [2,4]])
sage: X.is_balanced()
False
```

Any barycentric division is balanced:

```
sage: X = SimplicialComplex([[1,2,3], [1,2,4], [2,3,4]])
sage: X.is_balanced()
False
sage: X.barycentric_subdivision().is_balanced()
True
```

A non-pure balanced complex:

```
sage: X = SimplicialComplex([[1,2,3], [3,4]])
sage: X.is_balanced(check_purity=True)
False
sage: sorted(X.is_balanced(certificate=True))
[[1, 4], [2], [3]]
```

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

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

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

INPUT:

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

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

EXAMPLES:

Spheres are Cohen-Macaulay:

```
sage: S = SimplicialComplex([[1,2],[2,3],[3,1]])
sage: S.is_cohen_macaulay(ncpus=3)                                          #␣
↪needs sage.modules
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)                                          #␣
↪needs sage.modules
False
```

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

```
sage: X = simplicial_complexes.RealProjectivePlane()
sage: X.is_cohen_macaulay()                                                 #␣
↪needs sage.modules
True
sage: X.is_cohen_macaulay(ZZ)                                               #␣
↪needs sage.modules
False
```

**is_flag_complex**()

Return `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(); 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_golod**()

Return whether `self` is Golod.

A simplicial complex is Golod if multiplication and all higher Massey operations in the associated Tor-algebra are trivial. This is done by checking the bigraded Betti numbers.

EXAMPLES:

```
sage: # needs sage.modules
sage: X = SimplicialComplex([[0,1],[1,2],[2,3],[3,0]])
sage: Y = SimplicialComplex([[0,1,2],[0,2],[0,4]])
sage: X.is_golod()
False
sage: Y.is_golod()
True
```

**is_immutable**()

Return `True` if immutable.

EXAMPLES:

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

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

Check whether two simplicial complexes are isomorphic.

INPUT:

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

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

EXAMPLES:

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

We check that Issue #20751 is fixed:

```
sage: C1 = SimplicialComplex([[1,2,3], [2,4], [3,5], [5,6]])
sage: C2 = SimplicialComplex([['a','b','c'], ['b','d'], ['c','e'], ['e','f']])
sage: C1.is_isomorphic(C2, certificate=True)
(True, {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 6: 'f'})
```

**is_minimally_non_golod**()

Return whether `self` is minimally non-Golod.

If a simplicial complex itself is not Golod, but deleting any vertex gives us a full subcomplex that is Golod, then we say that a simplicial complex is minimally non-Golod.

**See also**

See *is_golod()* for more information.

EXAMPLES:

```
sage: # needs sage.modules
sage: X = SimplicialComplex([[0,1],[1,2],[2,3],[3,0]])
sage: Y = SimplicialComplex([[1,2,3],[1,2,4],[3,5],[4,5]])
sage: X.is_golod()
False
```

(continues on next page)

(continued from previous page)

```
sage: X.is_minimally_non_golod()
True
sage: Y.is_golod()
False
sage: Y.is_minimally_non_golod()
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_partitionable**(*certificate*, *solver*, *integrality_tolerance=False*)

Determine whether `self` is partitionable.

A partitioning of a simplicial complex $X$ is a decomposition of its face poset into disjoint Boolean intervals $[R, F]$, where $F$ ranges over all facets of $X$.

The method sets up an integer program with:

- a variable $y_i$ for each pair $(R, F)$, where $F$ is a facet of $X$ and $R$ is a subface of $F$
- a constraint $y_i + y_j \leq 1$ for each pair $(R_i, F_i)$, $(R_j, F_j)$ whose Boolean intervals intersect nontrivially (equivalent to $(R_i \subseteq F_j and R_j \subseteq F_i)$)
- objective function equal to the sum of all $y_i$

INPUT:

- `certificate` – boolean (default: `False`); if `True`, and `self` is partitionable, then return a list of pairs $(R, F)$ that form a partitioning.
- `solver` – (default: `None`) specifies a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`

EXAMPLES:

Simplices are trivially partitionable:

```
sage: X = SimplicialComplex([[1,2,3,4]])
sage: X.is_partitionable()                                                  #␣
↪needs sage.numerical.mip
True
```

(continues on next page)

(continued from previous page)

```
sage: X.is_partitionable(certificate=True)                                    #␣
↪needs sage.numerical.mip
[((), (1, 2, 3, 4), 4)]
```

Shellable complexes are partitionable:

```
sage: # needs sage.numerical.mip, long time
sage: X = SimplicialComplex([[1,3,5], [1,3,6], [1,4,5], [1,4,6],
....:                        [2,3,5], [2,3,6], [2,4,5]])
sage: X.is_partitionable()
True
sage: P = X.is_partitionable(certificate=True)
sage: def n_intervals_containing(f):
....:     return len([RF for RF in P
....:                 if RF[0].is_face(f) and f.is_face(RF[1])])
sage: all(n_intervals_containing(f) == 1
....:     for k in X.faces().keys() for f in X.faces()[k])
True
```

A non-shellable, non-Cohen-Macaulay, partitionable example, constructed by Björner:

```
sage: X = SimplicialComplex([[1,2,3], [1,2,4], [1,3,4], [2,3,4], [1,5,6]])
sage: X.is_partitionable()                                                    #␣
↪needs sage.numerical.mip
True
```

The bowtie complex is not partitionable:

```
sage: X = SimplicialComplex([[1,2,3], [1,4,5]])
sage: X.is_partitionable()                                                    #␣
↪needs sage.numerical.mip
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)
```

(continues on next page)

(continued from previous page)

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

**is_pure**()

Return `True` iff this simplicial complex is pure.

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

**Warning**

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

EXAMPLES:

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

Demonstration of the warning:

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

**is_shellable**(*certificate=False*)

Return if `self` is shellable.

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

**Note**

1. This method can check all orderings of the facets by brute force, hence can be very slow.
2. This is shellability in the general (nonpure) sense of Bjorner and Wachs [BW1996]. This method does not check purity.

**See also**

*is_shelling_order()*

INPUT:

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

EXAMPLES:

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

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

Examples from Figure 1 in [BW1996]:

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

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

REFERENCES:

- Wikipedia article Shelling_(topology)

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

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

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

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

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

INPUT:

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

**See also**

*is_shellable()*

EXAMPLES:

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

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

A non-pure example:

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

REFERENCES:

- [BW1996]

**is_subcomplex**(*other*)

Return `True` if this is a subcomplex of `other`.

INPUT:

- `other` – another simplicial complex

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: S1.is_subcomplex(S1)
True
sage: Empty = SimplicialComplex()
sage: Empty.is_subcomplex(S1)
True
sage: S1.is_subcomplex(Empty)
False

sage: sorted(S1.facets())
[(0, 1), (0, 2), (1, 2)]
sage: T = S1.product(S1)
sage: sorted(T.facets())[0] # typical facet in T
('L0R0', 'L0R1', 'L1R1')
sage: S1.is_subcomplex(T)
False
sage: T._contractible_subcomplex().is_subcomplex(T)
True
```

**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 $T$.

INPUT:

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

- `rename_vertices` – boolean (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 (default: `True`); determine whether 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 {(0, 2), (0, 3), (1, 2), (1, 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.

INPUT:

- `simplex` – a simplex in this simplicial complex
- `is_mutable` – boolean (default: `True`); determine whether 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 {(0,), (3,)}
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: sorted(Y.maximal_faces())
[(0, 2), (1, 4)]
```

`facets` is a synonym for `maximal_faces`:

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

**minimal_nonfaces**()

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

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

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

EXAMPLES:

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

**moment_angle_complex**()

Return the moment-angle complex of `self`.

A moment-angle complex is a topological space created from this simplicial complex, which holds a lot of information about the simplicial complex itself.

**See also**

See *sage.topology.moment_angle_complex* for more information on moment-angle complexes.

EXAMPLES:

```
sage: X = SimplicialComplex([[0,1,2,3], [1,4], [3,2,4]])
sage: X.moment_angle_complex()
Moment-angle complex of Simplicial complex with vertex set
(0, 1, 2, 3, 4) and facets {(1, 4), (2, 3, 4), (0, 1, 2, 3)}
sage: K = simplicial_complexes.KleinBottle()
sage: K.moment_angle_complex()
Moment-angle complex of Simplicial complex with vertex set
(0, 1, 2, 3, 4, 5, 6, 7) and 16 facets
```

We can also create it explicitly:

```
sage: Z = MomentAngleComplex(K); Z
Moment-angle complex of Simplicial complex with vertex set
(0, 1, 2, 3, 4, 5, 6, 7) and 16 facets
```

**n_faces**(*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 cells which are *not* in the subcomplex.

INPUT:

- `n` – nonnegative integer; the dimension

- `subcomplex` – (optional) a subcomplex of this cell complex; return the cells which are not in this subcomplex

**Note**

The resulting list need not be sorted. If you want a sorted list of $n$-cells, use `_n_cells_sorted()`.

EXAMPLES:

```
sage: delta_complexes.Torus().n_cells(1)
[(0, 0), (0, 0), (0, 0)]
sage: cubical_complexes.Cube(1).n_cells(0)
[[1,1], [0,0]]
```

**n_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 $n$.

INPUT:

- `n` – nonnegative 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 {(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}
sage: X.set_immutable()
sage: X.n_skeleton(2)
Simplicial complex with vertex set (0, 1, 2, 3) and
 facets {(0, 1), (0, 2, 3), (1, 2, 3)}
sage: X.n_skeleton(4)
Simplicial complex with vertex set (0, 1, 2, 3) and
 facets {(0, 1), (0, 2, 3), (1, 2, 3)}
```

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

The product of this simplicial complex with another one.

INPUT:

- `right` – the other simplicial complex (the right-hand factor)
- `rename_vertices` – boolean (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 (default: `True`); determines whether 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: Cyl = S.product(K)  # cylinder
sage: sorted(Cyl.vertices())
['L0R0', 'L0R1', 'L1R0', 'L1R1', 'L2R0', 'L2R1']
sage: Cyl2 = S.product(K, rename_vertices=False)
sage: sorted(Cyl2.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()                                                          #␣
↪needs sage.modules
{0: 0, 1: Z x Z, 2: Z}
```

These can get large pretty quickly:

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

**remove_face**(*face*, *check=False*)

Remove a face from this simplicial complex.

INPUT:

- `face` – a face of the simplicial complex
- `check` – boolean (default: `False`); if `True`, raise an error if `face` is not a face of this simplicial complex

This does not return anything; instead, it *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]])
```

(continues on next page)

(continued from previous page)

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

**remove_faces** (*faces*, *check=False*)

Remove a collection of faces from this simplicial complex.

INPUT:

- `faces` – list (or any iterable) of faces of the simplicial complex
- `check` – boolean (default: `False`); if `True`, raise an error if any element of `faces` is not a face of this simplicial complex

This does not return anything; instead, it *changes* the simplicial complex.

ALGORITHM:

Run `self.remove_face(f)` repeatedly, for `f` in `faces`.

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_faces([[1,2]])
sage: Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 3, 4), (2, 3,␣
↪4)}

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

**restriction_sets** (*order*)

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

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

**See also**

*is_shelling_order()*

EXAMPLES:

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

(continues on next page)

(continued from previous page)

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

**set_immutable**()

Make this simplicial complex immutable.

EXAMPLES:

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

**stanley_reisner_ring**(*base_ring=Integer Ring*)

The Stanley-Reisner ring of this simplicial complex.

INPUT:

- `base_ring` – a commutative ring (default: `ZZ`)

OUTPUT: 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 nonnegative 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([[0,1,2], [0,2,3]])
sage: X.stanley_reisner_ring()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring
 by the ideal (x1*x3)
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
```

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

Return the star of a simplex in this simplicial complex.

The star of `simplex` is the simplicial complex formed by all simplices which contain `simplex`.

INPUT:

- `simplex` – a simplex in this simplicial complex
- `is_mutable` – boolean (default: `True`); determines if the output is mutable

EXAMPLES:

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

**stellar_subdivision**(*simplex*, *inplace=False*, *is_mutable=True*)

Return the stellar subdivision of a simplex in this simplicial complex.

The stellar subdivision of a face is obtained by adding a new vertex to the simplicial complex `self` joined to the star of the face and then deleting the face `simplex` to the result.

INPUT:

- `simplex` – a simplex face of `self`
- `inplace` – boolean (default: `False`); determines if the operation is done on `self` or on a copy
- `is_mutable` – boolean (default: `True`); determines if the output is mutable

OUTPUT:

- A simplicial complex obtained by the stellar subdivision of the face `simplex`

EXAMPLES:

```
sage: SC = SimplicialComplex([[0,1,2],[1,2,3]])
sage: F1 = Simplex([1,2])
sage: F2 = Simplex([1,3])
sage: F3 = Simplex([1,2,3])
sage: SC.stellar_subdivision(F1)
Simplicial complex with vertex set (0, 1, 2, 3, 4) and
 facets {(0, 1, 4), (0, 2, 4), (1, 3, 4), (2, 3, 4)}
sage: SC.stellar_subdivision(F2)
Simplicial complex with vertex set (0, 1, 2, 3, 4) and
 facets {(0, 1, 2), (1, 2, 4), (2, 3, 4)}
sage: SC.stellar_subdivision(F3)
Simplicial complex with vertex set (0, 1, 2, 3, 4) and
 facets {(0, 1, 2), (1, 2, 4), (1, 3, 4), (2, 3, 4)}
sage: SC.stellar_subdivision(F3, inplace=True);SC
Simplicial complex with vertex set (0, 1, 2, 3, 4) and
 facets {(0, 1, 2), (1, 2, 4), (1, 3, 4), (2, 3, 4)}
```

The simplex to subdivide should be a face of self:

```
sage: SC = SimplicialComplex([[0,1,2],[1,2,3]])
sage: F4 = Simplex([3,4])
sage: SC.stellar_subdivision(F4)
Traceback (most recent call last):
...
ValueError: the face to subdivide is not a face of self
```

One can not modify an immutable simplicial complex:

```
sage: SC = SimplicialComplex([[0,1,2],[1,2,3]], is_mutable=False)
sage: SC.stellar_subdivision(F1, inplace=True)
Traceback (most recent call last):
...
ValueError: this simplicial complex is not mutable
```

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

The suspension of this simplicial complex.

INPUT:

- `n` – positive integer (default: 1); suspend this many times
- `is_mutable` – boolean (default: `True`); determine whether the output is mutable

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

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

- $u * \alpha$ where $\alpha$ is a facet of $M$ not containing $u$
- $w * \beta$ where $\beta$ is any facet of $M$.

EXAMPLES:

```
sage: S0 = SimplicialComplex([[0], [1]])
sage: S0.suspension() == simplicial_complexes.Sphere(1)
True
sage: S3 = S0.suspension(3)  # the 3-sphere
sage: S3.homology()                                                          #␣
↪needs sage.modules
{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: sorted(T.join(S0).vertices())       # 9 vertices
['L0', 'L1', 'L2', 'L3', 'L4', 'L5', 'L6', 'R0', 'R1']
sage: T.suspension().vertices()  # 8 vertices
(0, 1, 2, 3, 4, 5, 6, 7)
```

**vertices**()

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

EXAMPLES:

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

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

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

INPUT:

- `right` – the other simplicial complex (the right-hand factor)
- `rename_vertices` – boolean (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 (default: `True`); determine whether 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()                                                  #␣
↪needs sage.modules
{0: 0, 1: Z, 2: Z}
```

sage.topology.simplicial_complex.**facets_for_K3**()

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

sage.topology.simplicial_complex.**facets_for_RP4**()

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

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

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

EXAMPLES:

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

sage.topology.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.

INPUT:

- `t1` – list or other iterable; labeling for vertices
- `t2` – list or other iterable; labeling for vertices
- `length` – integer or `None` (default: `None`); if not `None`, then an integer, the length of the desired path

OUTPUT: list of lists of vertices making up the paths

This is used when triangulating the product of simplices. The optional argument `length` is used for $\Delta$-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.topology.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.topology.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`.

INPUT:

- `n` – a 'vertex'; either an integer or a string
- `keep` – list of three vertices
- `left` – boolean (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.topology.simplicial_complex import rename_vertex
sage: rename_vertex(6, [5, 6, 7])
1
sage: rename_vertex(3, [5, 6, 7, 8, 9])
'L3'
sage: rename_vertex(3, [5, 6, 7], left=False)
'R3'
```

CHAPTER

TWO

# MORPHISMS OF SIMPLICIAL COMPLEXES

AUTHORS:

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

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

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

EXAMPLES:

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

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

(continues on next page)

(continued from previous page)

```
sage: i.is_identity()
True

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

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

Bases: Morphism

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

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

Return the associated chain complex morphism of `self`.

EXAMPLES:

```
sage: # needs sage.modules
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); x
```

(continues on next page)

(continued from previous page)

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

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

```
sage: # needs sage.modules
sage: g = {0:1, 1:2, 2:0}
sage: H(g).associated_chain_complex_morphism()._matrix_dictionary
{0: [0 0 1]
    [1 0 0]
    [0 1 0]
    [0 0 0],
 1: [ 0 -1  0]
    [ 0  0 -1]
    [ 0  0  0]
    [ 1  0  0]
    [ 0  0  0]
    [ 0  0  0],
 2: []}
```

(continues on next page)

(continued from previous page)

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

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

Fiber product of `self` and `other`.

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

EXAMPLES:

```
sage: S = SimplicialComplex([[0,1],[1,2]], is_mutable=False)
sage: T = SimplicialComplex([[0,2],[1]], is_mutable=False)
sage: U = SimplicialComplex([[0,1],[2]], is_mutable=False)
sage: H = Hom(S,U)
sage: G = Hom(T,U)
sage: f = {0:0,1:1,2:0}
sage: g = {0:0,1:1,2:1}
sage: x = H(f)
sage: y = G(g)
sage: z = x.fiber_product(y)
sage: z
Simplicial complex morphism:
  From: Simplicial complex with 4 vertices and facets {...}
  To:   Simplicial complex with vertex set (0, 1, 2) and facets {(2,), (0, 1)}
  Defn: L0R0 |--> 0
        L1R1 |--> 1
        L1R2 |--> 1
        L2R0 |--> 0
```

**image**()

Compute 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}
```

(continues on next page)

(continued from previous page)

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

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

Return the map in (co)homology induced by this map.

INPUT:

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

EXAMPLES:

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

We can view the matrix form for the homomorphism:

```
sage: h.to_matrix(0)  # in degree 0                                            #␣
↪needs sage.modules
[1]
sage: h.to_matrix(1)  # in degree 1                                            #␣
↪needs sage.modules
[1]
[1]
sage: h.to_matrix()   # the entire homomorphism                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
[1|0]
[-+-]
[0|1]
[0|1]
[-+-]
[0|0]
```

The map on cohomology should be dual to the map on homology:

```
sage: coh = diag.induced_homology_morphism(QQ, cohomology=True)                #␣
↪needs sage.modules
sage: coh.to_matrix(1)                                                         #␣
↪needs sage.modules
[1 1]
sage: h.to_matrix() == coh.to_matrix().transpose()                             #␣
↪needs sage.modules
True
```

We can evaluate the map on (co)homology classes:

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

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

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

**is_contiguous_to**(*other*)

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

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

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

See Section 3.5 of Spanier [Spa1966] for details.

ALGORITHM:

It is enough to check when $\sigma$ ranges over the facets.

INPUT:

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

EXAMPLES:

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

**is_identity**()

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

EXAMPLES:

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

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

**is_injective**()

Return `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**()

Return `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()              #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
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()          #␣
↪needs sage.modules
True
```

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

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

EXAMPLES:

```
sage: from sage.topology.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)
doctest:warning...
DeprecationWarning: The function is_SimplicialComplexMorphism is deprecated;
use 'isinstance(..., SimplicialComplexMorphism)' instead.
See https://github.com/sagemath/sage/issues/38103 for details.
True
```

CHAPTER

THREE

# HOMSETS BETWEEN SIMPLICIAL COMPLEXES

AUTHORS:

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

EXAMPLES:

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

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

Bases: Homset

**an_element**()

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

EXAMPLES:

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

(continues on next page)

(continued from previous page)

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

**diagonal_morphism**(*rename_vertices=True*)

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

EXAMPLES:

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

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

**identity**()

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

EXAMPLES:

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

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

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

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

EXAMPLES:

```
sage: S = SimplicialComplex(is_mutable=False)
sage: T = SimplicialComplex(is_mutable=False)
sage: H = Hom(S, T)
sage: H
Set of Morphisms from Simplicial complex with vertex set () and facets {()}
 to Simplicial complex with vertex set () and facets {()}
 in Category of finite simplicial complexes
sage: from sage.topology.simplicial_complex_homset import is_
↪SimplicialComplexHomset
sage: is_SimplicialComplexHomset(H)
doctest:warning...
DeprecationWarning: the function is_SimplicialComplexHomset is deprecated;
use 'isinstance(..., SimplicialComplexHomset)' instead
See https://github.com/sagemath/sage/issues/37922 for details.
True
```

CHAPTER

FOUR

# EXAMPLES OF SIMPLICIAL COMPLEXES

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

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

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

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

- *BarnetteSphere()*
- *BrucknerGrunbaumSphere()*
- *ChessboardComplex()*
- *ComplexProjectivePlane()*
- *DunceHat()*
- *FareyMap()*
- *GenusSix()*
- *K3Surface()*
- *KleinBottle()*
- *MatchingComplex()*
- *MooreSpace()*
- *NotIConnectedGraphs()*
- *PoincareHomologyThreeSphere()*
- *QuaternionicProjectivePlane()*
- *RandomComplex()*
- *RandomTwoSphere()*
- *RealProjectivePlane()*
- *RealProjectiveSpace()*
- *RudinBall()*
- *ShiftedComplex()*

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

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

EXAMPLES:

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

sage.topology.simplicial_complex_examples.**BarnetteSphere**()

Return Barnette's triangulation of the 3-sphere.

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

EXAMPLES:

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

sage.topology.simplicial_complex_examples.**BrucknerGrunbaumSphere**()

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

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

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

EXAMPLES:

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

sage.topology.simplicial_complex_examples.**ChessboardComplex**(*n*, *i*)

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

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

INPUT:

- `n`, `i` – positive integers

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

EXAMPLES:

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

sage.topology.simplicial_complex_examples.**ComplexProjectivePlane**()

A minimal triangulation of the complex projective plane.

This was constructed by Kühnel and Banchoff [KB1983].

EXAMPLES:

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

sage.topology.simplicial_complex_examples.**DunceHat**()

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

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

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
True
```

sage.topology.simplicial_complex_examples.**FareyMap**(*p*)

Return a discrete surface associated with $PSL(2, \mathbf{F}_(p))$.

INPUT:

- p – a prime number

The vertices are the nonzero pairs $(x, y)$ in $\mathbf{F}_(p)^2$ modulo the identification of $(-x, -y)$ with $(x, y)$.

The triangles are the images of the base triangle ((1,0),(0,1),(1,1)) under the action of $PSL(2, \mathbf{F}_(p))$.

For $p = 3$, the result is a tetrahedron, for $p = 5$ an icosahedron, and for $p = 7$ a triangulation of the Klein quartic of genus 3.

As a Riemann surface, this is the quotient of the upper half plane by the principal congruence subgroup $\Gamma(p)$.

EXAMPLES:

```
sage: S5 = simplicial_complexes.FareyMap(5); S5                                #␣
↪needs sage.groups
Simplicial complex with 12 vertices and 20 facets
sage: S5.automorphism_group().cardinality()                                    #␣
↪needs sage.groups
120

sage: S7 = simplicial_complexes.FareyMap(7); S7                                #␣
↪needs sage.groups
Simplicial complex with 24 vertices and 56 facets
sage: S7.f_vector()                                                            #␣
↪needs sage.groups
[1, 24, 84, 56]
```

REFERENCES:

- [ISS2019] Ioannis Ivrissimtzis, David Singerman and James Strudwick, *From Farey Fractions to the Klein Quartic and Beyond*. arXiv 1909.08568

sage.topology.simplicial_complex_examples.**GenusSix**()

Return a triangulated surface of genus 6.

This is triangulated with 12 vertices, 66 edges and 44 faces. Each vertex is neighbour to all other vertices.

It appears as number 58 in the classification of Altshuler, Bokowski and Schuchert in [ABS96], where it is the unique surface with the largest symmetry group, of order 12. This article refers for this surface to Ringel.

EXAMPLES:

```
sage: S = simplicial_complexes.GenusSix()
sage: S.automorphism_group().cardinality()                                     #␣
↪needs sage.groups
12
sage: S.betti()                                                                #␣
↪needs sage.modules
{0: 1, 1: 12, 2: 1}
sage: S.f_vector()
[1, 12, 66, 44]
```

REFERENCES:

sage.topology.simplicial_complex_examples.**K3Surface**()

Return a minimal triangulation of the K3 surface.

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

EXAMPLES:

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

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

sage.topology.simplicial_complex_examples.**KleinBottle**()

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

EXAMPLES:

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

sage.topology.simplicial_complex_examples.**MatchingComplex**($n$)

The matching complex of graphs on $n$ vertices.

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

INPUT:

- `n` – positive integer

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

EXAMPLES:

```
sage: M = simplicial_complexes.MatchingComplex(7)
sage: H = M.homology(); H                                                    #␣
↪needs sage.modules
{0: 0, 1: C3, 2: Z^20}
sage: H[2].ngens()                                                           #␣
↪needs sage.modules
20
sage: M8 = simplicial_complexes.MatchingComplex(8)
sage: M8.homology(2)                    # long time (6s on sage.math, 2012),␣
↪needs sage.modules
Z^132
```

sage.topology.simplicial_complex_examples.**MooreSpace**($q$)

Triangulation of the mod $q$ Moore space.

INPUT:

- q – 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
$ZZ/q$
$ZZ$ in dimension 1, zero otherwise.

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

EXAMPLES:

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

sage.topology.simplicial_complex_examples.**NotIConnectedGraphs**(*n*, *i*)

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

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

INPUT:

- n, i – nonnegative 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: NICG52 = simplicial_complexes.NotIConnectedGraphs(5, 2)
sage: NICG52.f_vector()
[1, 10, 45, 120, 210, 240, 140, 20]
sage: NICG52.homology(5).ngens()                                                #␣
↪needs sage.modules
6
```

sage.topology.simplicial_complex_examples.**PoincareHomologyThreeSphere**()

A triangulation of the Poincaré homology 3-sphere.

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

EXAMPLES:

```
sage: S3 = simplicial_complexes.Sphere(3)
sage: Sigma3 = simplicial_complexes.PoincareHomologyThreeSphere()
sage: S3.homology() == Sigma3.homology()                                    #␣
↪needs sage.modules
True
sage: Sigma3.fundamental_group().cardinality()  # long time                 #␣
↪needs sage.groups
120
```

sage.topology.simplicial_complex_examples.**ProjectivePlane**()

A minimal triangulation of the real projective plane.

EXAMPLES:

```
sage: P = simplicial_complexes.RealProjectivePlane()
sage: Q = simplicial_complexes.ProjectivePlane()
sage: P == Q
True

sage: # needs sage.modules
sage: P.cohomology(1)
0
sage: P.cohomology(2)
C2
sage: P.cohomology(1, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
sage: P.cohomology(2, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
```

sage.topology.simplicial_complex_examples.**QuaternionicProjectivePlane**()

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

**Warning**

This was proven to be a triangulation of the projective plane $HP^2$ over the ring of quaternions by Gorodkov in 2016 [Gor2016].

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

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

EXAMPLES:

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

Checking its automorphism group:

```
sage: HP2.automorphism_group().is_isomorphic(AlternatingGroup(5))                #␣
↪needs sage.groups
True
```

sage.topology.simplicial_complex_examples.**RandomComplex**(*n*, *d*, *p=0.5*)

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

INPUT:

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

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

EXAMPLES:

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

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

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

sage.topology.simplicial_complex_examples.**RandomTwoSphere**(*n*)

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

INPUT:

- `n` – integer

OUTPUT:

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

ALGORITHM:

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

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

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

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

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

EXAMPLES:

```
sage: G = simplicial_complexes.RandomTwoSphere(6); G
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 8 facets
sage: G.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z}
sage: G.is_pure()
True
sage: fg = G.flip_graph(); fg
Graph on 8 vertices
sage: fg.is_planar() and fg.is_regular(3)
True
```

sage.topology.simplicial_complex_examples.**RealProjectivePlane**()

A minimal triangulation of the real projective plane.

EXAMPLES:

```
sage: P = simplicial_complexes.RealProjectivePlane()
sage: Q = simplicial_complexes.ProjectivePlane()
sage: P == Q
True

sage: # needs sage.modules
sage: P.cohomology(1)
0
sage: P.cohomology(2)
C2
sage: P.cohomology(1, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
sage: P.cohomology(2, base_ring=GF(2))
Vector space of dimension 1 over Finite Field of size 2
```

sage.topology.simplicial_complex_examples.**RealProjectiveSpace**(*n*)

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

INPUT:

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

The first few cases are pretty trivial:

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

ALGORITHM: For $n < 4$, these are constructed explicitly by listing the facets. For $n = 4$, this is constructed by specifying 16 vertices, two facets, and a certain subgroup $G$ of the symmetric group $S_{16}$. Then the set of all facets

is the $G$-orbit of the two given facets. This is implemented here by explicitly listing all of the facets; the facets can be computed by the function `facets_for_RP4()`, but running the function takes a few seconds.

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

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

EXAMPLES:

```
sage: P3 = simplicial_complexes.RealProjectiveSpace(3)
sage: P3.f_vector()
[1, 11, 51, 80, 40]
sage: P3.homology()                                                          #␣
↪needs sage.modules
{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.

```
sage: P5.homology()  # not tested
{0: 0, 1: C2, 2: 0, 3: C2, 4: 0, 5: Z}
sage: simplicial_complexes.RealProjectiveSpace(2).dimension()
2
sage: P3.dimension()
3
sage: P4.dimension() # long time
4
sage: P5.dimension() # long time
5
```

`sage.topology.simplicial_complex_examples.`**`RudinBall`**`()`

Return the non-shellable ball constructed by Rudin.

This complex is a non-shellable triangulation of the 3-ball with 14 vertices and 41 facets, constructed by Rudin in [Rud1958].

EXAMPLES:

```
sage: R = simplicial_complexes.RudinBall(); R
Rudin ball
sage: R.f_vector()
[1, 14, 66, 94, 41]
sage: R.homology()                                                           #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0}
```

(continues on next page)

(continued from previous page)

```
sage: R.is_cohen_macaulay()                                                 #␣
↪needs sage.modules
True
```

sage.topology.simplicial_complex_examples.**ShiftedComplex**(*generators*)

Return the smallest shifted simplicial complex containing `generators` as faces.

Let $V$ be a set of vertices equipped with a total order. The 'componentwise partial ordering' on k-subsets of $V$ is defined as follows: if $A = \{a_1 < \cdots < a_k\}$ and $B = \{b_1 < \cdots < b_k\}$, then $A \leq_C B$ iff $a_i \leq b_i$ for all $i$. A simplicial complex $X$ on vertex set $[n]$ is *shifted* if its faces form an order ideal under the componentwise partial ordering, i.e., if $B \in X$ and $A \leq_C B$ then $A \in X$. Shifted complexes of dimension 1 are also known as threshold graphs.

**Note**

This method assumes that $V$ consists of positive integers with the natural ordering.

INPUT:

- `generators` – list of generators of the order ideal, which may be lists, tuples or simplices

EXAMPLES:

```
sage: # needs sage.combinat
sage: X = simplicial_complexes.ShiftedComplex([Simplex([1, 6]), (2, 4), [8]])
sage: sorted(X.facets())
[(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 3), (2, 4), (7,), (8,)]
sage: X = simplicial_complexes.ShiftedComplex([[2, 3, 5]])
sage: sorted(X.facets())
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (2, 3, 4), (2, 3, 5)]
sage: X = simplicial_complexes.ShiftedComplex([[1, 3, 5], [2, 6]])
sage: sorted(X.facets())
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 6), (2, 6)]
```

sage.topology.simplicial_complex_examples.**Simplex**(*n*)

An $n$-dimensional simplex, as a simplicial complex.

INPUT:

- `n` – nonnegative integer

OUTPUT: the simplicial complex consisting of the $n$-simplex on vertices $(0, 1, ..., n)$ and all of its faces.

EXAMPLES:

```
sage: simplicial_complexes.Simplex(3)
The 3-simplex
sage: simplicial_complexes.Simplex(5).euler_characteristic()
1
```

sage.topology.simplicial_complex_examples.**Sphere**(*n*)

A minimal triangulation of the $n$-dimensional sphere.

INPUT:

- `n` – positive integer

EXAMPLES:

```
sage: simplicial_complexes.Sphere(2)
Minimal triangulation of the 2-sphere
sage: simplicial_complexes.Sphere(5).homology()                                 #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: Z}
sage: [simplicial_complexes.Sphere(n).euler_characteristic() for n in range(6)]
[2, 0, 2, 0, 2, 0]
sage: [simplicial_complexes.Sphere(n).f_vector() for n in range(6)]
[[1, 2],
 [1, 3, 3],
 [1, 4, 6, 4],
 [1, 5, 10, 10, 5],
 [1, 6, 15, 20, 15, 6],
 [1, 7, 21, 35, 35, 21, 7]]
```

sage.topology.simplicial_complex_examples.**SumComplex**(*n*, *A*)

The sum complexes of Linial, Meshulam, and Rosenthal [LMR2010].

If $k+1$ is the cardinality of $A$, then this returns a $k$-dimensional simplicial complex $X_A$ with vertices $\mathbf{Z}/(n)$, and facets given by all $k+1$-tuples $(x_0, x_1, ..., x_k)$ such that the sum $\sum x_i$ is in $A$. See the paper by Linial, Meshulam, and Rosenthal [LMR2010], in which they prove various results about these complexes; for example, if $n$ is prime, then $X_A$ is rationally acyclic, and if in addition $A$ forms an arithmetic progression in $\mathbf{Z}/(n)$, then $X_A$ is $\mathbf{Z}$-acyclic. Throughout their paper, they assume that $n$ and $k$ are relatively prime, but the construction makes sense in general.

In addition to the results from the cited paper, these complexes can have large torsion, given the number of vertices; for example, if $n = 10$, and $A = \{0, 1, 2, 3, 6\}$, then $H_3(X_A)$ is cyclic of order 2728, and there is a 4-dimensional complex on 13 vertices with $H_3$ having a cyclic summand of order

$$706565607945 = 3 \cdot 5 \cdot 53 \cdot 79 \cdot 131 \cdot 157 \cdot 547.$$

See the examples.

INPUT:

- `n` – positive integer
- `A` – a subset of $\mathbf{Z}/(n)$

EXAMPLES:

```
sage: S = simplicial_complexes.SumComplex(10, [0, 1, 2, 3, 6]); S
Sum complex on vertices Z/10Z associated to {0, 1, 2, 3, 6}
sage: S.homology()                                                              #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: C2728, 4: 0}
sage: factor(2728)
2^3 * 11 * 31

sage: S = simplicial_complexes.SumComplex(11, [0, 1, 3]); S
Sum complex on vertices Z/11Z associated to {0, 1, 3}
sage: S.homology(1)                                                             #␣
↪needs sage.modules
C23
sage: S = simplicial_complexes.SumComplex(11, [0, 1, 2, 3, 4, 7]); S
Sum complex on vertices Z/11Z associated to {0, 1, 2, 3, 4, 7}
sage: S.homology()                          # long time                         #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0, 4: C645679, 5: 0}
sage: factor(645679)
```

(continues on next page)

(continued from previous page)

```
23 * 67 * 419

sage: S = simplicial_complexes.SumComplex(13, [0, 1, 3]); S
Sum complex on vertices Z/13Z associated to {0, 1, 3}
sage: S.homology(1)                                                          #␣
↪needs sage.modules
C159
sage: factor(159)
3 * 53
sage: S = simplicial_complexes.SumComplex(13, [0, 1, 2, 5]); S
Sum complex on vertices Z/13Z associated to {0, 1, 2, 5}
sage: S.homology()                          # long time                      #␣
↪needs sage.modules
{0: 0, 1: 0, 2: C146989209, 3: 0}
sage: factor(1648910295)
3^2 * 5 * 53 * 521 * 1327
sage: S = simplicial_complexes.SumComplex(13, [0, 1, 2, 3, 5]); S
Sum complex on vertices Z/13Z associated to {0, 1, 2, 3, 5}
sage: S.homology()                          # long time                      #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: C3 x C237 x C706565607945, 4: 0}
sage: factor(706565607945)                                                   #␣
↪needs sage.libs.pari
3 * 5 * 53 * 79 * 131 * 157 * 547

sage: S = simplicial_complexes.SumComplex(17, [0, 1, 4]); S
Sum complex on vertices Z/17Z associated to {0, 1, 4}
sage: S.homology(1)                                                          #␣
↪needs sage.modules
C140183
sage: factor(140183)
103 * 1361
sage: S = simplicial_complexes.SumComplex(19, [0, 1, 4]); S
Sum complex on vertices Z/19Z associated to {0, 1, 4}
sage: S.homology(1)                                                          #␣
↪needs sage.modules
C5670599
sage: factor(5670599)
11 * 191 * 2699
sage: S = simplicial_complexes.SumComplex(31, [0, 1, 4]); S
Sum complex on vertices Z/31Z associated to {0, 1, 4}
sage: S.homology(1)                         # long time                      #␣
↪needs sage.modules
C5 x C5 x C5 x C5 x C26951480558170926865
sage: factor(26951480558170926865)                                           #␣
↪needs sage.libs.pari
5 * 311 * 683 * 1117 * 11657 * 1948909
```

sage.topology.simplicial_complex_examples.**SurfaceOfGenus**(*g*, *orientable=True*)

A surface of genus *g*.

INPUT:

- `g` – nonnegative integer; the desired genus
- `orientable` – boolean (default: `True`); if `True`, return an orientable surface, and if `False`, return a non-orientable surface.

In the orientable case, return a sphere if $g$ is zero, and otherwise return a $g$-fold connected sum of a torus with itself.

In the non-orientable case, raise an error if $g$ is zero. If $g$ is positive, return a $g$-fold connected sum of a real projective plane with itself.

EXAMPLES:

```
sage: simplicial_complexes.SurfaceOfGenus(2)
Triangulation of an orientable surface of genus 2
sage: simplicial_complexes.SurfaceOfGenus(1, orientable=False)
Triangulation of a non-orientable surface of genus 1
```

sage.topology.simplicial_complex_examples.**Torus**()

A minimal triangulation of the torus.

This is a simplicial complex with 7 vertices, 21 edges and 14 faces. It is the unique triangulation of the torus with 7 vertices, and has been found by Möbius in 1861.

This is also the combinatorial structure of the Császár polyhedron (see Wikipedia article Császár_polyhedron).

EXAMPLES:

```
sage: T = simplicial_complexes.Torus()
sage: T.homology(1)                                                         #␣
↪needs sage.modules
Z x Z
sage: T.f_vector()
[1, 7, 21, 14]
```

REFERENCES:

- [Lut2002]

**class** sage.topology.simplicial_complex_examples.**UniqueSimplicialComplex**(*maximal_faces=None*, *name=None*, ***kwds*)

Bases: *SimplicialComplex*, UniqueRepresentation

This combines *SimplicialComplex* and UniqueRepresentation. It is intended to be used to make standard examples of simplicial complexes unique. See Issue #13566.

INPUT:

- the inputs are the same as for a *SimplicialComplex*, with one addition and two exceptions. The exceptions are that `is_mutable` and `is_immutable` are ignored: all instances of this class are immutable. The addition:
- `name` – string (optional); the string representation for this complex

EXAMPLES:

```
sage: from sage.topology.simplicial_complex_examples import␣
↪UniqueSimplicialComplex
sage: SimplicialComplex([[0, 1]]) is SimplicialComplex([[0, 1]])
False
sage: UniqueSimplicialComplex([[0, 1]]) is UniqueSimplicialComplex([[0, 1]])
True
sage: UniqueSimplicialComplex([[0, 1]])
Simplicial complex with vertex set (0, 1) and facets {(0, 1)}
```

(continues on next page)

(continued from previous page)

```
sage: UniqueSimplicialComplex([[0, 1]], name='The 1-simplex')
The 1-simplex
```

sage.topology.simplicial_complex_examples.**ZieglerBall**()

Return the non-shellable ball constructed by Ziegler.

This complex is a non-shellable triangulation of the 3-ball with 10 vertices and 21 facets, constructed by Ziegler in [Zie1998] and the smallest such complex known.

EXAMPLES:

```
sage: Z = simplicial_complexes.ZieglerBall(); Z
Ziegler ball
sage: Z.f_vector()
[1, 10, 38, 50, 21]
sage: Z.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0}
sage: Z.is_cohen_macaulay()                                                 #␣
↪needs sage.modules
True
```

sage.topology.simplicial_complex_examples.**facets_for_K3**()

Return 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.topology.simplicial_complex_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.topology.simplicial_complex_examples.**facets_for_RP4**()

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

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

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

EXAMPLES:

```
sage: from sage.topology.simplicial_complex_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.topology.simplicial_complex_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.topology.simplicial_complex_examples import matching
sage: matching([1, 2], [3, 4])
[{(1, 3), (2, 4)}, {(1, 4), (2, 3)}]
sage: matching([0, 2], [0])
[{(0, 0)}, {(2, 0)}]
```

CHAPTER

FIVE

# FINITE DELTA-COMPLEXES

AUTHORS:

- John H. Palmieri (2009-08)

This module implements the basic structure of finite $\Delta$-complexes. For full mathematical details, see Hatcher [Hat2002], especially Section 2.1 and the Appendix on "Simplicial CW Structures". As Hatcher points out, $\Delta$-complexes were first introduced by Eilenberg and Zilber [EZ1950], although they called them "semi-simplicial complexes".

A $\Delta$-complex is a generalization of a `simplicial complex`; a $\Delta$-complex $X$ consists of sets $X_n$ for each nonnegative 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 $\Delta$-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.

$\Delta$-complexes are an alternative to simplicial complexes. Every simplicial complex is automatically a $\Delta$-complex; in the other direction, though, it seems in practice that one can often construct $\Delta$-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 $\Delta$-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.

**class** sage.topology.delta_complex.**DeltaComplex**(*data=None*, *check_validity=True*)

Bases: *GenericCellComplex*

Define a $\Delta$-complex.

INPUT:

- `data` – see below for a description of the options
- `check_validity` – boolean (default: `True`); if `True`, check that the simplicial identities hold

OUTPUT: a $\Delta$-complex

Use `data` to define a $\Delta$-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 i-th entry is the i-th face of S, given as a simplex,
  - another d-simplex $T$, in which case the i-th face of $S$ is declared to be the same as the i-th 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 $\Delta$-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:

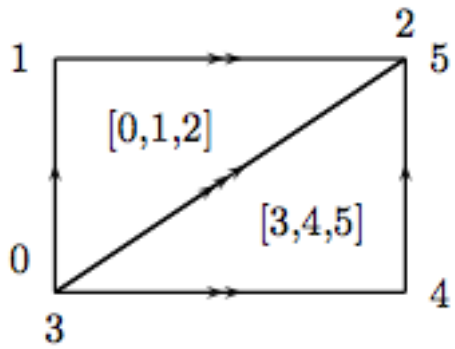
```
sage: S5.homology()
↪  # needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: Z}
sage: S5.f_vector()  # number of simplices in each dimension
[1, 6, 15, 20, 15, 6, 2]
sage: simplicial_complexes.Sphere(5).f_vector()
[1, 7, 21, 35, 35, 21, 7]
```

Both contain a single (-1)-simplex, the empty simplex; other than that, the $\Delta$-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)
↪  # needs sage.modules
{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 $\Delta$-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 n-th face is obtained by omitting the n-th 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 n-th entry in the list is a list of the n-simplices in the complex, and each n-simplex is encoded as a list, the i-th entry of which is its i-th face. Each face is represented by an integer, giving its index in the list of (n-1)-faces. For example, consider this:

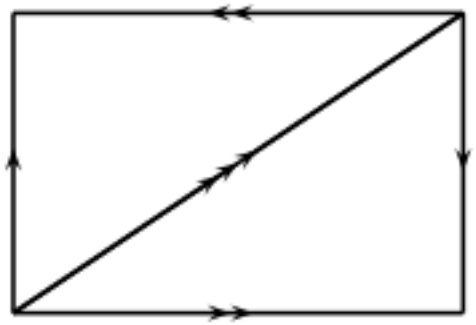
```
sage: P = DeltaComplex( [ [(), ()],  [(1,0), (1,0), (0,0)],
....:                     [(1,0,2), (0, 1, 2)] ])
```

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)
↪  # needs sage.modules
C2
sage: P.cohomology(2)
↪  # needs sage.modules
C2
```

Closely related to this form for `data` is `X.cells()` for a $\Delta$-complex `X`: this is a dictionary, indexed by dimension `d`, whose `d`-th entry is a list of the `d`-simplices, as a list:

```
sage: P.cells()
{-1: ((),),
 0: ((), ()),
```

(continues on next page)

(continued from previous page)

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

- `data` may be a dictionary indexed by integers. For each integer $n$, the entry with key $n$ is the list of $n$-simplices: this is the same format as is output by the *`cells()`* method.

```
sage: P = DeltaComplex( [ [(), ()],  [(1,0), (1,0), (0,0)],
....:                     [(1,0,2), (0, 1, 2)] ])
sage: cells_dict = P.cells()
sage: cells_dict
{-1: ((),),
 0: ((), ()),
 1: ((1, 0), (1, 0), (0, 0)),
 2: ((1, 0, 2), (0, 1, 2))}
sage: DeltaComplex(cells_dict)
Delta complex with 2 vertices and 8 simplices
sage: P == DeltaComplex(cells_dict)
True
```

Since $\Delta$-complexes are generalizations of simplicial complexes, any simplicial complex may be viewed as a $\Delta$-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, $\Delta$-complex constructions for several familiar spaces are available as follows:

```
sage: delta_complexes.Sphere(4)  # the 4-sphere
Delta complex with 5 vertices and 33 simplices
sage: delta_complexes.KleinBottle()
Delta complex with 1 vertex and 7 simplices
sage: delta_complexes.RealProjectivePlane()
Delta complex with 2 vertices and 8 simplices
```

Type `delta_complexes.` and then hit the `Tab` key to get the full list.

**alexander_whitney**(*cell*, *dim_left*)

Subdivide `cell` in this $\Delta$-complex into a pair of simplices.

For an abstract simplex with vertices $v_0$, $v_1$, ..., $v_n$, then subdivide it into simplices $(v_0, v_1, ..., v_{dim_left})$ and $(v_{dim_left}, v_{dim_left+1}, ..., v_n)$. In a $\Delta$-complex, instead take iterated faces: take top faces to get the left factor, take bottom faces to get the right factor.

INPUT:

- `cell` – a simplex in this complex, given as a pair `(idx, tuple)`, where `idx` is its index in the list of cells in the given dimension, and `tuple` is the tuple of its faces
- `dim_left` – integer between 0 and one more than the dimension of this simplex

OUTPUT: list containing just the triple `(1, left, right)`, where `left` and `right` are the two cells described above, each given as pairs `(idx, tuple)`.

EXAMPLES:

```
sage: X = delta_complexes.Torus()
sage: X.n_cells(2)
[(1, 2, 0), (0, 2, 1)]
sage: X.alexander_whitney((0, (1, 2, 0)), 1)
[(1, (0, (0, 0)), (1, (0, 0)))]
sage: X.alexander_whitney((0, (1, 2, 0)), 0)
[(1, (0, ()), (0, (1, 2, 0)))]
sage: X.alexander_whitney((1, (0, 2, 1)), 2)
[(1, (1, (0, 2, 1)), (0, ()))]
```

**algebraic_topological_model**(*base_ring=None*)

Algebraic topological model for this $\Delta$-complex with coefficients in `base_ring`.

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

INPUT:

- `base_ring` – coefficient ring (default: `QQ`); must be a field

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

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

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

OUTPUT: a pair consisting of

- chain contraction `phi` associated to $C$, $M$, $\pi$, and $\iota$
- the chain complex $M$

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

EXAMPLES:

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

**barycentric_subdivision**()

Not implemented.

EXAMPLES:

```
sage: K = delta_complexes.KleinBottle()
sage: K.barycentric_subdivision()
Traceback (most recent call last):
...
NotImplementedError: barycentric subdivisions are not implemented for Delta␣
↪complexes
```

**cells** (*subcomplex=None*)

The cells of this $\Delta$-complex.

INPUT:

- `subcomplex` – a subcomplex of this complex (default: `None`)

The cells of this $\Delta$-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 i-th entry of which gives the index of its i-th face in the list of (d-1)-cells.

If the optional argument `subcomplex` is present, then "return only the faces which are *not* in the subcomplex". To preserve the indexing, which is necessary to compute the relative chain complex, this actually replaces the faces in `subcomplex` with `None`.

EXAMPLES:

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

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

The chain complex associated to this $\Delta$-complex.

INPUT:

- `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` – commutative ring (default: `ZZ`)
- `subcomplex` – a subcomplex of this simplicial complex (default: empty). Compute the chain complex relative to this subcomplex.
- `augmented` – boolean (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 (default: `False`); if `True`, return the cochain complex (that is, the dual of the chain complex)
- `verbose` – boolean (default: `False`); if `True`, print some messages as the chain complex is computed

- `check` – boolean (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: # needs sage.modules
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 $\Delta$-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()                                                   #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0}
```

**connected_sum**(*other*)

Return the connected sum of `self` with `other`.

INPUT:

- `other` – another $\Delta$-complex

OUTPUT: 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: # needs sage.modules
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 $\Delta$-complex with another one.

INPUT:

- `right` – the other $\Delta$-complex (the right-hand factor)

EXAMPLES:

```
sage: S1 = delta_complexes.Sphere(1)
sage: S2 = delta_complexes.Sphere(2)
sage: S1.disjoint_union(S2).homology()                                          #␣
↪needs sage.modules
{0: Z, 1: Z, 2: Z}
```

**elementary_subdivision**(*idx=-1*)

Perform an "elementary subdivision" on a top-dimensional simplex in this $\Delta$-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.

INPUT:

- `idx` – integer (default: -1); index specifying which simplex to subdivide

OUTPUT: $\Delta$-complex with one simplex subdivided

*Elementary subdivision* of a simplex means replacing that simplex with the cone on its boundary. That is, given a $\Delta$-complex containing a $d$-simplex $S$ with vertices $v_0$, …, $v_d$, form a new $\Delta$-complex by

- removing $S$
- adding a vertex $w$ (thought of as being in the interior of $S$)
- adding all simplices with vertices $v_{i_0}, \ldots, v_{i_k}, w$, 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 $\pi$ 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 subdivision" is taken from p. 112 in John M. Lee's book [Lee2011].

EXAMPLES:

```
sage: T = delta_complexes.Torus()
sage: T.n_cells(2)
[(1, 2, 0), (0, 2, 1)]
sage: T.elementary_subdivision(0)  # subdivide first triangle
Delta complex with 2 vertices and 13 simplices
sage: X = T.elementary_subdivision(); X  # subdivide last triangle
Delta complex with 2 vertices and 13 simplices
sage: X.elementary_subdivision()
Delta complex with 3 vertices and 19 simplices
sage: X.homology() == T.homology()                                              #␣
↪needs sage.modules
True
```

**face_poset**()

The face poset of this $\Delta$-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 $\Delta$-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 $\Delta$-complex with another one.

INPUT:

- `other` – another $\Delta$-complex (the right-hand factor)

OUTPUT: the join `self * other`

The join of two $\Delta$-complexes $S$ and $T$ is the $\Delta$-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 i-th 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), needs sage.modules
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()                                                    #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: Z}
```

**n_chains**(*n*, *base_ring=None*, *cochains=False*)

Return the free module of chains in degree `n` over `base_ring`.

INPUT:

- `n` – integer
- `base_ring` – ring (default: **Z**)
- `cochains` – boolean (default: `False`); if `True`, return cochains instead

Since the list of $n$-cells for a $\Delta$-complex may have some ambiguity – for example, the list of edges may look like `[(0, 0), (0, 0), (0, 0)]` if each edge starts and ends at vertex 0 – we record the indices of the cells along with their tuples. So the basis of chains in such a case would look like `[(0, (0, 0)), (1, (0, 0)), (2, (0, 0))]`.

The only difference between chains and cochains is notation: the dual cochain to the chain basis element `b` is written as `\chi_b`.

EXAMPLES:

```
sage: T = delta_complexes.Torus()
sage: T.n_chains(1, QQ)                                               #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
Free module generated by {(0, (0, 0)), (1, (0, 0)), (2, (0, 0))}
 over Rational Field
sage: list(T.n_chains(1, QQ, cochains=False).basis())                          #␣
↪needs sage.modules
[(0, (0, 0)), (1, (0, 0)), (2, (0, 0))]
sage: list(T.n_chains(1, QQ, cochains=True).basis())                           #␣
↪needs sage.modules
[\chi_(0, (0, 0)), \chi_(1, (0, 0)), \chi_(2, (0, 0))]
```

**n_skeleton**(*n*)

The n-skeleton of this $\Delta$-complex.

- `n` – nonnegative 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()                                              #␣
↪needs sage.modules
{0: 0, 1: Z x Z x Z}
```

**product**(*other*)

The product of this $\Delta$-complex with another one.

INPUT:

- `other` – another $\Delta$-complex (the right-hand factor)

OUTPUT: the product `self x other`

**Warning**

If `X` and `Y` are $\Delta$-complexes, then `X*Y` returns their join, not their product.

EXAMPLES:

```
sage: K = delta_complexes.KleinBottle()
sage: X = K.product(K)

sage: # needs sage.modules
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,
```

(continues on next page)

(continued from previous page)

```
 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()                    #␣
↪needs sage.modules
True
sage: S1.product(S1) == delta_complexes.Torus()
True
```

**subcomplex**(*data*)

Create a subcomplex.

INPUT:

- `data` – dictionary indexed by dimension or a list (or tuple); in either case, data[n] should be the list (or tuple or set) of the indices of the simplices to be included in the subcomplex

This automatically includes all faces of the simplices in `data`, so you only have to specify the simplices which are maximal with respect to inclusion.

EXAMPLES:

```
sage: X = delta_complexes.Torus()
sage: A = X.subcomplex({2: [0]})  # one of the triangles of X
sage: X.homology(subcomplex=A)                                                #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z}
```

In the following, `line` is a line segment and `ends` is the complex consisting of its two endpoints, so the relative homology of the two is isomorphic to the homology of a circle:

```
sage: line = delta_complexes.Simplex(1) # an edge
sage: line.cells()
{-1: ((),), 0: ((), ()), 1: ((0, 1),)}
sage: ends = line.subcomplex({0: (0, 1)})
sage: ends.cells()
{-1: ((),), 0: ((), ())}
sage: line.homology(subcomplex=ends)                                          #␣
↪needs sage.modules
{0: 0, 1: Z}
```

**suspension**(*n=1*)

The suspension of this $\Delta$-complex.

- `n` – positive integer (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()                                                           #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: Z}
```

**wedge**(*right*)

The wedge (one-point union) of this $\Delta$-complex with another one.

- `right` – the other $\Delta$-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()                                                  #␣
↪needs sage.modules
{0: 0, 1: Z, 2: Z}
```

**class** sage.topology.delta_complex.**DeltaComplexExamples**

Bases: `object`

Some examples of $\Delta$-complexes.

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

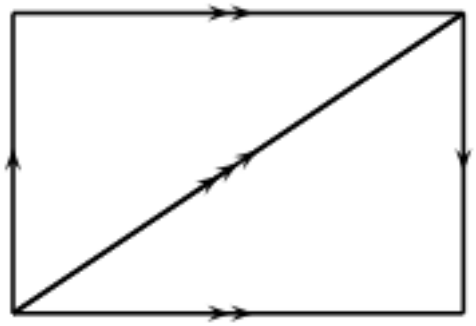
```
Sphere
Torus
RealProjectivePlane
KleinBottle
Simplex
SurfaceOfGenus
```

EXAMPLES:

```
sage: S = delta_complexes.Sphere(6) # the 6-sphere
sage: S.dimension()
6
sage: S.cohomology(6)                                                          #␣
↪needs sage.modules
Z
sage: delta_complexes.Torus() == delta_complexes.Sphere(3)
False
```

**KleinBottle**()

A $\Delta$-complex representation of the Klein bottle, consisting of one vertex, three edges, and two triangles.

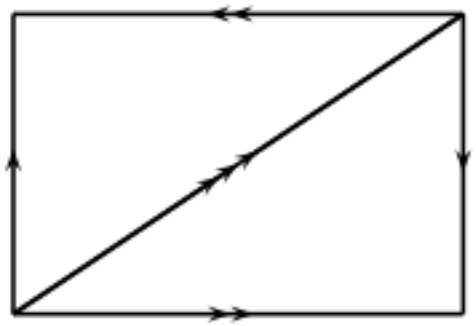


EXAMPLES:

```
sage: delta_complexes.KleinBottle()
Delta complex with 1 vertex and 7 simplices
```

**RealProjectivePlane**()

A $\Delta$-complex representation of the real projective plane, consisting of two vertices, three edges, and two triangles.



EXAMPLES:

```
sage: # needs sage.modules
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 $\Delta$-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 $\Delta$-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.

INPUT:

- `n` – dimension of the sphere

EXAMPLES:

```
sage: delta_complexes.Sphere(4).cohomology(4, base_ring=GF(3))                #␣
↪needs sage.modules
Vector space of dimension 1 over Finite Field of size 3
```

**SurfaceOfGenus**($g$, *orientable=True*)

A surface of genus g as a $\Delta$-complex.

INPUT:

- `g` – nonnegative integer; the genus
- `orientable` – boolean (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:

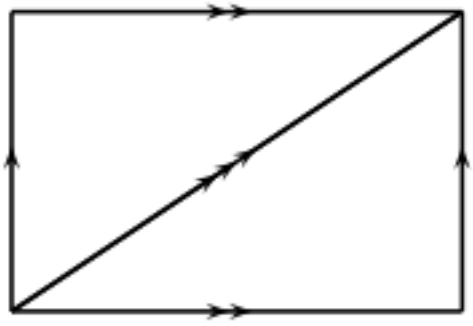
```
sage: delta_complexes.SurfaceOfGenus(1, orientable=False)
Delta complex with 2 vertices and 8 simplices
sage: delta_complexes.SurfaceOfGenus(3, orientable=False).homology(1)       #␣
↪needs sage.modules
Z x Z x C2
sage: delta_complexes.SurfaceOfGenus(3, orientable=False).homology(2)       #␣
↪needs sage.modules
0
```

Compare to simplicial complexes:

```
sage: delta_g4 = delta_complexes.SurfaceOfGenus(4)
sage: delta_g4.f_vector()
[1, 3, 27, 18]
sage: simpl_g4 = simplicial_complexes.SurfaceOfGenus(4)
sage: simpl_g4.f_vector()
[1, 19, 75, 50]
sage: delta_g4.homology() == simpl_g4.homology()                            #␣
↪needs sage.modules
True
```

**Torus**()

A $\Delta$-complex representation of the torus, consisting of one vertex, three edges, and two triangles.



EXAMPLES:

```
sage: delta_complexes.Torus().homology(1)                                  #␣
↪needs sage.modules sage.rings.finite_rings
Z x Z
```

CHAPTER

SIX

# FINITE CUBICAL COMPLEXES

AUTHORS:

- John H. Palmieri (2009-08)

This module implements the basic structure of finite cubical complexes. For full mathematical details, see Kaczynski, Mischaikow, and Mrozek [KMM2004], for example.

Cubical complexes are topological spaces built from gluing together cubes of various dimensions; the collection of cubes must be closed under taking faces, just as with a simplicial complex. In this context, a "cube" means a product of intervals of length 1 or length 0 (degenerate intervals), with integer endpoints, and its faces are obtained by using the nondegenerate intervals: if $C$ is a cube – a product of degenerate and nondegenerate intervals – and if $[i, i+1]$ is the $k$-th nondegenerate factor, then $C$ has two faces indexed by $k$: the cubes obtained by replacing $[i, i+1]$ with $[i, i]$ or $[i+1, i+1]$.

So to construct a space homeomorphic to a circle as a cubical complex, we could take for example the four line segments in the plane from $(0, 2)$ to $(0, 3)$ to $(1, 3)$ to $(1, 2)$ to $(0, 2)$. In Sage, this is done with the following command:

```
sage: S1 = CubicalComplex([([0,0], [2,3]), ([0,1], [3,3]),
....:                      ([0,1], [2,2]), ([1,1], [2,3])]); S1
Cubical complex with 4 vertices and 8 cubes
```

The argument to `CubicalComplex` is a list of the maximal "cubes" in the complex. Each "cube" can be an instance of the class `Cube` or a list (or tuple) of "intervals", and an "interval" is a pair of integers, of one of the two forms $[i, i]$ or $[i, i+1]$. So the cubical complex `S1` above has four maximal cubes:

```
sage: len(S1.maximal_cells())
4
sage: sorted(S1.maximal_cells())
[[0,0] x [2,3], [0,1] x [2,2], [0,1] x [3,3], [1,1] x [2,3]]
```

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()                                                             #␣
↪needs sage.modules
{0: 0, 1: Z}
sage: X = CubicalComplex([([0,0], [2,3], [2]), ([0,1], [3,3], [2]),
....:                     ([0,1], [2,2], [2]), ([1,1], [2,3], [2])])
sage: X.homology()                                                              #␣
↪needs sage.modules
{0: 0, 1: Z}
```

Topologically, the cubical complex `X` consists of four edges of a square in $\mathbf{R}^3$: the same unit square as `S1`, but embedded in $\mathbf{R}^3$ with $z$-coordinate equal to 2. Thus `X` is homeomorphic to `S1` (in fact, they're "cubically equivalent"), and this is reflected in the fact that they have isomorphic homology groups.

**Note**

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

**class** sage.topology.cubical_complex.**Cube**(*data*)

Bases: `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 $\mathbf{R}^4$.

INPUT:

- `data` – list or tuple of terms of the form `(i,i+1)` or `(i,i)` or `(i,)`; the last two are degenerate intervals

OUTPUT: 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.topology.cubical_complex import Cube
sage: C = Cube([[1,2], [5,], [6,7], [-1, 0]]); C
[1,2] x [5,5] x [6,7] x [-1,0]
sage: C.dimension() # number of nondegenerate intervals
3
sage: C.nondegenerate_intervals()  # indices of these intervals
[0, 2, 3]
sage: C.face(1, upper=False)
[1,2] x [5,5] x [6,6] x [-1,0]
sage: C.face(1, upper=True)
[1,2] x [5,5] x [7,7] x [-1,0]
sage: Cube(()).dimension()  # empty cube has dimension -1
-1
```

**alexander_whitney**(*dim*)

Subdivide this cube into pairs of cubes.

This provides a cubical approximation for the diagonal map $K \to K \times K$.

INPUT:

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

OUTPUT:

- a list containing triples `(coeff, left, right)`

This uses the algorithm described by Pilarczyk and Réal [PR2015] on p. 267; the formula is originally due to Serre. Calling this method `alexander_whitney` is an abuse of notation, since the actual Alexander-Whitney map goes from $C(K \times L) \to C(K) \otimes C(L)$, where $C(-)$ denotes the associated chain complex, but this subdivision of cubes is at the heart of it.

EXAMPLES:

```
sage: from sage.topology.cubical_complex import Cube
sage: C1 = Cube([[0,1], [3,4]])
sage: C1.alexander_whitney(0)
[(1, [0,0] x [3,3], [0,1] x [3,4])]
sage: C1.alexander_whitney(1)
[(1, [0,1] x [3,3], [1,1] x [3,4]), (-1, [0,0] x [3,4], [0,1] x [4,4])]
sage: C1.alexander_whitney(2)
[(1, [0,1] x [3,4], [1,1] x [4,4])]
```

**dimension**()

The dimension of this cube: the number of its nondegenerate intervals.

EXAMPLES:

```
sage: from sage.topology.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 $n$-th primary face of this cube.

INPUT:

- `n` – integer between 0 and one less than the dimension of this cube
- `upper` – boolean (default=True);if `True`, return the "upper" $n$-th primary face; otherwise, return the "lower" $n$-th primary face

OUTPUT: the cube obtained by replacing the $n$-th non-degenerate interval with either its upper or lower endpoint.

EXAMPLES:

```
sage: from sage.topology.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 n-th face if 0 <= n < dim
```

**faces**()

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

EXAMPLES:

```
sage: from sage.topology.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.topology.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.topology.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.topology.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.

INPUT:

- `other` – another cube

OUTPUT: the product of `self` and `other`, as a Cube

EXAMPLES:

```
sage: from sage.topology.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.topology.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.topology.cubical_complex.**CubicalComplex**(*maximal_faces=None*, *maximality_check=True*)

Bases: *GenericCellComplex*

Define a cubical complex.

INPUT:

- maximal_faces – set of maximal faces
- maximality_check – boolean (default: True); see below

OUTPUT: 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])]); S1
Cubical complex with 4 vertices and 8 cubes
sage: S1.homology()                                                    #␣
↪needs sage.modules
{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()                                                    #␣
↪needs sage.modules
{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()                                                      #␣
↪needs sage.modules
{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))           #␣
↪needs sage.modules
True
```

You can get the set of maximal cells or a dictionary of all cells:

```
sage: X.maximal_cells() # random: order may depend on the version of Python
{[0,0] x [2,3] x [-12,-12], [0,1] x [3,3] x [5,5], [0,1] x [2,2] x [3,3], [0,1] x␣
↪[2,2] x [0,0], [0,1] x [3,3] x [6,6], [1,1] x [2,3] x [0,0], [0,1] x [2,2] x [-
↪12,-12], [0,0] x [2,3] x [6,6], [1,1] x [2,3] x [-12,-12], [1,1] x [2,3] x [5,
↪5], [0,1] x [2,2] x [5,5], [0,1] x [3,3] x [3,3], [1,1] x [2,3] x [3,3], [0,0]␣
↪x [2,3] x [5,5], [0,1] x [3,3] x [0,0], [1,1] x [2,3] x [6,6], [0,1] x [2,2] x␣
↪[6,6], [0,0] x [2,3] x [0,0], [0,0] x [2,3] x [3,3], [0,1] x [3,3] x [-12,-12]}
sage: sorted(X.maximal_cells())
[[0,0] x [2,3] x [-12,-12],
 [0,0] x [2,3] x [0,0],
 [0,0] x [2,3] x [3,3],
 [0,0] x [2,3] x [5,5],
 [0,0] x [2,3] x [6,6],
 [0,1] x [2,2] x [-12,-12],
 [0,1] x [2,2] x [0,0],
 [0,1] x [2,2] x [3,3],
 [0,1] x [2,2] x [5,5],
 [0,1] x [2,2] x [6,6],
 [0,1] x [3,3] x [-12,-12],
 [0,1] x [3,3] x [0,0],
 [0,1] x [3,3] x [3,3],
 [0,1] x [3,3] x [5,5],
 [0,1] x [3,3] x [6,6],
 [1,1] x [2,3] x [-12,-12],
 [1,1] x [2,3] x [0,0],
 [1,1] x [2,3] x [3,3],
 [1,1] x [2,3] x [5,5],
 [1,1] x [2,3] x [6,6]]
sage: S1.cells()
{-1: set(),
 0: {[0,0] x [2,2], [0,0] x [3,3], [1,1] x [2,2], [1,1] x [3,3]},
 1: {[0,0] x [2,3], [0,1] x [2,2], [0,1] x [3,3], [1,1] x [2,3]}}
```

Chain complexes, homology, and cohomology:

```
sage: T = S1.product(S1); T
Cubical complex with 16 vertices and 64 cubes
sage: T.chain_complex()                                                    #␣
↪needs sage.modules
Chain complex with at most 3 nonzero terms over Integer Ring
sage: T.homology(base_ring=QQ)                                             #␣
↪needs sage.modules
{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))                          #␣
↪needs sage.modules
{1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
```

Joins are not implemented:

```
sage: S1.join(S1)
Traceback (most recent call last):
...
NotImplementedError: joins are not implemented for cubical complexes
```

Therefore, neither are cones or suspensions.

**alexander_whitney**(*cube*, *dim_left*)

Subdivide `cube` in this cubical complex into pairs of cubes.

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

INPUT:

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

OUTPUT: list containing triples `(coeff, left, right)`

EXAMPLES:

```
sage: C = cubical_complexes.Cube(3)
sage: c = list(C.n_cubes(3))[0]; c
[0,1] x [0,1] x [0,1]
sage: C.alexander_whitney(c, 1)
[(1, [0,1] x [0,0] x [0,0], [1,1] x [0,1] x [0,1]),
 (-1, [0,0] x [0,1] x [0,0], [0,1] x [1,1] x [0,1]),
 (1, [0,0] x [0,0] x [0,1], [0,1] x [0,1] x [1,1])]
```

**algebraic_topological_model**(*base_ring=None*)

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

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

INPUT:

- `base_ring` – coefficient ring (default: `QQ`); must be a field

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

- $\phi\phi = 0$,

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

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

OUTPUT: a pair consisting of

- chain contraction `phi` associated to $C$, $M$, $\pi$, and $\iota$
- the chain complex $M$

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

EXAMPLES:

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

**cells** (*subcomplex=None*)

The cells of this cubical complex, in the form of a dictionary: the keys are integers, representing dimension, and the value associated to an integer d is the list of d-cells.

If the optional argument `subcomplex` is present, then return only the faces which are *not* in the subcomplex.

INPUT:

- `subcomplex` – a subcomplex of this cubical complex (default: `None`)

OUTPUT: dictionary; the cells of this complex not contained in `subcomplex`

EXAMPLES:

```
sage: S2 = cubical_complexes.Sphere(2)
sage: sorted(S2.cells()[2])
[[0,0] x [0,1] x [0,1],
 [0,1] x [0,0] x [0,1],
 [0,1] x [0,1] x [0,0],
 [0,1] x [0,1] x [1,1],
 [0,1] x [1,1] x [0,1],
 [1,1] x [0,1] x [0,1]]
```

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

The chain complex associated to this cubical complex.

INPUT:

- `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` – commutative ring (default: ZZ)
- `subcomplex` – a subcomplex of this cubical complex (default: empty). Compute the chain complex relative to this subcomplex.
- `augmented` – boolean (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 (default: `False`); if `True`, return the cochain complex (that is, the dual of the chain complex).
- `verbose` – boolean (default: `False`); if `True`, print some messages as the chain complex is computed.
- `check` – boolean (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: # needs sage.modules
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}
```

Check that Issue #32203 has been fixed:

```
sage: # needs sage.modules
sage: Square = CubicalComplex([([0,1],[0,1])])
sage: EdgesLTR = CubicalComplex([([0,0],[0,1]),([0,1],[1,1]),([1,1],[0,1])])
sage: EdgesLBR = CubicalComplex([([0,0],[0,1]),([0,1],[0,0]),([1,1],[0,1])])
sage: Square.homology(subcomplex=EdgesLTR)[2] == Square.
↪homology(subcomplex=EdgesLBR)[2]
True
```

**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`.

INPUT:

- `other` – another cubical complex

OUTPUT: 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()                    #␣
↪needs sage.modules
True
sage: RP2 = cubical_complexes.RealProjectivePlane()
sage: T.connected_sum(RP2).homology(1)                                      #␣
↪needs sage.modules
Z x Z x C2
sage: RP2.connected_sum(RP2).connected_sum(RP2).homology(1)                 #␣
↪needs sage.modules
Z x Z x C2
```

**disjoint_union**(*other*)

The disjoint union of this cubical complex with another one.

INPUT:

- `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()                                      #␣
↪needs sage.modules
{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**()

Return `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`.

INPUT:

- `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.

INPUT:

- `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.

OUTPUT: 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.

INPUT:

- `n` – integer; dimension
- `subcomplex` – a subcomplex of this cubical complex (default: `None`)

OUTPUT: set; cells in dimension `n`

EXAMPLES:

```
sage: C = cubical_complexes.Cube(3)
sage: C.n_cubes(3)
{[0,1] x [0,1] x [0,1]}
sage: sorted(C.n_cubes(2))
```

(continues on next page)

(continued from previous page)

```
[[0,0] x [0,1] x [0,1],
 [0,1] x [0,0] x [0,1],
 [0,1] x [0,1] x [0,0],
 [0,1] x [0,1] x [1,1],
 [0,1] x [1,1] x [0,1],
 [1,1] x [0,1] x [0,1]]
```

**n_skeleton**(*n*)

The n-skeleton of this cubical complex.

INPUT:

- `n` – nonnegative integer; dimension

OUTPUT: cubical complex

EXAMPLES:

```
sage: S2 = cubical_complexes.Sphere(2)
sage: C3 = cubical_complexes.Cube(3)
sage: S2 == C3.n_skeleton(2)
True
```

**product**(*other*)

Return the product of this cubical complex with another one.

INPUT:

- `other` – another cubical complex

EXAMPLES:

```
sage: RP2 = cubical_complexes.RealProjectivePlane()
sage: S1 = cubical_complexes.Sphere(1)
sage: RP2.product(S1).homology()[1] # long time: 5 seconds
Z x C2
```

**suspension**(*n=1*)

The suspension of this cubical complex.

NOT IMPLEMENTED

INPUT:

- `n` – positive integer (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.

INPUT:

- `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()                                                  #␣
↪needs sage.modules
{0: 0, 1: Z, 2: Z}
```

**class** sage.topology.cubical_complex.**CubicalComplexExamples**

Bases: `object`

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.

INPUT:

- `n` – nonnegative 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.

INPUT:

- `n` – nonnegative 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.

INPUT:

- `g` – nonnegative integer; the genus
- `orientable` – boolean (default: `True`); whether the surface should be orientable

In the orientable case, return a sphere if $g$ is zero, and otherwise return a $g$-fold connected sum of a torus with itself.

In the non-orientable case, raise an error if $g$ is zero. If $g$ is positive, return a $g$-fold connected sum of a real projective plane with itself.

EXAMPLES:

```
sage: cubical_complexes.SurfaceOfGenus(2)
Cubical complex with 32 vertices and 134 cubes
sage: cubical_complexes.SurfaceOfGenus(1, orientable=False)
Cubical complex with 21 vertices and 81 cubes
```

**Torus**()

A cubical complex representation of the torus, obtained by taking the product of the circle with itself.

EXAMPLES:

```
sage: cubical_complexes.Torus()
Cubical complex with 16 vertices and 64 cubes
```

CHAPTER

SEVEN

# SIMPLICIAL SETS

AUTHORS:

- John H. Palmieri (2016-07)

This module implements simplicial sets.

A *simplicial set* $X$ is a collection of sets $X_n$ indexed by the nonnegative integers; the set $X_n$ is called the set of $n$-simplices. These sets are connected by maps

$$d_i : X_n \to X_{n-1}, \ \ 0 \leq i \leq n \ \text{ (face maps)}$$
$$s_j : X_n \to X_{n+1}, \ \ 0 \leq j \leq n \ \text{ (degeneracy maps)}$$

satisfying the *simplicial identities*:

$$d_i d_j = d_{j-1} d_i \ \text{ if } i < j$$
$$d_i s_j = s_{j-1} d_i \ \text{ if } i < j$$
$$d_j s_j = 1 = d_{j+1} s_j$$
$$d_i s_j = s_j d_{i-1} \ \text{ if } i > j+1$$
$$s_i s_j = s_{j+1} s_i \ \text{ if } i < j+1$$

See Wikipedia article Simplicial_set, Peter May's seminal book [May1967], or Greg Friedman's "Illustrated introduction" arXiv 0809.4221 for more information.

Several simplicial sets are predefined, and users can construct others either by hand (using *SimplicialSet_finite*) or from existing ones using pushouts, pullbacks, etc.

EXAMPLES:

Some of the predefined simplicial sets:

```
sage: simplicial_sets.Torus()
Torus
sage: simplicial_sets.RealProjectiveSpace(7)                                  #␣
↪needs sage.groups
RP^7
sage: S5 = simplicial_sets.Sphere(5); S5
S^5
sage: S5.nondegenerate_simplices()
[v_0, sigma_5]
```

One class of infinite simplicial sets is available: classifying spaces of groups, or more generally, nerves of finite monoids:

```
sage: Sigma4 = groups.permutation.Symmetric(4)                                #␣
↪needs sage.groups
```

(continues on next page)

(continued from previous page)

```
sage: Sigma4.nerve()                                                    #␣
↪needs sage.groups
Nerve of Symmetric group of order 4! as a permutation group
```

The same simplicial set (albeit with a different name) can also be constructed as

```
sage: simplicial_sets.ClassifyingSpace(Sigma4)                          #␣
↪needs sage.groups
Classifying space of Symmetric group of order 4! as a permutation group
```

Type `simplicial_sets.` and hit the `Tab` key to get a full list of the predefined simplicial sets.

You can construct new simplicial sets from old by taking quotients, subsimplicial sets, disjoint unions, wedges (if they are pointed), smash products (if they are pointed and finite), products, pushouts, pullbacks, cones, and suspensions, most of which also have maps associated with them. Wedges, for example:

```
sage: T = simplicial_sets.Torus()
sage: S3 = simplicial_sets.Sphere(3)
sage: T.is_pointed() and S3.is_pointed()
True
sage: T.wedge(S3)
Wedge: (Torus v S^3)
sage: T.disjoint_union(S3) == T.coproduct(S3)
False

sage: W = T.wedge(S3)
sage: W.inclusion_map(0).domain()
Torus
sage: W.projection_map(1).codomain()
Quotient: (Wedge: (Torus v S^3)/Simplicial set with 6 non-degenerate simplices)
```

If the 1-sphere were not already available via `simplicial_sets.Sphere(1)`, you could construct it as follows:

```
sage: pt = simplicial_sets.Simplex(0)
sage: edge = pt.cone()
sage: S1 = edge.quotient(edge.n_skeleton(0))
sage: S1
Quotient: (Cone of 0-simplex/Simplicial set with 2 non-degenerate simplices)
```

At this point, `S1` is pointed: every quotient is automatically given a base point, namely the image of the subcomplex. So its suspension is the reduced suspension, and therefore is small:

```
sage: S5 = S1.suspension(4)
sage: S5
Sigma^4(Quotient: (Cone of 0-simplex/Simplicial set with 2 non-degenerate simplices))
sage: S5.f_vector()
[1, 0, 0, 0, 0, 1]
```

If we forget about the base point in `S1`, we would get the unreduced suspension instead:

```
sage: Z1 = S1.unset_base_point()
sage: Z1.suspension(4).f_vector()
[2, 2, 2, 2, 1, 1]
```

The cone on a pointed simplicial set is the reduced cone. The $n$-simplex in Sage is not pointed, but the simplicial set `Point` is.

```
sage: simplicial_sets.Simplex(0).cone().f_vector()
[2, 1]
sage: simplicial_sets.Point().cone().f_vector()
[1]
```

For most simplicial sets (the `Point` is the main exception), each time it is constructed, it gives a distinct copy, and two distinct simplicial sets are never equal:

```
sage: T = simplicial_sets.Torus()
sage: T == simplicial_sets.Torus()
False
sage: T == T
True
sage: simplicial_sets.Torus() == simplicial_sets.Torus()
False
sage: simplicial_sets.Point() == simplicial_sets.Point()
True
```

You can construct subsimplicial sets by specifying a list of simplices, and then you can define the quotient simplicial set:

```
sage: X = simplicial_sets.Simplex(2)
sage: e,f,g = X.n_cells(1)
sage: Y = X.subsimplicial_set([e,f,g])
sage: Z = X.quotient(Y)
```

Or equivalently:

```
sage: Y = X.n_skeleton(1)
sage: Z = X.quotient(Y)
sage: Z
Quotient: (2-simplex/Simplicial set with 6 non-degenerate simplices)
```

Note that subsimplicial sets and quotients come equipped with inclusion and quotient morphisms:

```
sage: inc = Y.inclusion_map()
sage: inc.domain() == Y and inc.codomain() == X
True
sage: quo = Z.quotient_map()
sage: quo.domain()
2-simplex
sage: quo.codomain() == Z
True
```

You can compute homology groups and the fundamental group of any simplicial set:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: eight = S1.wedge(S1)
sage: eight.fundamental_group()                                                 #␣
↪needs sage.groups
Finitely presented group < e0, e1 | >

sage: # needs sage.groups
sage: Sigma3 = groups.permutation.Symmetric(3)
sage: BSigma3 = Sigma3.nerve()
sage: pi = BSigma3.fundamental_group(); pi
Finitely presented group < e1, e2 | e2^2, e1^3, (e2*e1)^2 >
sage: pi.order()
```

(continues on next page)

(continued from previous page)

```
6
sage: pi.is_abelian()
False

sage: RP6 = simplicial_sets.RealProjectiveSpace(6)                              #␣
↪needs sage.groups
sage: RP6.homology(reduced=False, base_ring=GF(2))                              #␣
↪needs sage.groups sage.modules
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2,
 3: Vector space of dimension 1 over Finite Field of size 2,
 4: Vector space of dimension 1 over Finite Field of size 2,
 5: Vector space of dimension 1 over Finite Field of size 2,
 6: Vector space of dimension 1 over Finite Field of size 2}
sage: RP6.homology(reduced=False, base_ring=QQ)                                 #␣
↪needs sage.groups sage.modules
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 0 over Rational Field,
 4: Vector space of dimension 0 over Rational Field,
 5: Vector space of dimension 0 over Rational Field,
 6: Vector space of dimension 0 over Rational Field}
```

When infinite simplicial sets are involved, most computations are done by taking an $n$-skeleton for an appropriate $n$, either implicitly or explicitly:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([3])
sage: B3 = simplicial_sets.ClassifyingSpace(G)
sage: B3.disjoint_union(B3).n_skeleton(3)
Disjoint union: (Simplicial set with 15 non-degenerate simplices
                 u Simplicial set with 15 non-degenerate simplices)
sage: S1 = simplicial_sets.Sphere(1)
sage: B3.product(S1).homology(range(4))                                         #␣
↪needs sage.modules
{0: 0, 1: Z x C3, 2: C3, 3: C3}
```

Without the `range` argument, this would raise an error, since `B3` is infinite:

```
sage: B3.product(S1).homology()                                                 #␣
↪needs sage.groups sage.modules
Traceback (most recent call last):
...
NotImplementedError: this simplicial set may be infinite,
so specify dimensions when computing homology
```

It should be easy to construct many simplicial sets from the predefined ones using pushouts, pullbacks, etc., but they can also be constructed "by hand": first define some simplices, then define a simplicial set by specifying their faces:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: f = AbstractSimplex(1, name='f')
sage: X = SimplicialSet({e: (v,w), f: (w,w)})
```

Now $e$ is an edge from $v$ to $w$ and $f$ is an edge starting and ending at $w$. Therefore the first homology group of $X$ should be a copy of the integers:

```
sage: X.homology(1)                                                          #␣
↪needs sage.modules
Z
```

sage.topology.simplicial_set.**AbstractSimplex**(*dim*, *degeneracies=()*, *underlying=None*, *name=None*, *latex_name=None*)

An abstract simplex, a building block of a simplicial set.

In a simplicial set, a simplex either is non-degenerate or is obtained by applying degeneracy maps to a non-degenerate simplex.

INPUT:

- `dim` – nonnegative integer; the dimension of the underlying non-degenerate simplex
- `degeneracies` – (default: `None`) list or tuple of nonnegative integers, the degeneracies to be applied
- `underlying` – (optional) a non-degenerate simplex to which the degeneracies are being applied
- `name` – (optional) string; a name for this simplex
- `latex_name` – (optional) string; a name for this simplex to use in the LaTeX representation

So to define a simplex formed by applying the degeneracy maps $s_2 s_1$ to a 1-simplex, call `AbstractSimplex(1, (2, 1))`.

Specify `underlying` if you need to keep explicit track of the underlying non-degenerate simplex, for example when computing faces of another simplex. This is mainly for use by the method *AbstractSimplex_class.apply_degeneracies()*.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: AbstractSimplex(3, (3, 1))
s_3 s_1 Delta^3
sage: AbstractSimplex(3, None)
Delta^3
sage: AbstractSimplex(3)
Delta^3
```

Simplices may be named (or renamed), affecting how they are printed:

```
sage: AbstractSimplex(0)
Delta^0
sage: v = AbstractSimplex(0, name='v')
sage: v
v
sage: v.rename('w_0')
sage: v
w_0
sage: latex(v)
w_0
sage: latex(AbstractSimplex(0, latex_name='\\sigma'))
\sigma
```

The simplicial identities are used to put the degeneracies in standard decreasing form:

```
sage: x = AbstractSimplex(0, (0, 0, 0))
sage: x
s_2 s_1 s_0 Delta^0
sage: x.degeneracies()
[2, 1, 0]
```

Use of the `underlying` argument:

```
sage: v = AbstractSimplex(0, name='v')
sage: e = AbstractSimplex(0, (0,), underlying=v)
sage: e
s_0 v
sage: e.nondegenerate() is v
True

sage: e.dimension()
1
sage: e.is_degenerate()
True
```

Distinct non-degenerate simplices are never equal:

```
sage: AbstractSimplex(0, None) == AbstractSimplex(0, None)
False
sage: AbstractSimplex(0, (2,1,0)) == AbstractSimplex(0, (2,1,0))
False

sage: e = AbstractSimplex(0, ((0,)))
sage: f = AbstractSimplex(0, ((0,)))
sage: e == f
False
sage: e.nondegenerate() == f.nondegenerate()
False
```

This means that if, when defining a simplicial set, you specify the faces of a 2-simplex as:

```
(e, e, e)
```

then the faces are the same degenerate vertex, but if you specify the faces as:

```
(AbstractSimplex(0, ((0,))), AbstractSimplex(0, ((0,))), AbstractSimplex(0, ((0,
↪))))
```

then the faces are three different degenerate vertices.

View a command like `AbstractSimplex(0, (2,1,0))` as first constructing `AbstractSimplex(0)` and then applying degeneracies to it, and you always get distinct simplices from different calls to `AbstractSimplex(0)`. On the other hand, if you apply degeneracies to the same non-degenerate simplex, the resulting simplices are equal:

```
sage: v = AbstractSimplex(0)
sage: v.apply_degeneracies(1, 0) == v.apply_degeneracies(1, 0)
True
sage: AbstractSimplex(1, (0,), underlying=v) == AbstractSimplex(1, (0,),␣
↪underlying=v)
True
```

**class** sage.topology.simplicial_set.**AbstractSimplex_class**(*dim*, *degeneracies=()*, *underlying=None*, *name=None*, *latex_name=None*)

Bases: SageObject

A simplex of dimension dim.

INPUT:

- dim – integer; the dimension
- degeneracies – (optional) iterable, the indices of the degeneracy maps
- underlying – (optional) a non-degenerate simplex
- name – (optional) string
- latex_name – (optional) string

Users should not call this directly, but instead use *AbstractSimplex()*. See that function for more documentation.

**apply_degeneracies**(**args*)

Apply the degeneracies given by the arguments args to this simplex.

INPUT:

- args – integer

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: v = AbstractSimplex(0)
sage: e = v.apply_degeneracies(0)
sage: e.nondegenerate() == v
True
sage: f = e.apply_degeneracies(0)
sage: f
s_1 s_0 Delta^0
sage: f.degeneracies()
[1, 0]
sage: f.nondegenerate() == v
True
sage: v.apply_degeneracies(1, 0)
s_1 s_0 Delta^0
```

**degeneracies**()

Return the list of indices for the degeneracy maps for this simplex.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: AbstractSimplex(4, (0,0,0)).degeneracies()
[2, 1, 0]
sage: AbstractSimplex(4, None).degeneracies()
[]
```

**dimension**()

The dimension of this simplex.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: AbstractSimplex(3, (2,1)).dimension()
5
sage: AbstractSimplex(3, None).dimension()
3
sage: AbstractSimplex(7).dimension()
7
```

**is_degenerate**()

Return `True` if this simplex is degenerate.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: AbstractSimplex(3, (2,1)).is_degenerate()
True
sage: AbstractSimplex(3, None).is_degenerate()
False
```

**is_nondegenerate**()

Return `True` if this simplex is non-degenerate.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: AbstractSimplex(3, (2,1)).is_nondegenerate()
False
sage: AbstractSimplex(3, None).is_nondegenerate()
True
sage: AbstractSimplex(5).is_nondegenerate()
True
```

**nondegenerate**()

The non-degenerate simplex underlying this one.

Therefore return itself if this simplex is non-degenerate.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: v = AbstractSimplex(0, name='v')
sage: sigma = v.apply_degeneracies(1, 0)
sage: sigma.nondegenerate()
v
sage: tau = AbstractSimplex(1, (3,2,1))
sage: x = tau.nondegenerate(); x
Delta^1
sage: x == tau.nondegenerate()
True

sage: AbstractSimplex(1, None)
Delta^1
sage: AbstractSimplex(1, None) == x
False
sage: AbstractSimplex(1, None) == tau.nondegenerate()
False
```

**class** sage.topology.simplicial_set.**NonDegenerateSimplex**(*dim*, *name=None*, *latex_name=None*)

Bases: *AbstractSimplex_class*, WithEqualityById

A nondegenerate simplex.

INPUT:

- dim – nonnegative integer; the dimension
- name – (optional) string; a name for this simplex
- latex_name – (optional) string; a name for this simplex to use in the LaTeX representation

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex
sage: v = AbstractSimplex(0, name='v')
sage: v
v
sage: type(v)
<class 'sage.topology.simplicial_set.NonDegenerateSimplex'>
```

Distinct non-degenerate simplices should never be equal, even if they have the same starting data.

```
sage: v == AbstractSimplex(0, name='v')
False
sage: AbstractSimplex(3) == AbstractSimplex(3)
False

sage: from sage.topology.simplicial_set import NonDegenerateSimplex
sage: x = NonDegenerateSimplex(0, name='x')
sage: x == NonDegenerateSimplex(0, name='x')
False
```

sage.topology.simplicial_set.**SimplicialSet**

alias of *SimplicialSet_finite*

**class** sage.topology.simplicial_set.**SimplicialSet_arbitrary**

Bases: Parent

A simplicial set.

A simplicial set $X$ is a collection of sets $X_n$, the *n-simplices*, indexed by the nonnegative integers, together with maps

$$d_i : X_n \to X_{n-1}, \ \ 0 \leq i \leq n \ \text{(face maps)}$$
$$s_j : X_n \to X_{n+1}, \ \ 0 \leq j \leq n \ \text{(degeneracy maps)}$$

satisfying the *simplicial identities*:

$$d_i d_j = d_{j-1} d_i \ \text{ if } i < j$$
$$d_i s_j = s_{j-1} d_i \ \text{ if } i < j$$
$$d_j s_j = 1 = d_{j+1} s_j$$
$$d_i s_j = s_j d_{i-1} \ \text{ if } i > j+1$$
$$s_i s_j = s_{j+1} s_i \ \text{ if } i < j+1$$

This class is not fully implemented and is not intended to be called directly by users. It is intended instead to be used by other classes which inherit from this one. See *SimplicialSet_finite* and Nerve for two examples. In particular, any such class must implement a method n_skeleton – without this, most computations will be

impossible. It must also implement an `__init__` method which should also set the category, so that methods defined at the category level, like `is_pointed` and `is_finite`, work correctly.

Note that the method *subsimplicial_set()* calls `n_skeleton()`, so to avoid circularity, the `n_skeleton()` method should call *simplicial_set_constructions.SubSimplicialSet* directly, not *subsimplicial_set()*.

**alexander_whitney**(*simplex*, *dim_left*)

Return the 'subdivision' of `simplex` in this simplicial set into a pair of simplices.

The left factor should have dimension `dim_left`, so the right factor should have dimension `dim - dim_left`, if `dim` is the dimension of the starting simplex. The results are obtained by applying iterated face maps to `simplex`. Writing $d$ for `dim` and $j$ for `dim_left`: apply $d_{j+1}d_{j+2}...d_d$ to get the left factor, $d_0...d_0$ to get the right factor.

INPUT:

- `dim_left` – integer; the dimension of the left-hand factor

OUTPUT: list containing the triple `(c, left, right)`, where `left` and `right` are the two simplices described above. If either `left` or `right` is degenerate, `c` is 0; otherwise, `c` is 1. This is so that, when used to compute cup products, it is easy to ignore terms which have degenerate factors.

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: sigma = S2.n_cells(2)[0]
sage: S2.alexander_whitney(sigma, 0)
[(1, v_0, sigma_2)]
sage: S2.alexander_whitney(sigma, 1)
[(0, s_0 v_0, s_0 v_0)]
```

**all_n_simplices**(*n*)

Return a list of all simplices, non-degenerate and degenerate, in dimension `n`.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: degen = v.apply_degeneracies(0)
sage: tau = AbstractSimplex(2, name='tau')
sage: Y = SimplicialSet({tau: (degen, degen, degen), w: None})
```

`Y` is the disjoint union of a 2-sphere, with vertex `v` and non-degenerate 2-simplex `tau`, and a point `w`.

```
sage: Y.all_n_simplices(0)
[v, w]
sage: Y.all_n_simplices(1)
[s_0 v, s_0 w]
sage: Y.all_n_simplices(2)
[tau, s_1 s_0 v, s_1 s_0 w]
```

An example involving an infinite simplicial set:

```
sage: C3 = groups.misc.MultiplicativeAbelian([3])                    #␣
↪needs sage.groups
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)                     #␣
↪needs sage.groups
```

(continues on next page)

(continued from previous page)

```
sage: BC3.all_n_simplices(2)                                                #␣
↪needs sage.groups
[f * f,
 f * f^2,
 f^2 * f,
 f^2 * f^2, s_0 f, s_0 f^2, s_1 f, s_1 f^2, s_1 s_0 1]
```

**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 $i$-th Betti number is the rank of the $i$-th homology group.

INPUT:

- `dim` – (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)`.)
- `subcomplex` – (default: `None`) a subcomplex of this cell complex. Compute the Betti numbers of the homology relative to this subcomplex.

**Note**

If this simplicial set is not finite, you must specify dimensions in which to compute Betti numbers via the argument `dim`.

EXAMPLES:

Build the two-sphere as a three-fold join of a two-point space with itself:

```
sage: simplicial_sets.Sphere(5).betti()                                     #␣
↪needs sage.modules
{0: 1, 1: 0, 2: 0, 3: 0, 4: 0, 5: 1}

sage: C3 = groups.misc.MultiplicativeAbelian([3])                           #␣
↪needs sage.groups
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)                            #␣
↪needs sage.groups
sage: BC3.betti(range(4))                                                   #␣
↪needs sage.groups sage.modules
{0: 1, 1: 0, 2: 0, 3: 0}
```

**cartesian_product** (**others*)

Return the product of this simplicial set with `others`.

INPUT:

- `others` – one or several simplicial sets

If $X$ and $Y$ are simplicial sets, then their product $X \times Y$ is defined to be the simplicial set with $n$-simplices $X_n \times Y_n$. See *simplicial_set_constructions.ProductOfSimplicialSets* for more information.

If a simplicial set is constructed as a product, the factors are recorded and are accessible via the method *simplicial_set_constructions.Factors.factors()*. If each factor is finite, then you can also construct the projection maps onto each factor, the wedge as a subcomplex, and the fat wedge as a subcomplex.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, w)})
sage: square = X.product(X)
```

`square` is now the standard triangulation of the square: 4 vertices, 5 edges (the four on the border and the diagonal), 2 triangles:

```
sage: square.f_vector()
[4, 5, 2]

sage: S1 = simplicial_sets.Sphere(1)
sage: T = S1.product(S1)
sage: T.homology(reduced=False)                                                 #␣
↪needs sage.modules
{0: Z, 1: Z x Z, 2: Z}
```

Since `S1` is pointed, so is `T`:

```
sage: S1.is_pointed()
True
sage: S1.base_point()
v_0
sage: T.is_pointed()
True
sage: T.base_point()
(v_0, v_0)

sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: S2xS3 = S2.product(S3)
sage: S2xS3.homology(reduced=False)                                             #␣
↪needs sage.modules
{0: Z, 1: 0, 2: Z, 3: Z, 4: 0, 5: Z}

sage: S2xS3.factors() == (S2, S3)
True
sage: S2xS3.factors() == (S3, S2)
False

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: B.rename('RP^oo')
sage: X = B.product(B, S2); X
RP^oo x RP^oo x S^2
sage: X.factor(1)
RP^oo
sage: X.factors()
(RP^oo, RP^oo, S^2)
```

Projection maps and wedges:

```
sage: S2xS3.projection_map(0)
Simplicial set morphism:
  From: S^2 x S^3
  To:   S^2
  Defn: ...
sage: S2xS3.wedge_as_subset().homology()                                    #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z, 3: Z}
```

In the case of pointed simplicial sets, there is an inclusion of each factor into the product. These are not automatically defined in Sage, but they are easy to construct using identity maps and constant maps and the universal property of the product:

```
sage: one = S2.identity()
sage: const = S2.constant_map(codomain=S3)
sage: S2xS3.universal_property(one, const)
Simplicial set morphism:
  From: S^2
  To:   S^2 x S^3
  Defn: [v_0, sigma_2] --> [(v_0, v_0), (sigma_2, s_1 s_0 v_0)]
```

**cells**(*subcomplex=None*, *max_dim=None*)

Return a dictionary of all non-degenerate simplices.

INPUT:

- `subcomplex` – (optional) a subsimplicial set of this simplicial set. If `subcomplex` is specified, then return the simplices in the quotient by the subcomplex.
- `max_dim` – (default: `None`) if specified, return the non-degenerate simplices of this dimension or smaller. This argument is required if this simplicial set is infinite.

Each key is a dimension, and the corresponding value is the list of simplices in that dimension.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0)
sage: w = AbstractSimplex(0)
sage: S0 = SimplicialSet({v: None, w: None})
sage: S0.cells()
{0: [Delta^0, Delta^0]}

sage: v.rename('v')
sage: w.rename('w')
sage: S0.cells()
{0: [v, w]}

sage: e = AbstractSimplex(1, name='e')
sage: S1 = SimplicialSet({e: (v, v)})
sage: S1.cells()
{0: [v], 1: [e]}

sage: S0.cells(S0.subsimplicial_set([v, w]))
{0: [*]}

sage: X = SimplicialSet({e: (v,w)})
sage: X.cells(X.subsimplicial_set([v, w]))
{0: [*], 1: [e]}
```

Test an infinite example:

```
sage: # needs sage.groups
sage: C3 = groups.misc.MultiplicativeAbelian([3])
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)
sage: BC3.cells(max_dim=2)
{0: [1], 1: [f, f^2], 2: [f * f, f * f^2, f^2 * f, f^2 * f^2]}
sage: BC3.cells()
Traceback (most recent call last):
...
NotImplementedError: this simplicial set may be infinite, so specify max_dim
```

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

Return the normalized chain complex.

INPUT:

- `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` – (default: $\mathbf{Z}$) commutative ring
- `augmented` – boolean (default: `False`); if `True`, return the augmented chain complex (that is, include a class in dimension $-1$ corresponding to the empty cell)
- `cochain` – boolean (default: `False`); if `True`, return the cochain complex (that is, the dual of the chain complex)
- `verbose` – boolean (default: `False`); ignored
- `subcomplex` – (default: `None`) if present, compute the chain complex relative to this subcomplex
- `check` – boolean (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 this simplicial set is not finite, you must specify dimensions in which to compute its chain complex via the argument `dimensions`.

EXAMPLES:

```
sage: simplicial_sets.Sphere(5).chain_complex()                             #␣
↪needs sage.modules
Chain complex with at most 3 nonzero terms over Integer Ring

sage: C3 = groups.misc.MultiplicativeAbelian([3])                           #␣
↪needs sage.groups
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)                            #␣
↪needs sage.groups
sage: BC3.chain_complex(range(4), base_ring=GF(3))                          #␣
↪needs sage.groups sage.modules
Chain complex with at most 4 nonzero terms over Finite Field of size 3
```

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

Return the cohomology of this simplicial set.

INPUT:

- dim – (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 – (default: ZZ) commutative ring; must be ZZ or a field

Other arguments are also allowed, the same as for the *homology()* method – see *cell_complex.GenericCellComplex.homology()* for complete documentation – 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 argument cohomology=True.

**Note**

If this simplicial set is not finite, you must specify dimensions in which to compute homology via the argument dim.

EXAMPLES:

```
sage: simplicial_sets.KleinBottle().homology(1)                                 #␣
↪needs sage.modules
Z x C2
sage: simplicial_sets.KleinBottle().cohomology(1)                               #␣
↪needs sage.modules
Z
sage: simplicial_sets.KleinBottle().cohomology(2)                               #␣
↪needs sage.modules
C2
```

**cone**()

Return the (reduced) cone on this simplicial set.

If this simplicial set $X$ is not pointed, construct the ordinary cone: add a point $v$ (which will become the base point) and for each simplex $\sigma$ in $X$, add both $\sigma$ and a simplex made up of $v$ and $\sigma$ (topologically, form the join of $v$ and $\sigma$).

If this simplicial set is pointed, then construct the reduced cone: take the quotient of the unreduced cone by the 1-simplex connecting the old base point to the new one.

In either case, as long as the simplicial set is finite, it comes equipped in Sage with a map from it into the cone.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, v)})
sage: CX = X.cone()  # unreduced cone, since X not pointed
sage: CX.nondegenerate_simplices()
[*, v, (v,*), e, (e,*)]
sage: CX.base_point()
*
```

$X$ as a subset of the cone, and also the map from $X$, in the unreduced case:

```
sage: CX.base_as_subset()
Simplicial set with 2 non-degenerate simplices
```

(continues on next page)

(continued from previous page)

```
sage: CX.map_from_base()
Simplicial set morphism:
From: Simplicial set with 2 non-degenerate simplices
  To:   Cone of Simplicial set with 2 non-degenerate simplices
  Defn: [v, e] --> [v, e]
```

In the reduced case, only the map from $X$ is available:

```
sage: X = X.set_base_point(v)
sage: CX = X.cone()  # reduced cone
sage: CX.nondegenerate_simplices()
[*, e, (e,*)]
sage: CX.map_from_base()
Simplicial set morphism:
  From: Simplicial set with 2 non-degenerate simplices
  To:   Reduced cone of Simplicial set with 2 non-degenerate simplices
  Defn: [v, e] --> [*, e]
```

**constant_map**(*codomain=None*, *point=None*)

Return a constant map with this simplicial set as its domain.

INPUT:

- `codomain` – (default: `None`) if `None`, the codomain is the standard one-point space constructed by `Point()`. Otherwise, either the codomain must be a pointed simplicial set, in which case the map is constant at the base point, or `point` must be specified.
- `point` – (default: `None`) if specified, it must be a 0-simplex in the codomain, and it will be the target of the constant map

EXAMPLES:

```
sage: S4 = simplicial_sets.Sphere(4)
sage: S4.constant_map()
Simplicial set morphism:
  From: S^4
  To:   Point
  Defn: Constant map at *
sage: S0 = simplicial_sets.Sphere(0)
sage: S4.constant_map(codomain=S0)
Simplicial set morphism:
  From: S^4
  To:   S^0
  Defn: Constant map at v_0

sage: Sigma3 = groups.permutation.Symmetric(3)                                  #␣
↪needs sage.groups
sage: Sigma3.nerve().constant_map()                                             #␣
↪needs sage.groups
Simplicial set morphism:
  From: Nerve of Symmetric group of order 3! as a permutation group
  To:   Point
  Defn: Constant map at *
```

**coproduct**(**others*)

Return the coproduct of this simplicial set with `others`.

INPUT:

- others – one or several simplicial sets

If these simplicial sets are pointed, return their wedge sum; if they are not, return their disjoint union. If some are pointed and some are not, raise an error: it is not clear in which category to work.

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: K = simplicial_sets.KleinBottle()
sage: D3 = simplicial_sets.Simplex(3)
sage: Y = S2.unset_base_point()
sage: Z = K.unset_base_point()

sage: S2.coproduct(K).is_pointed()
True
sage: S2.coproduct(K)
Wedge: (S^2 v Klein bottle)
sage: D3.coproduct(Y, Z).is_pointed()
False
sage: D3.coproduct(Y, Z)
Disjoint union: (3-simplex u Simplicial set with 2 non-degenerate simplices
                 u Simplicial set with 6 non-degenerate simplices)
```

The coproduct comes equipped with an inclusion map from each summand, as long as the summands are all finite:

```
sage: S2.coproduct(K).inclusion_map(0)
Simplicial set morphism:
  From: S^2
  To:   Wedge: (S^2 v Klein bottle)
  Defn: [v_0, sigma_2] --> [*, sigma_2]
sage: D3.coproduct(Y, Z).inclusion_map(2)
Simplicial set morphism:
  From: Simplicial set with 6 non-degenerate simplices
  To:   Disjoint union: (3-simplex
                         u Simplicial set with 2 non-degenerate simplices
                         u Simplicial set with 6 non-degenerate simplices)
  Defn: [Delta_{0,0}, Delta_{1,0}, Delta_{1,1}, Delta_{1,2}, Delta_{2,0},␣
↪Delta_{2,1}]
        --> [Delta_{0,0}, Delta_{1,0}, Delta_{1,1}, Delta_{1,2}, Delta_{2,0},␣
↪Delta_{2,1}]
```

**disjoint_union**(**others*)

Return the disjoint union of this simplicial set with others.

INPUT:

- others – one or several simplicial sets

As long as the factors are all finite, the inclusion map from each factor is available. Any factors which are empty are ignored completely: they do not appear in the list of factors, etc.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: f = AbstractSimplex(1, name='f')
sage: X = SimplicialSet({e: (v, v)})
```

(continues on next page)

(continued from previous page)

```
sage: Y = SimplicialSet({f: (v, w)})
sage: Z = X.disjoint_union(Y)
```

Since `X` and `Y` have simplices in common, Sage uses a copy of `Y` when constructing the disjoint union. Note the name conflict in the list of simplices: `v` appears twice:

```
sage: Z = X.disjoint_union(Y)
sage: Z.nondegenerate_simplices()
[v, v, w, e, f]
```

Factors and inclusion maps:

```
sage: T = simplicial_sets.Torus()
sage: S2 = simplicial_sets.Sphere(2)
sage: A = T.disjoint_union(S2)
sage: A.factors()
(Torus, S^2)
sage: i = A.inclusion_map(0)
sage: i.domain()
Torus
sage: i.codomain()
Disjoint union: (Torus u S^2)
```

Empty factors are ignored:

```
sage: from sage.topology.simplicial_set_examples import Empty
sage: E = Empty()
sage: K = S2.disjoint_union(S2, E, E, S2)
sage: K == S2.disjoint_union(S2, S2)
True
sage: K.factors()
(S^2, S^2, S^2)
```

**face** (*simplex*, *i*)

Return the $i$-th face of `simplex` in this simplicial set.

INPUT:

- `simplex` – a simplex in this simplicial set
- `i` – integer

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: sigma = S2.n_cells(2)[0]
sage: v_0 = S2.n_cells(0)[0]
sage: S2.face(sigma, 0)
s_0 v_0
sage: S2.face(sigma, 0) == v_0.apply_degeneracies(0)
True
sage: S2.face(S2.face(sigma, 0), 0) == v_0
True
```

**faces** (*simplex*)

Return the list of faces of `simplex` in this simplicial set.

INPUT:

- simplex – a simplex in this simplicial set, either degenerate or not

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: sigma = S2.n_cells(2)[0]
sage: S2.faces(sigma)
(s_0 v_0, s_0 v_0, s_0 v_0)
sage: S2.faces(sigma.apply_degeneracies(0))
[sigma_2, sigma_2, s_1 s_0 v_0, s_1 s_0 v_0]

sage: # needs sage.groups
sage: C3 = groups.misc.MultiplicativeAbelian([3])
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)
sage: f2 = BC3.n_cells(1)[1]; f2
f^2
sage: BC3.faces(f2)
(1, 1)
```

**graph**()

Return the 1-skeleton of this simplicial set, as a graph.

EXAMPLES:

```
sage: Delta3 = simplicial_sets.Simplex(3)
sage: G = Delta3.graph()
sage: G.edges(sort=True)
[((0,), (1,), (0, 1)),
 ((0,), (2,), (0, 2)),
 ((0,), (3,), (0, 3)),
 ((1,), (2,), (1, 2)),
 ((1,), (3,), (1, 3)),
 ((2,), (3,), (2, 3))]

sage: T = simplicial_sets.Torus()
sage: T.graph()
Looped multi-graph on 1 vertex
sage: len(T.graph().edges(sort=False))
3

sage: # needs pyparsing
sage: CP3 = simplicial_sets.ComplexProjectiveSpace(3)
sage: G = CP3.graph()
sage: len(G.vertices(sort=False))
1
sage: len(G.edges(sort=False))
0

sage: Sigma3 = groups.permutation.Symmetric(3)                                  #␣
↪needs sage.groups
sage: Sigma3.nerve().is_connected()                                             #␣
↪needs sage.groups
True
```

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

Return the (reduced) homology of this simplicial set.

INPUT:

- dim – (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 – (default: ZZ) commutative ring; must be ZZ or a field

Other arguments are also allowed: see the documentation for *cell_complex.GenericCellComplex.homology()*.

**Note**

If this simplicial set is not finite, you must specify dimensions in which to compute homology via the argument dim.

EXAMPLES:

```
sage: simplicial_sets.Sphere(5).homology()                                  #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: Z}

sage: C3 = groups.misc.MultiplicativeAbelian([3])                           #␣
↪needs sage.groups
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)                            #␣
↪needs sage.groups
sage: BC3.homology(range(4), base_ring=GF(3))                               #␣
↪needs sage.groups sage.modules
{0: Vector space of dimension 0 over Finite Field of size 3,
 1: Vector space of dimension 1 over Finite Field of size 3,
 2: Vector space of dimension 1 over Finite Field of size 3,
 3: Vector space of dimension 1 over Finite Field of size 3}

sage: # needs sage.groups
sage: C2 = groups.misc.MultiplicativeAbelian([2])
sage: BC2 = simplicial_sets.ClassifyingSpace(C2)
sage: BK = BC2.product(BC2)
sage: BK.homology(range(4))                                                 #␣
↪needs sage.modules
{0: 0, 1: C2 x C2, 2: C2, 3: C2 x C2 x C2}
```

**identity**()

Return the identity map on this simplicial set.

EXAMPLES:

```
sage: S3 = simplicial_sets.Sphere(3)
sage: S3.identity()
Simplicial set endomorphism of S^3
  Defn: Identity map

sage: # needs sage.groups
sage: C3 = groups.misc.MultiplicativeAbelian([3])
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)
sage: one = BC3.identity()
sage: [(sigma, one(sigma)) for sigma in BC3.n_cells(2)]
[(f * f, f * f),
 (f * f^2, f * f^2),
```

(continues on next page)

(continued from previous page)

```
 (f^2 * f, f^2 * f),
 (f^2 * f^2, f^2 * f^2)]
```

**is_connected**()

Return `True` if this simplicial set is connected.

EXAMPLES:

```
sage: T = simplicial_sets.Torus()
sage: K = simplicial_sets.KleinBottle()
sage: X = T.disjoint_union(K)
sage: T.is_connected()
True
sage: K.is_connected()
True
sage: X.is_connected()
False
sage: simplicial_sets.Sphere(0).is_connected()
False
```

**is_reduced**()

Return `True` if this simplicial set has only one vertex.

EXAMPLES:

```
sage: simplicial_sets.Sphere(0).is_reduced()
False
sage: simplicial_sets.Sphere(3).is_reduced()
True
```

**join**(**others*)

The join of this simplicial set with `others`.

Not implemented. See https://ncatlab.org/nlab/show/join+of+simplicial+sets for a few descriptions, for anyone interested in implementing this. See also P. J. Ehlers and Tim Porter, Joins for (Augmented) Simplicial Sets, Jour. Pure Applied Algebra, 145 (2000) 37-44 arXiv 9904039.

- `others` – one or several simplicial sets

EXAMPLES:

```
sage: K = simplicial_sets.Simplex(2)
sage: K.join(K)
Traceback (most recent call last):
...
NotImplementedError: joins are not implemented for simplicial sets
```

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

Return the list of cells of dimension `n` of this cell complex. If the optional argument `subcomplex` is present, then return the `n`-dimensional faces in the quotient by this subcomplex.

INPUT:

- `n` – the dimension
- `subcomplex` – (default: `None`) a subcomplex of this cell complex. Return the cells which are in the quotient by this subcomplex.

EXAMPLES:

```
sage: simplicial_sets.Sphere(3).n_cells(3)
[sigma_3]
sage: simplicial_sets.Sphere(3).n_cells(2)
[]
sage: C2 = groups.misc.MultiplicativeAbelian([2])                    #␣
↪needs sage.groups
sage: BC2 = C2.nerve()                                               #␣
↪needs sage.groups
sage: BC2.n_cells(3)                                                 #␣
↪needs sage.groups
[f * f * f]
```

**n_chains**(*n*, *base_ring=Integer Ring*, *cochains=False*)

Return the free module of (normalized) chains in degree `n` over `base_ring`.

This is the free module on the nondegenerate simplices in the given dimension.

INPUT:

- `n` – integer
- `base_ring` – ring (default: **Z**)
- `cochains` – boolean (default: `False`); if `True`, return cochains instead

The only difference between chains and cochains is notation: the generator corresponding to the dual of a simplex `sigma` is written as `'\chi_sigma'` in the group of cochains.

EXAMPLES:

```
sage: S3 = simplicial_sets.Sphere(3)
sage: C = S3.n_chains(3, cochains=True)                              #␣
↪needs sage.modules
sage: list(C.basis())                                                #␣
↪needs sage.modules
[\chi_sigma_3]

sage: # needs sage.groups
sage: Sigma3 = groups.permutation.Symmetric(3)
sage: BSigma3 = simplicial_sets.ClassifyingSpace(Sigma3)
sage: list(BSigma3.n_chains(1).basis())                              #␣
↪needs sage.modules
[(1,2), (1,2,3), (1,3), (1,3,2), (2,3)]
sage: list(BSigma3.n_chains(1, cochains=True).basis())               #␣
↪needs sage.modules
[\chi_(1,2), \chi_(1,2,3), \chi_(1,3), \chi_(1,3,2), \chi_(2,3)]
```

**nondegenerate_simplices**(*max_dim=None*)

Return the sorted list of non-degenerate simplices in this simplicial set.

INPUT:

- `max_dim` – (default: `None`) if specified, return the non-degenerate simplices of this dimension or smaller. This argument is required if this simplicial set is infinite.

The sorting is in increasing order of dimension, and within each dimension, by the name (if present) of each simplex.

**Note**

The sorting is done when the simplicial set is constructed, so changing the name of a simplex after construction will not affect the ordering.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0)
sage: w = AbstractSimplex(0)
sage: S0 = SimplicialSet({v: None, w: None})
sage: S0.nondegenerate_simplices()
[Delta^0, Delta^0]
```

Name the vertices and reconstruct the simplicial set: they should be ordered alphabetically:

```
sage: v.rename('v')
sage: w.rename('w')
sage: S0 = SimplicialSet({v: None, w: None})
sage: S0.nondegenerate_simplices()
[v, w]
```

Rename but do not reconstruct the set; the ordering does not take the new names into account:

```
sage: v.rename('z')
sage: S0.nondegenerate_simplices() # old ordering is used
[z, w]

sage: X0 = SimplicialSet({v: None, w: None})
sage: X0.nondegenerate_simplices() # new ordering is used
[w, z]
```

Test an infinite example:

```
sage: # needs sage.groups
sage: C3 = groups.misc.MultiplicativeAbelian([3])
sage: BC3 = simplicial_sets.ClassifyingSpace(C3)
sage: BC3.nondegenerate_simplices(2)
[1, f, f^2, f * f, f * f^2, f^2 * f, f^2 * f^2]
sage: BC3.nondegenerate_simplices()
Traceback (most recent call last):
...
NotImplementedError: this simplicial set may be infinite, so specify max_dim
```

**product** (**others*)

Return the product of this simplicial set with `others`.

INPUT:

- `others` – one or several simplicial sets

If $X$ and $Y$ are simplicial sets, then their product $X \times Y$ is defined to be the simplicial set with $n$-simplices $X_n \times Y_n$. See *simplicial_set_constructions.ProductOfSimplicialSets* for more information.

If a simplicial set is constructed as a product, the factors are recorded and are accessible via the method *simplicial_set_constructions.Factors.factors()*. If each factor is finite, then you can

also construct the projection maps onto each factor, the wedge as a subcomplex, and the fat wedge as a subcomplex.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, w)})
sage: square = X.product(X)
```

`square` is now the standard triangulation of the square: 4 vertices, 5 edges (the four on the border and the diagonal), 2 triangles:

```
sage: square.f_vector()
[4, 5, 2]

sage: S1 = simplicial_sets.Sphere(1)
sage: T = S1.product(S1)
sage: T.homology(reduced=False)                                                 #␣
↪needs sage.modules
{0: Z, 1: Z x Z, 2: Z}
```

Since `S1` is pointed, so is `T`:

```
sage: S1.is_pointed()
True
sage: S1.base_point()
v_0
sage: T.is_pointed()
True
sage: T.base_point()
(v_0, v_0)

sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: S2xS3 = S2.product(S3)
sage: S2xS3.homology(reduced=False)                                             #␣
↪needs sage.modules
{0: Z, 1: 0, 2: Z, 3: Z, 4: 0, 5: Z}

sage: S2xS3.factors() == (S2, S3)
True
sage: S2xS3.factors() == (S3, S2)
False

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: B.rename('RP^oo')
sage: X = B.product(B, S2); X
RP^oo x RP^oo x S^2
sage: X.factor(1)
RP^oo
sage: X.factors()
(RP^oo, RP^oo, S^2)
```

Projection maps and wedges:

```
sage: S2xS3.projection_map(0)
Simplicial set morphism:
  From: S^2 x S^3
  To:   S^2
  Defn: ...
sage: S2xS3.wedge_as_subset().homology()                                    #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z, 3: Z}
```

In the case of pointed simplicial sets, there is an inclusion of each factor into the product. These are not automatically defined in Sage, but they are easy to construct using identity maps and constant maps and the universal property of the product:

```
sage: one = S2.identity()
sage: const = S2.constant_map(codomain=S3)
sage: S2xS3.universal_property(one, const)
Simplicial set morphism:
  From: S^2
  To:   S^2 x S^3
  Defn: [v_0, sigma_2] --> [(v_0, v_0), (sigma_2, s_1 s_0 v_0)]
```

**pullback**(**maps*)

Return the pullback obtained from given `maps`.

INPUT:

- `maps` – several maps of simplicial sets, each of which has this simplicial set as its codomain

If only a single map $f : X \to Y$ is given, then return $X$. If more than one map is given, say $f_i : X_i \to Y$ for $0 \leq i \leq m$, then return the pullback defined by those maps. If no maps are given, return the one-point simplicial set.

In addition to the defining maps $f_i$ used to construct the pullback $P$, there are also maps $\bar{f}_i : P \to X_i$, which we refer to as *structure maps* or *projection maps*. The pullback also has a universal property: given maps $g_i : Z \to X_i$ such that $f_i g_i = f_j g_j$ for all $i, j$, then there is a unique map $g : Z \to P$ making the appropriate diagram commute: that is, $\bar{f}_i g = g_i$ for all $i$. For example, given maps $f : X \to Y$ and $g : X \to Z$, there is an induced map $g : X \to Y \times Z$.

In Sage, a pullback is equipped with its defining maps, and as long as the simplicial sets involved are finite, you can also access the structure maps and the universal property.

EXAMPLES:

Construct a product as a pullback:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: pt = simplicial_sets.Point()
sage: P = pt.pullback(S2.constant_map(), S2.constant_map())
sage: P.homology(2)                                                         #␣
↪needs sage.modules
Z x Z
```

If the pullback is defined via maps $f_i : X_i \to Y$, then there are structure maps $\bar{f}_i : Y_i \to P$. The structure maps are only available in Sage when all of the maps involved have finite domains.

```
sage: S2 = simplicial_sets.Sphere(2)
sage: one = S2.Hom(S2).identity()
sage: P = S2.pullback(one, one)
sage: P.homology()                                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
{0: 0, 1: 0, 2: Z}

sage: P.defining_map(0) == one
True
sage: P.structure_map(1)
Simplicial set morphism:
  From: Pullback of maps:
  Simplicial set endomorphism of S^2
    Defn: Identity map
  Simplicial set endomorphism of S^2
    Defn: Identity map
  To:   S^2
  Defn: [(v_0, v_0), (sigma_2, sigma_2)] --> [v_0, sigma_2]
sage: P.structure_map(0).domain() == P
True
sage: P.structure_map(0).codomain() == S2
True
```

The universal property:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: T = S1.product(S1)
sage: K = T.factor(0, as_subset=True)
sage: f = S1.Hom(T)({S1.n_cells(0)[0]: K.n_cells(0)[0],
....:                S1.n_cells(1)[0]: K.n_cells(1)[0]})
sage: D = S1.cone()           # the cone C(S^1)
sage: g = D.map_from_base()   # map from S^1 to C(S^1)
sage: P = T.product(D)
sage: h = P.universal_property(f, g)
sage: h.domain() == S1
True
sage: h.codomain() == P
True
```

**pushout** (**maps*)

Return the pushout obtained from given `maps`.

INPUT:

- `maps` – several maps of simplicial sets, each of which has this simplicial set as its domain

If only a single map $f : X \to Y$ is given, then return $Y$. If more than one map is given, say $f_i : X \to Y_i$ for $0 \leq i \leq m$, then return the pushout defined by those maps. If no maps are given, return the empty simplicial set.

In addition to the defining maps $f_i$ used to construct the pushout $P$, there are also maps $\bar{f}_i : Y_i \to P$, which we refer to as *structure maps*. The pushout also has a universal property: given maps $g_i : Y_i \to Z$ such that $g_i f_i = g_j f_j$ for all $i$, $j$, then there is a unique map $g : P \to Z$ making the appropriate diagram commute: that is, $g\bar{f}_i = g_i$ for all $i$.

In Sage, a pushout is equipped with its defining maps, and as long as the simplicial sets involved are finite, you can also access the structure maps and the universal property.

EXAMPLES:

Construct the 4-sphere as a quotient of a 4-simplex:

```
sage: K = simplicial_sets.Simplex(4)
sage: L = K.n_skeleton(3)
sage: S4 = L.pushout(L.constant_map(), L.inclusion_map()); S4
Pushout of maps:
  Simplicial set morphism:
    From: Simplicial set with 30 non-degenerate simplices
    To:   Point
    Defn: Constant map at *
  Simplicial set morphism:
    From: Simplicial set with 30 non-degenerate simplices
    To:   4-simplex
    Defn: [(0,), (1,), (2,), (3,), (4,),
           (0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4),
           (2, 3), (2, 4), (3, 4),
           (0, 1, 2), (0, 1, 3), (0, 1, 4), (0, 2, 3), (0, 2, 4),
           (0, 3, 4), (1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4),
           (0, 1, 2, 3), (0, 1, 2, 4), (0, 1, 3, 4), (0, 2, 3, 4),
           (1, 2, 3, 4)]
           --> [(0,), (1,), (2,), (3,), (4,),
                (0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4),
                (2, 3), (2, 4), (3, 4),
                (0, 1, 2), (0, 1, 3), (0, 1, 4), (0, 2, 3), (0, 2, 4), (0, 3,␣
↪4),
                (1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4),
                (0, 1, 2, 3), (0, 1, 2, 4), (0, 1, 3, 4), (0, 2, 3, 4), (1, 2,
↪ 3, 4)]
sage: len(S4.nondegenerate_simplices())
2
sage: S4.homology(4)                                                          #␣
↪needs sage.modules
Z
```

The associated maps:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: T = S1.product(S1)
sage: K = T.factor(0, as_subset=True)
sage: W = S1.wedge(T)  # wedge, constructed as a pushout
sage: W.defining_map(1)
Simplicial set morphism:
  From: Point
  To:   S^1 x S^1
  Defn: Constant map at (v_0, v_0)
sage: W.structure_map(0)
Simplicial set morphism:
  From: S^1
  To:   Wedge: (S^1 v S^1 x S^1)
  Defn: [v_0, sigma_1] --> [*, sigma_1]

sage: f = S1.Hom(T)({S1.n_cells(0)[0]: K.n_cells(0)[0],
....:                S1.n_cells(1)[0]: K.n_cells(1)[0]})
```

The maps $f : S^1 \to T$ and $1 : T \to T$ induce a map $S^1 \vee T \to T$:

```
sage: g = W.universal_property(f, Hom(T,T).identity())
sage: g.domain() == W
True
```

(continues on next page)

(continued from previous page)

```
sage: g.codomain() == T
True
```

**quotient**(*subcomplex*, *vertex_name='*'*)

Return the quotient of this simplicial set by `subcomplex`.

That is, `subcomplex` is replaced by a vertex.

INPUT:

- `subcomplex` – subsimplicial set of this simplicial set, or a list, tuple, or set of simplices defining a subsimplicial set
- `vertex_name` – string (default: `'*'`); name to be given to the new vertex

In Sage, from a quotient simplicial set, you can recover the ambient space, the subcomplex, and (if the ambient space is finite) the quotient map.

Base points: if the original simplicial set has a base point not contained in `subcomplex` and if the original simplicial set is finite, then use its image as the base point for the quotient. In all other cases, `*` is the base point.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: f = AbstractSimplex(1, name='f')
sage: X = SimplicialSet({e: (v, w), f: (v, w)})
sage: Y = X.quotient([f])
sage: Y.nondegenerate_simplices()
[*, e]
sage: Y.homology(1)                                                             #␣
↪needs sage.modules
Z

sage: E = SimplicialSet({e: (v, w)})
sage: Z = E.quotient([v, w])
sage: Z.nondegenerate_simplices()
[*, e]
sage: Z.homology(1)                                                             #␣
↪needs sage.modules
Z

sage: F = E.quotient([v])
sage: F.nondegenerate_simplices()
[*, w, e]
sage: F.base_point()
*

sage: # needs sage.groups
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: RP2 = RP5.n_skeleton(2)
sage: RP5_2 = RP5.quotient(RP2)
sage: RP5_2.homology(base_ring=GF(2))                                           #␣
↪needs sage.modules
{0: Vector space of dimension 0 over Finite Field of size 2,
```

(continues on next page)

(continued from previous page)

```
 1: Vector space of dimension 0 over Finite Field of size 2,
 2: Vector space of dimension 0 over Finite Field of size 2,
 3: Vector space of dimension 1 over Finite Field of size 2,
 4: Vector space of dimension 1 over Finite Field of size 2,
 5: Vector space of dimension 1 over Finite Field of size 2}
sage: RP5_2.ambient()
RP^5
sage: RP5_2.subcomplex()
Simplicial set with 3 non-degenerate simplices
sage: RP5_2.quotient_map()
Simplicial set morphism:
  From: RP^5
  To:   Quotient: (RP^5/Simplicial set with 3 non-degenerate simplices)
  Defn: [1, f, f * f, f * f * f, f * f * f * f, f * f * f * f * f]
        --> [*, s_0 *, s_1 s_0 *, f * f * f, f * f * f * f, f * f * f * f * f]
```

Behavior of base points:

```
sage: K = simplicial_sets.Simplex(3)
sage: K.is_pointed()
False
sage: L = K.subsimplicial_set([K.n_cells(1)[-1]])
sage: L.nondegenerate_simplices()
[(2,), (3,), (2, 3)]
sage: K.quotient([K.n_cells(1)[-1]]).base_point()
*

sage: K = K.set_base_point(K.n_cells(0)[0])
sage: K.base_point()
(0,)
sage: L = K.subsimplicial_set([K.n_cells(1)[-1]])
sage: L.nondegenerate_simplices()
[(2,), (3,), (2, 3)]
sage: K.quotient(L).base_point()
(0,)
```

**reduce**()

Reduce this simplicial set.

That is, take the quotient by a spanning tree of the 1-skeleton, so that the resulting simplicial set has only one vertex. This only makes sense if the simplicial set is connected, so raise an error if not. If already reduced, return itself.

EXAMPLES:

```
sage: K = simplicial_sets.Simplex(2)
sage: K.is_reduced()
False
sage: X = K.reduce()
sage: X.is_reduced()
True
```

`X` is reduced, so calling `reduce` on it again returns `X` itself:

```
sage: X is X.reduce()
True
```

(continues on next page)

(continued from previous page)

```
sage: K is K.reduce()
False
```

Raise an error for disconnected simplicial sets:

```
sage: S0 = simplicial_sets.Sphere(0)
sage: S0.reduce()
Traceback (most recent call last):
...
ValueError: this simplicial set is not connected
```

**rename_latex**(*s*)

Rename or set the LaTeX name for this simplicial set.

INPUT:

- `s` – string; the LaTeX representation. Or `s` can be `None`, in which case the LaTeX name is unset.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0)
sage: X = SimplicialSet({v: None}, latex_name='*')
sage: latex(X)
*
sage: X.rename_latex('x_0')
sage: latex(X)
x_0
```

**subsimplicial_set**(*simplices*)

Return the sub-simplicial set of this simplicial set determined by `simplices`, a set of nondegenerate simplices.

INPUT:

- `simplices` – set, list, or tuple of nondegenerate simplices in this simplicial set, or a simplicial complex – see below.

Each sub-simplicial set comes equipped with an inclusion map to its ambient space, and you can easily recover its ambient space.

If `simplices` is a simplicial complex, then the original simplicial set should itself have been converted from a simplicial complex, and `simplices` should be a subcomplex of that.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: f = AbstractSimplex(1, name='f')

sage: X = SimplicialSet({e: (v, w), f: (w, v)})
sage: Y = X.subsimplicial_set([e]); Y
Simplicial set with 3 non-degenerate simplices
sage: Y.nondegenerate_simplices()
[v, w, e]
```

(continues on next page)

(continued from previous page)

```
sage: S3 = simplicial_complexes.Sphere(3)
sage: K = SimplicialSet(S3)
sage: tau = K.n_cells(3)[0]
sage: tau.dimension()
3
sage: K.subsimplicial_set([tau])
Simplicial set with 15 non-degenerate simplices
```

A subsimplicial set knows about its ambient space and the inclusion map into it:

```
sage: # needs sage.groups
sage: RP4 = simplicial_sets.RealProjectiveSpace(4)
sage: M = RP4.n_skeleton(2); M
Simplicial set with 3 non-degenerate simplices
sage: M.ambient_space()
RP^4
sage: M.inclusion_map()
Simplicial set morphism:
  From: Simplicial set with 3 non-degenerate simplices
  To:   RP^4
  Defn: [1, f, f * f] --> [1, f, f * f]
```

An infinite ambient simplicial set:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: BxB = B.product(B)
sage: BxB.n_cells(2)[5:]
[(s_0 f, s_1 f), (s_1 f, f * f), (s_1 f, s_0 f), (s_1 s_0 1, f * f)]
sage: BxB.subsimplicial_set(BxB.n_cells(2)[5:])
Simplicial set with 8 non-degenerate simplices
```

**suspension**(*n=1*)

Return the (reduced) $n$-th suspension of this simplicial set.

INPUT:

- `n` – integer (default: 1); suspend this many times

If this simplicial set $X$ is not pointed, return the suspension: the quotient $CX/X$, where $CX$ is the (ordinary, unreduced) cone on $X$. If $X$ is pointed, then use the reduced cone instead, and so return the reduced suspension.

EXAMPLES:

```
sage: # needs sage.groups
sage: RP4 = simplicial_sets.RealProjectiveSpace(4)
sage: S1 = simplicial_sets.Sphere(1)
sage: SigmaRP4 = RP4.suspension()
sage: S1_smash_RP4 = S1.smash_product(RP4)
sage: SigmaRP4.homology() == S1_smash_RP4.homology()
True
```

The version of the suspension obtained by the smash product is typically less efficient than the reduced suspension produced here:

```
sage: SigmaRP4.f_vector()                                                    #␣
↪needs sage.groups
[1, 0, 1, 1, 1, 1]
sage: S1_smash_RP4.f_vector()                                                #␣
↪needs sage.groups
[1, 1, 4, 6, 8, 5]
```

**wedge**(**others*)

Return the wedge sum of this pointed simplicial set with `others`.

- `others` – one or several simplicial sets

This constructs the quotient of the disjoint union in which the base points of all of the simplicial sets have been identified. This is the coproduct in the category of pointed simplicial sets.

This raises an error if any of the factors is not pointed.

From the wedge, you can access the factors, and if the simplicial sets involved are all finite, you can also access the inclusion map of each factor into the wedge, as well as the projection map onto each factor.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: e = AbstractSimplex(1, name='e')
sage: w = AbstractSimplex(0, name='w')
sage: f = AbstractSimplex(1, name='f')
sage: X = SimplicialSet({e: (v, v)}, base_point=v)
sage: Y = SimplicialSet({f: (w, w)}, base_point=w)
sage: W = X.wedge(Y)
sage: W.nondegenerate_simplices()
[*, e, f]
sage: W.homology()                                                           #␣
↪needs sage.modules
{0: 0, 1: Z x Z}
sage: S2 = simplicial_sets.Sphere(2)
sage: X.wedge(S2).homology(reduced=False)                                    #␣
↪needs sage.modules
{0: Z, 1: Z, 2: Z}
sage: X.wedge(X).nondegenerate_simplices()
[*, e, e]

sage: S3 = simplicial_sets.Sphere(3)
sage: W = S2.wedge(S3, S2)
sage: W.inclusion_map(2)
Simplicial set morphism:
  From: S^2
  To:   Wedge: (S^2 v S^3 v S^2)
  Defn: [v_0, sigma_2] --> [*, sigma_2]
sage: W.projection_map(1)
Simplicial set morphism:
  From: Wedge: (S^2 v S^3 v S^2)
  To:   Quotient: (Wedge: (S^2 v S^3 v S^2)/Simplicial set with 3 non-
↪degenerate simplices)
  Defn: [*, sigma_2, sigma_2, sigma_3] --> [*, s_1 s_0 *, s_1 s_0 *, sigma_3]
```

Note that the codomain of the projection map is not identical to the original `S2`, but is instead a quotient of the wedge which is isomorphic to `S2`:

```
sage: S2.f_vector()
[1, 0, 1]
sage: W.projection_map(2).codomain().f_vector()
[1, 0, 1]
sage: (W.projection_map(2) * W.inclusion_map(2)).is_bijective()
True
```

**class** sage.topology.simplicial_set.**SimplicialSet_finite**(*data*, *base_point=None*, *name=None*, *check=True*, *category=None*, *latex_name=None*)

Bases: *SimplicialSet_arbitrary*, *GenericCellComplex*

A finite simplicial set.

A simplicial set $X$ is a collection of sets $X_n$, the *n-simplices*, indexed by the nonnegative integers, together with face maps $d_i$ and degeneracy maps $s_j$. A simplex is *degenerate* if it is in the image of some $s_j$, and a simplicial set is *finite* if there are only finitely many non-degenerate simplices.

INPUT:

- `data` – the data defining the simplicial set; see below for details
- `base_point` – (default: `None`) 0-simplex in this simplicial set, its base point
- `name` – string (default: `None`); the name of the simplicial set
- `check` – boolean (default: `True`); if `True`, check the simplicial identity on the face maps when defining the simplicial set
- `category` – (default: `None`) the category in which to define this simplicial set. The default is either finite simplicial sets or finite pointed simplicial sets, depending on whether a base point is defined.
- `latex_name` – string (default: `None`); the LaTeX representation of the simplicial set

`data` should have one of the following forms: it could be a simplicial complex or $\Delta$-complex, in case it is converted to a simplicial set. Alternatively, it could be a dictionary. The keys are the nondegenerate simplices of the simplicial set, and the value corresponding to a simplex $\sigma$ is a tuple listing the faces of $\sigma$. The 0-dimensional simplices may be omitted from `data` if they (or their degeneracies) are faces of other simplices; otherwise they must be included with value `None`.

See *simplicial_set* and the methods for simplicial sets for more information and examples.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: u = AbstractSimplex(0, name='u')
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: f = AbstractSimplex(1, name='f')
```

In the following simplicial set, `u` is an isolated vertex:

```
sage: X = SimplicialSet({e: (v,w), f: (w,w), u: None})
sage: X
Simplicial set with 5 non-degenerate simplices
sage: X.rename('X')
sage: X
X
```

(continues on next page)

(continued from previous page)

```
sage: X = SimplicialSet({e: (v,w), f: (w,w), u: None}, name='Y')
sage: X
Y
```

**algebraic_topological_model**(*base_ring=None*)

Return the algebraic topological model for this simplicial set with coefficients in `base_ring`.

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

INPUT:

- `base_ring` – coefficient ring (default: `QQ`); must be a field

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

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

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

OUTPUT: a pair consisting of

- chain contraction `phi` associated to $C$, $M$, $\pi$, and $\iota$
- the chain complex $M$

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

EXAMPLES:

```
sage: RP2 = simplicial_sets.RealProjectiveSpace(2)                    #␣
↪needs sage.groups
sage: phi, M = RP2.algebraic_topological_model(GF(2))                 #␣
↪needs sage.groups
sage: M.homology()                                                    #␣
↪needs sage.groups sage.modules
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}

sage: T = simplicial_sets.Torus()
sage: phi, M = T.algebraic_topological_model(QQ)                      #␣
↪needs sage.modules
sage: M.homology()                                                    #␣
↪needs sage.modules
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

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

Return the normalized chain complex.

INPUT:

- `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` – commutative ring (default: **Z**)
- `augmented` – boolean (default: `False`); if `True`, return the augmented chain complex (that is, include a class in dimension $-1$ corresponding to the empty cell).
- `cochain` – boolean (default: `False`); if `True`, return the cochain complex (that is, the dual of the chain complex).
- `verbose` – boolean (default: `False`); ignored
- `subcomplex` – (default: `None`) if present, compute the chain complex relative to this subcomplex
- `check` – boolean (default: `False`); if `True`, make sure that the chain complex is actually a chain complex: the differentials are composable and their product is zero.

The normalized chain complex of a simplicial set is isomorphic to the chain complex obtained by modding out by degenerate simplices, and the latter is what is actually constructed here.

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0)
sage: degen = v.apply_degeneracies(1, 0) # s_1 s_0 applied to v
sage: sigma = AbstractSimplex(3)
sage: S3 = SimplicialSet({sigma: (degen, degen, degen, degen)})  # the 3-
↪sphere
sage: S3.chain_complex().homology()
{0: Z, 3: Z}
sage: S3.chain_complex(augmented=True).homology()
{-1: 0, 0: 0, 3: Z}
sage: S3.chain_complex(dimensions=range(3), base_ring=QQ).homology()
{0: Vector space of dimension 1 over Rational Field}

sage: RP5 = simplicial_sets.RealProjectiveSpace(5)                          #␣
↪needs sage.groups
sage: RP2 = RP5.n_skeleton(2)                                               #␣
↪needs sage.groups
sage: RP5.chain_complex(subcomplex=RP2).homology()                          #␣
↪needs sage.groups sage.modules
{0: Z, 3: C2, 4: 0, 5: Z}
```

**euler_characteristic**()

Return the Euler characteristic of this simplicial set: the alternating sum over $n \geq 0$ of the number of nondegenerate $n$-simplices.

EXAMPLES:

```
sage: simplicial_sets.RealProjectiveSpace(4).euler_characteristic()         #␣
↪needs sage.groups
1
sage: simplicial_sets.Sphere(6).euler_characteristic()
2
sage: simplicial_sets.KleinBottle().euler_characteristic()
0
```

**f_vector**()

Return the list of the number of non-degenerate simplices in each dimension.

Unlike for some other cell complexes in Sage, this does not include the empty simplex in dimension $-1$; thus its $i$-th entry is the number of $i$-dimensional simplices.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0)
sage: w = AbstractSimplex(0)
sage: S0 = SimplicialSet({v: None, w: None})
sage: S0.f_vector()
[2]

sage: e = AbstractSimplex(1)
sage: S1 = SimplicialSet({e: (v, v)})
sage: S1.f_vector()
[1, 1]
sage: simplicial_sets.Sphere(3).f_vector()
[1, 0, 0, 1]
```

**face_data**()

Return the face-map data – a dictionary – defining this simplicial set.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, w)})
sage: X.face_data()[e]
(v, w)

sage: Y = SimplicialSet({v: None, w: None})
sage: v in Y.face_data()
True
sage: Y.face_data()[v] is None
True
```

**n_skeleton**($n$)

Return the $n$-skeleton of this simplicial set.

That is, the subsimplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- `n` – the dimension

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: degen = v.apply_degeneracies(0)
sage: tau = AbstractSimplex(2, name='tau')
sage: Y = SimplicialSet({tau: (degen, degen, degen), w: None})
```

`Y` is the disjoint union of a 2-sphere, with vertex `v` and non-degenerate 2-simplex `tau`, and a point `w`.

```
sage: Y.nondegenerate_simplices()
[v, w, tau]
sage: Y.n_skeleton(1).nondegenerate_simplices()
[v, w]
sage: Y.n_skeleton(2).nondegenerate_simplices()
[v, w, tau]
```

sage.topology.simplicial_set.**all_degeneracies**(*n*, *l=1*)

Return list of all composites of degeneracies (written in "admissible" form, i.e., as a strictly decreasing sequence) of length $l$ on an $n$-simplex.

INPUT:

- `n`, `l` – integers

On an $n$-simplex, one may apply the degeneracies $s_i$ for $0 \leq i \leq n$. Then on the resulting $n+1$-simplex, one may apply $s_i$ for $0 \leq i \leq n+1$, and so on. But one also has to take into account the simplicial identity

$$s_i s_j = s_{j+1} s_i \text{ if } i \leq j.$$

There are $\binom{l+n}{n}$ such composites: each non-degenerate $n$-simplex leads to $\binom{l+n}{n}$ degenerate $l+n$ simplices.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import all_degeneracies
sage: all_degeneracies(0, 3)
{(2, 1, 0)}
sage: all_degeneracies(1, 1)
{(0,), (1,)}
sage: all_degeneracies(1, 3)
{(2, 1, 0), (3, 1, 0), (3, 2, 0), (3, 2, 1)}
```

sage.topology.simplicial_set.**face_degeneracies**(*m*, *I*)

Return the result of applying the face map $d_m$ to the iterated degeneracy $s_I = s_{i_1} s_{i_2} \ldots s_{i_n}$.

INPUT:

- `m` – integer
- `I` – tuple `(i_1, i_2, ..., i_n)` of integers; we assume that this sequence is strictly decreasing

Using the simplicial identities (see *simplicial_set*), we can rewrite

$$d_m s_{i_1} s_{i_2} \ldots s_{i_n}$$

in one of the forms

$$s_{j_1} s_{j_2} \ldots s_{j_n} d_t, \quad s_{j_1} s_{j_2} \ldots s_{j_{n-1}}.$$

OUTPUT: the pair `(J, t)` or `(J, None)`; `J` is returned as a list

EXAMPLES:

```
sage: from sage.topology.simplicial_set import face_degeneracies
sage: face_degeneracies(0, (1, 0))
([0], None)
sage: face_degeneracies(1, (1, 0))
([0], None)
sage: face_degeneracies(2, (1, 0))
```

(continues on next page)

(continued from previous page)

```
([0], None)
sage: face_degeneracies(3, (1, 0))
([1, 0], 1)
sage: face_degeneracies(3, ())
([], 3)
```

sage.topology.simplicial_set.**shrink_simplicial_complex**(*K*)

Convert the simplicial complex K to a "small" simplicial set.

First convert K naively, then mod out by a large contractible subcomplex, as found by simplicial_complex.SimplicialComplex._contractible_subcomplex(). This will produce a simplicial set no larger than, and sometimes much smaller than, the initial simplicial complex.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import shrink_simplicial_complex
sage: K = simplicial_complexes.Simplex(3)
sage: X = shrink_simplicial_complex(K)
sage: X.f_vector()
[1]

sage: Y = simplicial_complexes.Sphere(2)
sage: S2 = shrink_simplicial_complex(Y); S2
Quotient: (Simplicial set with
           14 non-degenerate simplices/Simplicial set with
                                       13 non-degenerate simplices)
sage: S2.f_vector()
[1, 0, 1]
sage: S2.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z}

sage: Z = simplicial_complexes.SurfaceOfGenus(3)
sage: Z.f_vector()
[1, 15, 57, 38]
sage: Z.homology()                                                           #␣
↪needs sage.modules
{0: 0, 1: Z^6, 2: Z}
sage: M = shrink_simplicial_complex(Z)
sage: M.f_vector()  # random
[1, 32, 27]
sage: M.homology()                                                           #␣
↪needs sage.modules
{0: 0, 1: Z^6, 2: Z}
```

sage.topology.simplicial_set.**standardize_degeneracies**(**L*)

Return list of indices of degeneracy maps in standard (decreasing) order.

INPUT:

- L – list of integers representing a composition of degeneracies in a simplicial set

OUTPUT:

an equivalent list of degeneracies, standardized to be written in decreasing order, using the simplicial identity

$$s_i s_j = s_{j+1} s_i \text{ if } i \leq j.$$

For example, $s_0 s_2 = s_3 s_0$ and $s_0 s_0 = s_1 s_0$.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import standardize_degeneracies
sage: standardize_degeneracies(0, 0)
(1, 0)
sage: standardize_degeneracies(0, 0, 0, 0)
(3, 2, 1, 0)
sage: standardize_degeneracies(1, 2)
(3, 1)
```

sage.topology.simplicial_set.**standardize_face_maps**(*L)

Return list of indices of face maps in standard (non-increasing) order.

INPUT:

- L – list of integers representing a composition of face maps in a simplicial set

OUTPUT:

an equivalent list of face maps, standardized to be written in non-increasing order, using the simplicial identity

$$d_i d_j = d_{j-1} d_i \text{ if } i < j.$$

For example, $d_0 d_2 = d_1 d_0$ and $d_0 d_1 = d_0 d_0$.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import standardize_face_maps
sage: standardize_face_maps(0, 1)
(0, 0)
sage: standardize_face_maps(0, 2)
(1, 0)
sage: standardize_face_maps(1, 3, 5)
(3, 2, 1)
```

CHAPTER

EIGHT

# METHODS OF CONSTRUCTING SIMPLICIAL SETS

This implements various constructions on simplicial sets: subsimplicial sets, pullbacks, products, pushouts, quotients, wedges, disjoint unions, smash products, cones, and suspensions. The best way to access these is with methods attached to simplicial sets themselves, as in the following.

EXAMPLES:

```
sage: K = simplicial_sets.Simplex(1)
sage: square = K.product(K)

sage: K = simplicial_sets.Simplex(1)
sage: endpoints = K.n_skeleton(0)
sage: circle = K.quotient(endpoints)
```

The mapping cone of a morphism of simplicial sets is constructed as a pushout:

```
sage: eta = simplicial_sets.HopfMap()
sage: CP2 = eta.mapping_cone()
sage: type(CP2)
<class 'sage.topology.simplicial_set_constructions.PushoutOfSimplicialSets_finite_
↪with_category'>
```

See the main documentation for simplicial sets, as well as for the classes for pushouts, pullbacks, etc., for more details.

Many of the classes defined here inherit from `sage.structure.unique_representation.UniqueRepresentation`. This means that they produce identical output if given the same input, so for example, if `K` is a simplicial set, calling `K.suspension()` twice returns the same result both times:

```
sage: CP2.suspension() is CP2.suspension()
True
```

So on one hand, a command like `simplicial_sets.Sphere(2)` constructs a distinct copy of a 2-sphere each time it is called; on the other, once you have constructed a 2-sphere, then constructing its cone, its suspension, its product with another simplicial set, etc., will give you the same result each time:

```
sage: simplicial_sets.Sphere(2) == simplicial_sets.Sphere(2)
False
sage: S2 = simplicial_sets.Sphere(2)
sage: S2.product(S2) == S2.product(S2)
True
sage: S2.disjoint_union(CP2, S2) == S2.disjoint_union(CP2, S2)
True
```

AUTHORS:

- John H. Palmieri (2016-07)

**class** sage.topology.simplicial_set_constructions.**ConeOfSimplicialSet**(*base*)

Bases: *SimplicialSet_arbitrary*, UniqueRepresentation

Return the unreduced cone on a finite simplicial set.

INPUT:

- base – return the cone on this simplicial set

Add a point $*$ (which will become the base point) and for each simplex $\sigma$ in base, add both $\sigma$ and a simplex made up of $*$ and $\sigma$ (topologically, form the join of $*$ and $\sigma$).

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, v)})
sage: CX = X.cone() # indirect doctest
sage: CX.nondegenerate_simplices()
[*, v, (v,*), e, (e,*)]
sage: CX.base_point()
*
```

**n_skeleton**(*n*)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- n – the dimension

In the case when the cone is infinite, the $n$-skeleton of the cone is computed as the $n$-skeleton of the cone of the $n$-skeleton.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: X = B.disjoint_union(B)
sage: CX = B.cone()
sage: CX.n_skeleton(3).homology()                                              #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: Z}
```

**class** sage.topology.simplicial_set_constructions.**ConeOfSimplicialSet_finite**(*base*)

Bases: *ConeOfSimplicialSet*, *SimplicialSet_finite*

Return the unreduced cone on a finite simplicial set.

INPUT:

- base – return the cone on this simplicial set

Add a point $*$ (which will become the base point) and for each simplex $\sigma$ in base, add both $\sigma$ and a simplex made up of $*$ and $\sigma$ (topologically, form the join of $*$ and $\sigma$).

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, v)})
sage: CX = X.cone() # indirect doctest
sage: CX.nondegenerate_simplices()
[*, v, (v,*), e, (e,*)]
sage: CX.base_point()
*
```

**base_as_subset**()

If this is the cone $CX$ on $X$, return $X$ as a subsimplicial set.

EXAMPLES:

```
sage: # needs sage.groups
sage: X = simplicial_sets.RealProjectiveSpace(4).unset_base_point()
sage: Y = X.cone()
sage: Y.base_as_subset()
Simplicial set with 5 non-degenerate simplices
sage: Y.base_as_subset() == X
True
```

**map_from_base**()

If this is the cone $CX$ on $X$, return the inclusion map from $X$.

EXAMPLES:

```
sage: X = simplicial_sets.Simplex(2).n_skeleton(1)
sage: Y = X.cone()
sage: Y.map_from_base()
Simplicial set morphism:
  From: Simplicial set with 6 non-degenerate simplices
  To:   Cone of Simplicial set with 6 non-degenerate simplices
  Defn: [(0,), (1,), (2,), (0, 1), (0, 2), (1, 2)]
        --> [(0,), (1,), (2,), (0, 1), (0, 2), (1, 2)]
```

**class** sage.topology.simplicial_set_constructions.**DisjointUnionOfSimplicialSets**(*factors=None*)

Bases: *PushoutOfSimplicialSets*, *Factors*

Return the disjoint union of simplicial sets.

INPUT:

- `factors` – list or tuple of simplicial sets

Discard any factors which are empty and return the disjoint union of the remaining simplicial sets in `factors`. The disjoint union comes equipped with a map from each factor, as long as all of the factors are finite.

EXAMPLES:

```
sage: CP2 = simplicial_sets.ComplexProjectiveSpace(2)
sage: K = simplicial_sets.KleinBottle()
sage: W = CP2.disjoint_union(K)
sage: W.homology()                                                          #␣
↪needs sage.modules
{0: Z, 1: Z x C2, 2: Z, 3: 0, 4: Z}
```

(continues on next page)

(continued from previous page)

```
sage: W.inclusion_map(1)
Simplicial set morphism:
  From: Klein bottle
  To:   Disjoint union: (CP^2 u Klein bottle)
  Defn: [Delta_{0,0}, Delta_{1,0}, Delta_{1,1}, Delta_{1,2}, Delta_{2,0}, Delta_
↪{2,1}]
        --> [Delta_{0,0}, Delta_{1,0}, Delta_{1,1}, Delta_{1,2}, Delta_{2,0},␣
↪Delta_{2,1}]
```

**n_skeleton**($n$)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- `n` – the dimension

The $n$-skeleton of the disjoint union is computed as the disjoint union of the $n$-skeleta of the component simplicial sets.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: T = simplicial_sets.Torus()
sage: X = B.disjoint_union(T)
sage: X.n_skeleton(3).homology()                                              #␣
↪needs sage.modules
{0: Z, 1: Z x Z x C2, 2: Z, 3: Z}
```

**summand**($i$)

Return the $i$-th factor of this construction of simplicial sets.

INPUT:

- `i` – integer; the index of the factor

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: K = S2.disjoint_union(S3)
sage: K.factor(0)
S^2

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: X = B.wedge(S3, B)
sage: X.factor(1)
S^3
sage: X.factor(2)
Classifying space of Multiplicative Abelian group isomorphic to C2
```

**summands**()

Return the factors involved in this construction of simplicial sets.

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: S2.wedge(S3).factors() == (S2, S3)
True
sage: S2.product(S3).factors()[0]
S^2
```

**class** sage.topology.simplicial_set_constructions.**DisjointUnionOfSimplicialSets_finite**(*factors=N*

Bases: *DisjointUnionOfSimplicialSets*, *PushoutOfSimplicialSets_finite*

The disjoint union of finite simplicial sets.

**inclusion_map**(*i*)

Return the inclusion map of the $i$-th factor.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: S2 = simplicial_sets.Sphere(2)
sage: W = S1.disjoint_union(S2, S1)
sage: W.inclusion_map(1)
Simplicial set morphism:
  From: S^2
  To:   Disjoint union: (S^1 u S^2 u S^1)
  Defn: [v_0, sigma_2] --> [v_0, sigma_2]
sage: W.inclusion_map(0).domain()
S^1
sage: W.inclusion_map(2).domain()
S^1
```

**class** sage.topology.simplicial_set_constructions.**Factors**

Bases: object

Classes which inherit from this should define a _factors attribute for their instances, and this class accesses that attribute. This is used by *ProductOfSimplicialSets*, *WedgeOfSimplicialSets*, and *DisjointUnionOfSimplicialSets*.

**factor**(*i*)

Return the $i$-th factor of this construction of simplicial sets.

INPUT:

- i – integer; the index of the factor

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: K = S2.disjoint_union(S3)
sage: K.factor(0)
S^2

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: X = B.wedge(S3, B)
```

(continues on next page)

(continued from previous page)

```
sage: X.factor(1)
S^3
sage: X.factor(2)
Classifying space of Multiplicative Abelian group isomorphic to C2
```

**factors**()

Return the factors involved in this construction of simplicial sets.

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: S2.wedge(S3).factors() == (S2, S3)
True
sage: S2.product(S3).factors()[0]
S^2
```

**class** sage.topology.simplicial_set_constructions.**ProductOfSimplicialSets**(*factors=None*)

Bases: *PullbackOfSimplicialSets*, *Factors*

Return the product of simplicial sets.

INPUT:

- `factors` – list or tuple of simplicial sets

Return the product of the simplicial sets in `factors`.

If $X$ and $Y$ are simplicial sets, then their product $X \times Y$ is defined to be the simplicial set with $n$-simplices $X_n \times Y_n$. Therefore the simplices in the product have the form $(s_I\sigma, s_J\tau)$, where $s_I = s_{i_1}...s_{i_p}$ and $s_J = s_{j_1}...s_{j_q}$ are composites of degeneracy maps, written in decreasing order. Such a simplex is nondegenerate if the indices $I$ and $J$ are disjoint. Therefore if $\sigma$ and $\tau$ are nondegenerate simplices of dimensions $m$ and $n$, in the product they will lead to nondegenerate simplices up to dimension $m + n$, and no further.

This extends in the more or less obvious way to products with more than two factors: with three factors, a simplex $(s_I\sigma, s_J\tau, s_K\rho)$ is nondegenerate if $I \cap J \cap K$ is empty, etc.

If a simplicial set is constructed as a product, the factors are recorded and are accessible via the method *Factors.factors()*. If it is constructed as a product and then copied, this information is lost.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, w)})
sage: square = X.product(X)
```

`square` is now the standard triangulation of the square: 4 vertices, 5 edges (the four on the border plus the diagonal), 2 triangles:

```
sage: square.f_vector()
[4, 5, 2]

sage: S1 = simplicial_sets.Sphere(1)
sage: T = S1.product(S1)
```

(continues on next page)

(continued from previous page)

```
sage: T.homology(reduced=False)                                                 #␣
↪needs sage.modules
{0: Z, 1: Z x Z, 2: Z}
```

Since `S1` is pointed, so is `T`:

```
sage: S1.is_pointed()
True
sage: S1.base_point()
v_0
sage: T.is_pointed()
True
sage: T.base_point()
(v_0, v_0)

sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: Z = S2.product(S3)
sage: Z.homology()                                                              #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z, 3: Z, 4: 0, 5: Z}
```

Products involving infinite simplicial sets:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: B.rename('RP^oo')
sage: X = B.product(B); X
RP^oo x RP^oo
sage: X.n_cells(1)
[(f, f), (f, s_0 1), (s_0 1, f)]
sage: X.homology(range(3), base_ring=GF(2))                                     #␣
↪needs sage.modules
{0: Vector space of dimension 0 over Finite Field of size 2,
 1: Vector space of dimension 2 over Finite Field of size 2,
 2: Vector space of dimension 3 over Finite Field of size 2}
sage: Y = B.product(S2)
sage: Y.homology(range(5), base_ring=GF(2))                                     #␣
↪needs sage.modules
{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 2 over Finite Field of size 2,
 3: Vector space of dimension 2 over Finite Field of size 2,
 4: Vector space of dimension 2 over Finite Field of size 2}
```

**factor** (*i*, *as_subset=False*)

Return the $i$-th factor of the product.

INPUT:

- `i` – integer; the index of the factor
- `as_subset` – boolean (default: `False`)

If `as_subset` is `True`, return the $i$-th factor as a subsimplicial set of the product, identifying it with its product with the base point in each other factor. As a subsimplicial set, it comes equipped with an inclusion map. This option will raise an error if any factor does not have a base point.

If `as_subset` is `False`, return the $i$-th factor in its original form as a simplicial set.

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: K = S2.product(S3)
sage: K.factor(0)
S^2

sage: K.factor(0, as_subset=True)
Simplicial set with 2 non-degenerate simplices
sage: K.factor(0, as_subset=True).homology()                                #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z}

sage: K.factor(0) is S2
True
sage: K.factor(0, as_subset=True) is S2
False
```

**n_skeleton**(*n*)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- `n` – the dimension

In the finite case, this returns the ordinary $n$-skeleton. In the infinite case, it computes the $n$-skeleton of the product of the $n$-skeleta of the factors.

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: S2.product(S3).n_skeleton(2)
Simplicial set with 2 non-degenerate simplices

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: X = B.product(B)
sage: X.n_skeleton(2)
Simplicial set with 13 non-degenerate simplices
```

**class** sage.topology.simplicial_set_constructions.**ProductOfSimplicialSets_finite**(*factors=None*)

Bases: *ProductOfSimplicialSets*, *PullbackOfSimplicialSets_finite*

The product of finite simplicial sets.

When the factors are all finite, there are more methods available for the resulting product, as compared to products with infinite factors: projection maps, the wedge as a subcomplex, and the fat wedge as a subcomplex. See *projection_map()*, *wedge_as_subset()*, and *fat_wedge_as_subset()*

**fat_wedge_as_subset**()

Return the fat wedge as a subsimplicial set of this product of pointed simplicial sets.

The fat wedge consists of those terms where at least one factor is the base point. Thus with two factors this is the ordinary wedge, but with more factors, it is larger.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: X = S1.product(S1, S1)
sage: W = X.fat_wedge_as_subset()
sage: W.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: Z x Z x Z, 2: Z x Z x Z}
```

**projection_map**(*i*)

Return the map projecting onto the $i$-th factor.

INPUT:

- `i` – integer; the index of the projection map

EXAMPLES:

```
sage: T = simplicial_sets.Torus()
sage: f_0 = T.projection_map(0)
sage: f_1 = T.projection_map(1)

sage: # needs sage.modules
sage: m_0 = f_0.induced_homology_morphism().to_matrix(1)  # matrix in dim 1
sage: m_1 = f_1.induced_homology_morphism().to_matrix(1)
sage: m_0.rank()
1
sage: m_0 == m_1
False
```

**wedge_as_subset**()

Return the wedge as a subsimplicial set of this product of pointed simplicial sets.

This will raise an error if any factor is not pointed.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: e = AbstractSimplex(1, name='e')
sage: w = AbstractSimplex(0, name='w')
sage: f = AbstractSimplex(1, name='f')
sage: X = SimplicialSet({e: (v, v)}, base_point=v)
sage: Y = SimplicialSet({f: (w, w)}, base_point=w)
sage: P = X.product(Y)
sage: W = P.wedge_as_subset()
sage: W.nondegenerate_simplices()
[(v, w), (e, s_0 w), (s_0 v, f)]
sage: W.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: Z x Z}
```

**class** sage.topology.simplicial_set_constructions.**PullbackOfSimplicialSets**(*maps=None*)

Bases: *SimplicialSet_arbitrary*, UniqueRepresentation

Return the pullback obtained from the morphisms `maps`.

INPUT:

- maps – list or tuple of morphisms of simplicial sets

If only a single map $f : X \to Y$ is given, then return $X$. If no maps are given, return the one-point simplicial set. Otherwise, given a simplicial set $Y$ and maps $f_i : X_i \to Y$ for $0 \leq i \leq m$, construct the pullback $P$: see Wikipedia article Pullback_(category_theory). This is constructed as pullbacks of sets for each set of $n$-simplices, so $P_n$ is the subset of the product $\prod (X_i)_n$ consisting of those elements $(x_i)$ for which $f_i(x_i) = f_j(x_j)$ for all $i$, $j$.

This is pointed if the maps $f_i$ are.

EXAMPLES:

The pullback of a quotient map by a subsimplicial set and the base point map gives a simplicial set isomorphic to the original subcomplex:

```
sage: # needs sage.groups
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: K = RP5.quotient(RP5.n_skeleton(2))
sage: X = K.pullback(K.quotient_map(), K.base_point_map())
sage: X.homology() == RP5.n_skeleton(2).homology()                              #␣
↪needs sage.modules
True
```

Pullbacks of identity maps:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: one = S2.Hom(S2).identity()
sage: P = S2.pullback(one, one)
sage: P.homology()                                                              #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z}
```

The pullback is constructed in terms of the product – of course, the product is a special case of the pullback – and the simplices are named appropriately:

```
sage: P.nondegenerate_simplices()
[(v_0, v_0), (sigma_2, sigma_2)]
```

**defining_map**($i$)

Return the $i$-th map defining the pullback.

INPUT:

- i – integer

If this pullback was constructed as `Y.pullback(f_0, f_1, ...)`, this returns $f_i$.

EXAMPLES:

```
sage: # needs sage.groups
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: K = RP5.quotient(RP5.n_skeleton(2))
sage: Y = K.pullback(K.quotient_map(), K.base_point_map())
sage: Y.defining_map(1)
Simplicial set morphism:
  From: Point
  To:   Quotient: (RP^5/Simplicial set with 3 non-degenerate simplices)
  Defn: Constant map at *
sage: Y.defining_map(0).domain()
RP^5
```

**n_skeleton**(*n*)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- `n` – the dimension

The $n$-skeleton of the pullback is computed as the pullback of the $n$-skeleta of the component simplicial sets.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: one = Hom(B,B).identity()
sage: c = Hom(B,B).constant_map()
sage: P = B.pullback(one, c)
sage: P.n_skeleton(2)
Pullback of maps:
  Simplicial set endomorphism of Simplicial set with 3 non-degenerate␣
↪simplices
    Defn: Identity map
  Simplicial set endomorphism of Simplicial set with 3 non-degenerate␣
↪simplices
    Defn: Constant map at 1
sage: P.n_skeleton(3).homology()                                          #␣
↪needs sage.modules
{0: 0, 1: C2, 2: 0, 3: Z}
```

**class** sage.topology.simplicial_set_constructions.**PullbackOfSimplicialSets_finite**(*maps=None*)

Bases: *PullbackOfSimplicialSets*, *SimplicialSet_finite*

The pullback of finite simplicial sets obtained from `maps`.

When the simplicial sets involved are all finite, there are more methods available to the resulting pullback, as compared to case when some of the components are infinite: the structure maps from the pullback and the pullback's universal property: see *structure_map()* and *universal_property()*.

**projection_map**(*i*)

Return the $i$-th projection map of the pullback.

INPUT:

- `i` – integer

If this pullback $P$ was constructed as `Y.pullback(f_0, f_1, ...)`, where $f_i : X_i \to Y$, then there are structure maps $\bar{f}_i : P \to X_i$. This method constructs $\bar{f}_i$.

EXAMPLES:

```
sage: # needs sage.groups
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: K = RP5.quotient(RP5.n_skeleton(2))
sage: Y = K.pullback(K.quotient_map(), K.base_point_map())
sage: Y.structure_map(0)
Simplicial set morphism:
  From: Pullback of maps:
  Simplicial set morphism:
    From: RP^5
```

(continues on next page)

(continued from previous page)

```
    To:   Quotient: (RP^5/Simplicial set with 3 non-degenerate simplices)
    Defn: [1, f, f * f, f * f * f, f * f * f * f, f * f * f * f * f]
          --> [*, s_0 *, s_1 s_0 *, f * f * f, f * f * f * f, f * f * f * f *␣
↪f]
  Simplicial set morphism:
    From: Point
    To:   Quotient: (RP^5/Simplicial set with 3 non-degenerate simplices)
    Defn: Constant map at *
  To:   RP^5
  Defn: [(1, *), (f, s_0 *), (f * f, s_1 s_0 *)] --> [1, f, f * f]
sage: Y.structure_map(1).codomain()
Point
```

These maps are also accessible via *projection_map()*:

```
sage: Y.projection_map(1).codomain()                                    #␣
↪needs sage.groups
Point
```

**structure_map**(*i*)

Return the $i$-th projection map of the pullback.

INPUT:

- `i` – integer

If this pullback $P$ was constructed as `Y.pullback(f_0, f_1, ...)`, where $f_i : X_i \to Y$, then there are structure maps $\bar{f}_i : P \to X_i$. This method constructs $\bar{f}_i$.

EXAMPLES:

```
sage: # needs sage.groups
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: K = RP5.quotient(RP5.n_skeleton(2))
sage: Y = K.pullback(K.quotient_map(), K.base_point_map())
sage: Y.structure_map(0)
Simplicial set morphism:
  From: Pullback of maps:
  Simplicial set morphism:
    From: RP^5
    To:   Quotient: (RP^5/Simplicial set with 3 non-degenerate simplices)
    Defn: [1, f, f * f, f * f * f, f * f * f * f, f * f * f * f * f]
          --> [*, s_0 *, s_1 s_0 *, f * f * f, f * f * f * f, f * f * f * f *␣
↪f]
  Simplicial set morphism:
    From: Point
    To:   Quotient: (RP^5/Simplicial set with 3 non-degenerate simplices)
    Defn: Constant map at *
  To:   RP^5
  Defn: [(1, *), (f, s_0 *), (f * f, s_1 s_0 *)] --> [1, f, f * f]
sage: Y.structure_map(1).codomain()
Point
```

These maps are also accessible via *projection_map()*:

```
sage: Y.projection_map(1).codomain()                                    #␣
↪needs sage.groups
Point
```

**universal_property**(**maps*)

Return the map induced by `maps`.

INPUT:

- `maps` – maps from a simplicial set $Z$ to the "factors" $X_i$ forming the pullback

If the pullback $P$ is formed by maps $f_i : X_i \to Y$, then given maps $g_i : Z \to X_i$ such that $f_i g_i = f_j g_j$ for all $i$, $j$, then there is a unique map $g : Z \to P$ making the appropriate diagram commute. This constructs that map.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: T = S1.product(S1)
sage: K = T.factor(0, as_subset=True)
sage: f = S1.Hom(T)({S1.n_cells(0)[0]: K.n_cells(0)[0],
....:                S1.n_cells(1)[0]: K.n_cells(1)[0]})
sage: P = S1.product(T)
sage: P.universal_property(S1.Hom(S1).identity(), f)
Simplicial set morphism:
  From: S^1
  To:   S^1 x S^1 x S^1
  Defn: [v_0, sigma_1] --> [(v_0, (v_0, v_0)), (sigma_1, (sigma_1, s_0 v_0))]
```

**class** sage.topology.simplicial_set_constructions.**PushoutOfSimplicialSets**(*maps=None*, *vertex_name=None*)

Bases: *SimplicialSet_arbitrary*, UniqueRepresentation

Return the pushout obtained from the morphisms `maps`.

INPUT:

- `maps` – list or tuple of morphisms of simplicial sets
- `vertex_name` – (default: `None`)

If only a single map $f : X \to Y$ is given, then return $Y$. If no maps are given, return the empty simplicial set. Otherwise, given a simplicial set $X$ and maps $f_i : X \to Y_i$ for $0 \leq i \leq m$, construct the pushout $P$: see Wikipedia article Pushout_(category_theory). This is constructed as pushouts of sets for each set of $n$-simplices, so $P_n$ is the disjoint union of the sets $(Y_i)_n$, with elements $f_i(x)$ identified for $n$-simplex $x$ in $X$.

Simplices in the pushout are given names as follows: if a simplex comes from a single $Y_i$, it inherits its name. Otherwise it must come from a simplex (or several) in $X$, and then it inherits one of those names, and it should be the first alphabetically. For example, if vertices $v$, $w$, and $z$ in $X$ are glued together, then the resulting vertex in the pushout will be called $v$.

Base points are taken care of automatically: if each of the maps $f_i$ is pointed, so is the pushout. If $X$ is a point or if $X$ is nonempty and any of the spaces $Y_i$ is a point, use those for the base point. In all of these cases, if `vertex_name` is `None`, generate the name of the base point automatically; otherwise, use `vertex_name` for its name.

In all other cases, the pushout is not pointed.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: a = AbstractSimplex(0, name='a')
sage: b = AbstractSimplex(0, name='b')
```

(continues on next page)

(continued from previous page)

```
sage: c = AbstractSimplex(0, name='c')
sage: e0 = AbstractSimplex(1, name='e_0')
sage: e1 = AbstractSimplex(1, name='e_1')
sage: e2 = AbstractSimplex(1, name='e_2')
sage: X = SimplicialSet({e2: (b, a)})
sage: Y0 = SimplicialSet({e2: (b,a), e0: (c,b), e1: (c,a)})
sage: Y1 = simplicial_sets.Simplex(0)
sage: f0_data = {a:a, b:b, e2: e2}
sage: v = Y1.n_cells(0)[0]
sage: f1_data = {a:v, b:v, e2:v.apply_degeneracies(0)}
sage: f0 = X.Hom(Y0)(f0_data)
sage: f1 = X.Hom(Y1)(f1_data)
sage: P = X.pushout(f0, f1)
sage: P.nondegenerate_simplices()
[a, c, e_0, e_1]
```

There are defining maps $f_i : X \to Y_i$ and structure maps $\bar{f}_i : Y_i \to P$; the latter are only implemented in Sage when each $Y_i$ is finite.

```
sage: P.defining_map(0) == f0
True
sage: P.structure_map(1)
Simplicial set morphism:
  From: 0-simplex
  To:   Pushout of maps:
  Simplicial set morphism:
    From: Simplicial set with 3 non-degenerate simplices
    To:   Simplicial set with 6 non-degenerate simplices
    Defn: [a, b, e_2] --> [a, b, e_2]
  Simplicial set morphism:
    From: Simplicial set with 3 non-degenerate simplices
    To:   0-simplex
    Defn: Constant map at (0,)
  Defn: Constant map at a
sage: P.structure_map(0).domain() == Y0
True
sage: P.structure_map(0).codomain() == P
True
```

An inefficient way of constructing a suspension for an unpointed set: take the pushout of two copies of the inclusion map $X \to CX$:

```
sage: T = simplicial_sets.Torus()
sage: T = T.unset_base_point()
sage: CT = T.cone()
sage: inc = CT.base_as_subset().inclusion_map()
sage: P = T.pushout(inc, inc)
sage: P.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z x Z, 3: Z}
sage: len(P.nondegenerate_simplices())
20
```

It is more efficient to construct the suspension as the quotient $CX/X$:

```
sage: len(CT.quotient(CT.base_as_subset()).nondegenerate_simplices())
8
```

It is more efficient still if the original simplicial set has a base point:

```
sage: T = simplicial_sets.Torus()
sage: len(T.suspension().nondegenerate_simplices())
6

sage: S1 = simplicial_sets.Sphere(1)
sage: pt = simplicial_sets.Point()
sage: bouquet = pt.pushout(S1.base_point_map(),
....:                      S1.base_point_map(),
....:                      S1.base_point_map())
sage: bouquet.homology(1)                                                   #␣
↪needs sage.modules
Z x Z x Z
```

**defining_map**($i$)

Return the $i$-th map defining the pushout.

INPUT:

- `i` – integer

If this pushout was constructed as `X.pushout(f_0, f_1, ...)`, this returns $f_i$.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: T = simplicial_sets.Torus()
sage: X = S1.wedge(T)  # a pushout
sage: X.defining_map(0)
Simplicial set morphism:
  From: Point
  To:   S^1
  Defn: Constant map at v_0
sage: X.defining_map(1).domain()
Point
sage: X.defining_map(1).codomain()
Torus
```

**n_skeleton**($n$)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- `n` – the dimension

The $n$-skeleton of the pushout is computed as the pushout of the $n$-skeleta of the component simplicial sets.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: K = B.n_skeleton(3)
sage: Q = K.pushout(K.inclusion_map(), K.constant_map())
sage: Q.n_skeleton(5).homology()                                            #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0, 4: Z, 5: Z}
```

Of course, computing the $n$-skeleton and then taking homology need not yield the same answer as asking for homology through dimension $n$, since the latter computation will use the $(n+1)$-skeleton:

```
sage: Q.homology(range(6))                                                  #␣
↪needs sage.groups sage.modules
{0: 0, 1: 0, 2: 0, 3: 0, 4: Z, 5: C2}
```

**class** sage.topology.simplicial_set_constructions.**PushoutOfSimplicialSets_finite**(*maps=None*, *vertex_name=None*

Bases: *PushoutOfSimplicialSets*, *SimplicialSet_finite*

The pushout of finite simplicial sets obtained from `maps`.

When the simplicial sets involved are all finite, there are more methods available to the resulting pushout, as compared to case when some of the components are infinite: the structure maps to the pushout and the pushout's universal property: see *structure_map()* and *universal_property()*.

**structure_map**(*i*)

Return the $i$-th structure map of the pushout.

INPUT:

- `i` – integer

If this pushout $Z$ was constructed as `X.pushout(f_0, f_1, ...)`, where $f_i : X \to Y_i$, then there are structure maps $\bar{f}_i : Y_i \to Z$. This method constructs $\bar{f}_i$.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: T = simplicial_sets.Torus()
sage: X = S1.disjoint_union(T)  # a pushout
sage: X.structure_map(0)
Simplicial set morphism:
  From: S^1
  To:   Disjoint union: (S^1 u Torus)
  Defn: [v_0, sigma_1] --> [v_0, sigma_1]
sage: X.structure_map(1).domain()
Torus
sage: X.structure_map(1).codomain()
Disjoint union: (S^1 u Torus)
```

**universal_property**(**maps*)

Return the map induced by `maps`.

INPUT:

- `maps` – maps "factors" $Y_i$ forming the pushout to a fixed simplicial set $Z$

If the pushout $P$ is formed by maps $f_i : X \to Y_i$, then given maps $g_i : Y_i \to Z$ such that $g_i f_i = g_j f_j$ for all $i, j$, then there is a unique map $g : P \to Z$ making the appropriate diagram commute. This constructs that map.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: x = AbstractSimplex(0, name='x')
```

(continues on next page)

(continued from previous page)

```
sage: evw = AbstractSimplex(1, name='vw')
sage: evx = AbstractSimplex(1, name='vx')
sage: ewx = AbstractSimplex(1, name='wx')
sage: X = SimplicialSet({evw: (w, v), evx: (x, v)})
sage: Y_0 = SimplicialSet({evw: (w, v), evx: (x, v), ewx: (x, w)})
sage: Y_1 = SimplicialSet({evx: (x, v)})

sage: f_0 = Hom(X, Y_0)({v:v, w:w, x:x, evw:evw, evx:evx})
sage: f_1 = Hom(X, Y_1)({v:v, w:v, x:x,
....:                    evw:v.apply_degeneracies(0), evx:evx})
sage: P = X.pushout(f_0, f_1)

sage: one = Hom(Y_1, Y_1).identity()
sage: g = Hom(Y_0, Y_1)({v:v, w:v, x:x,
....:                    evw:v.apply_degeneracies(0), evx:evx, ewx:evx})
sage: P.universal_property(g, one)
Simplicial set morphism:
  From: Pushout of maps:
  Simplicial set morphism:
    From: Simplicial set with 5 non-degenerate simplices
    To:   Simplicial set with 6 non-degenerate simplices
    Defn: [v, w, x, vw, vx] --> [v, w, x, vw, vx]
  Simplicial set morphism:
    From: Simplicial set with 5 non-degenerate simplices
    To:   Simplicial set with 3 non-degenerate simplices
    Defn: [v, w, x, vw, vx] --> [v, v, x, s_0 v, vx]
  To:   Simplicial set with 3 non-degenerate simplices
  Defn: [v, x, vx, wx] --> [v, x, vx, vx]
```

**class** sage.topology.simplicial_set_constructions.**QuotientOfSimplicialSet**(*inclusion*, *vertex_name='*'*)

Bases: *PushoutOfSimplicialSets*

Return the quotient of a simplicial set by a subsimplicial set.

INPUT:

- `inclusion` – inclusion map of a subcomplex (= subsimplicial set) of a simplicial set
- `vertex_name` – string (default: `'*'`)

A subcomplex $A$ comes equipped with the inclusion map $A \to X$ to its ambient complex $X$, and this constructs the quotient $X/A$, collapsing $A$ to a point. The resulting point is called `vertex_name`, which is `'*'` by default.

When the simplicial sets involved are finite, there is a *QuotientOfSimplicialSet_finite.quotient_map()* method available.

EXAMPLES:

```
sage: # needs sage.groups
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: RP2 = RP5.n_skeleton(2)
sage: RP5_2 = RP5.quotient(RP2); RP5_2
Quotient: (RP^5/Simplicial set with 3 non-degenerate simplices)
sage: RP5_2.quotient_map()
Simplicial set morphism:
```

(continues on next page)

(continued from previous page)

```
From: RP^5
To:   Quotient: (RP^5/Simplicial set with 3 non-degenerate simplices)
Defn: [1, f, f * f, f * f * f, f * f * f * f, f * f * f * f * f]
      --> [*, s_0 *, s_1 s_0 *, f * f * f, f * f * f * f, f * f * f * f * f]
```

**ambient**()

Return the ambient space.

That is, if this quotient is $K/L$, return $K$.

EXAMPLES:

```
sage: # needs sage.groups
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: RP2 = RP5.n_skeleton(2)
sage: RP5_2 = RP5.quotient(RP2)
sage: RP5_2.ambient()
RP^5

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: K = B.n_skeleton(3)
sage: Q = B.quotient(K)
sage: Q.ambient()
Classifying space of Multiplicative Abelian group isomorphic to C2
```

**n_skeleton**(*n*)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- n – the dimension

The $n$-skeleton of the quotient is computed as the quotient of the $n$-skeleta.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: K = B.n_skeleton(3)
sage: Q = B.quotient(K)
sage: Q.n_skeleton(6)
Quotient: (Simplicial set with 7
           non-degenerate simplices/Simplicial set with 4
                                    non-degenerate simplices)
sage: Q.n_skeleton(6).homology()                                              #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: 0, 4: Z, 5: C2, 6: 0}
```

**subcomplex**()

Return the subcomplex space associated to this quotient.

That is, if this quotient is $K/L$, return $L$.

EXAMPLES:

```
sage: # needs sage.groups
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: RP2 = RP5.n_skeleton(2)
sage: RP5_2 = RP5.quotient(RP2)
sage: RP5_2.subcomplex()
Simplicial set with 3 non-degenerate simplices

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: K = B.n_skeleton(3)
sage: Q = B.quotient(K)
sage: Q.subcomplex()
Simplicial set with 4 non-degenerate simplices
```

**class** sage.topology.simplicial_set_constructions.**QuotientOfSimplicialSet_finite**(*inclusion*, *vertex_name='*'*)

Bases: *QuotientOfSimplicialSet*, *PushoutOfSimplicialSets_finite*

The quotient of finite simplicial sets.

When the simplicial sets involved are finite, there is a *quotient_map()* method available.

**quotient_map**()

Return the quotient map from the original simplicial set to the quotient.

EXAMPLES:

```
sage: K = simplicial_sets.Simplex(1)
sage: S1 = K.quotient(K.n_skeleton(0))
sage: q = S1.quotient_map()
sage: q
Simplicial set morphism:
  From: 1-simplex
  To:   Quotient: (1-simplex/Simplicial set with 2 non-degenerate simplices)
  Defn: [(0,), (1,), (0, 1)] --> [*, *, (0, 1)]
sage: q.domain() == K
True
sage: q.codomain() == S1
True
```

**class** sage.topology.simplicial_set_constructions.**ReducedConeOfSimplicialSet**(*base*)

Bases: *QuotientOfSimplicialSet*

Return the reduced cone on a simplicial set.

INPUT:

- `base` – return the cone on this simplicial set

Start with the unreduced cone: take `base` and add a point $*$ (which will become the base point) and for each simplex $\sigma$ in `base`, add both $\sigma$ and a simplex made up of $*$ and $\sigma$ (topologically, form the join of $*$ and $\sigma$).

Now reduce: take the quotient by the 1-simplex connecting the old base point to the new one.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, v)})
sage: X = X.set_base_point(v)
sage: CX = X.cone()  # indirect doctest
sage: CX.nondegenerate_simplices()
[*, e, (e,*)]
```

**n_skeleton**(*n*)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- `n` – the dimension

In the case when the cone is infinite, the $n$-skeleton of the cone is computed as the $n$-skeleton of the cone of the $n$-skeleton.

EXAMPLES:

```
sage: G = groups.misc.MultiplicativeAbelian([2])                              #␣
↪needs sage.groups
sage: B = simplicial_sets.ClassifyingSpace(G)                                 #␣
↪needs sage.groups
sage: B.cone().n_skeleton(3).homology()                                       #␣
↪needs sage.groups sage.modules
{0: 0, 1: 0, 2: 0, 3: Z}
```

**class** sage.topology.simplicial_set_constructions.**ReducedConeOfSimplicialSet_finite**(*base*)

Bases: *ReducedConeOfSimplicialSet*, *QuotientOfSimplicialSet_finite*

Return the reduced cone on a simplicial set.

INPUT:

- `base` – return the cone on this simplicial set

Start with the unreduced cone: take `base` and add a point $*$ (which will become the base point) and for each simplex $\sigma$ in `base`, add both $\sigma$ and a simplex made up of $*$ and $\sigma$ (topologically, form the join of $*$ and $\sigma$).

Now reduce: take the quotient by the 1-simplex connecting the old base point to the new one.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: e = AbstractSimplex(1, name='e')
sage: X = SimplicialSet({e: (v, v)})
sage: X = X.set_base_point(v)
sage: CX = X.cone()  # indirect doctest
sage: CX.nondegenerate_simplices()
[*, e, (e,*)]
```

**map_from_base**()

If this is the cone $\tilde{C}X$ on $X$, return the map from $X$.

The map is defined to be the composite $X \to CX \to \tilde{C}X$. This is used by the *SuspensionOfSimplicialSet_finite* class to construct the reduced suspension: take the quotient of the reduced cone by the image of $X$ therein.

EXAMPLES:

```
sage: S3 = simplicial_sets.Sphere(3)
sage: CS3 = S3.cone()
sage: CS3.map_from_base()
Simplicial set morphism:
  From: S^3
  To:   Reduced cone of S^3
  Defn: [v_0, sigma_3] --> [*, sigma_3]
```

**class** sage.topology.simplicial_set_constructions.**SmashProductOfSimplicialSets_finite**(*factors=No*

Bases: *QuotientOfSimplicialSet_finite*, *Factors*

Return the smash product of finite pointed simplicial sets.

INPUT:

- `factors` – list or tuple of simplicial sets

Return the smash product of the simplicial sets in `factors`: the smash product $X \wedge Y$ is defined to be the quotient $(X \times Y)/(X \vee Y)$, where $X \vee Y$ is the wedge sum.

Each element of `factors` must be finite and pointed. (As of July 2016, constructing the wedge as a subcomplex of the product is only possible in Sage for finite simplicial sets.)

EXAMPLES:

```
sage: T = simplicial_sets.Torus()
sage: S2 = simplicial_sets.Sphere(2)
sage: T.smash_product(S2).homology() == T.suspension(2).homology()           #␣
↪needs sage.modules
True
```

**class** sage.topology.simplicial_set_constructions.**SubSimplicialSet**(*data*, *ambient=None*)

Bases: *SimplicialSet_finite*, UniqueRepresentation

Return a finite simplicial set as a subsimplicial set of another simplicial set.

This keeps track of the ambient simplicial set and the inclusion map from the subcomplex into it.

INPUT:

- `data` – the data defining the subset: a dictionary where the keys are simplices from the ambient simplicial set and the values are their faces.
- `ambient` – the ambient simplicial set. If omitted, use the same simplicial set as the subset and the ambient complex.

EXAMPLES:

```
sage: S3 = simplicial_sets.Sphere(3)
sage: K = simplicial_sets.KleinBottle()
sage: X = S3.disjoint_union(K)
sage: Y = X.structure_map(0).image()  # the S3 summand
sage: Y.inclusion_map()
```

(continues on next page)

(continued from previous page)

```
Simplicial set morphism:
  From: Simplicial set with 2 non-degenerate simplices
  To:   Disjoint union: (S^3 u Klein bottle)
  Defn: [v_0, sigma_3] --> [v_0, sigma_3]
sage: Y.ambient_space()
Disjoint union: (S^3 u Klein bottle)
```

**ambient_space**()

Return the simplicial set of which this is a subsimplicial set.

EXAMPLES:

```
sage: T = simplicial_sets.Torus()
sage: eight = T.wedge_as_subset()
sage: eight
Simplicial set with 3 non-degenerate simplices
sage: eight.fundamental_group()                                             #␣
↪needs sage.groups
Finitely presented group < e0, e1 |  >
sage: eight.ambient_space()
Torus
```

**inclusion_map**()

Return the inclusion map from this subsimplicial set into its ambient space.

EXAMPLES:

```
sage: RP6 = simplicial_sets.RealProjectiveSpace(6)                          #␣
↪needs sage.groups
sage: K = RP6.n_skeleton(2)                                                 #␣
↪needs sage.groups
sage: K.inclusion_map()                                                     #␣
↪needs sage.groups
Simplicial set morphism:
  From: Simplicial set with 3 non-degenerate simplices
  To:   RP^6
  Defn: [1, f, f * f] --> [1, f, f * f]
```

$RP^6$ itself is constructed as a subsimplicial set of $RP^\infty$:

```
sage: latex(RP6.inclusion_map())                                            #␣
↪needs sage.groups
RP^{6} \to RP^{\infty}
```

**class** sage.topology.simplicial_set_constructions.**SuspensionOfSimplicialSet**(*base*)

Bases: *SimplicialSet_arbitrary*, UniqueRepresentation

Return the (reduced) suspension of a simplicial set.

INPUT:

- `base` – return the suspension of this simplicial set

If this simplicial set `X=base` is not pointed, or if it is itself an unreduced suspension, return the unreduced suspension: the quotient $CX/X$, where $CX$ is the (ordinary, unreduced) cone on $X$. If $X$ is pointed, then use the reduced cone instead, and so return the reduced suspension.

We use $S$ to denote unreduced suspension, $\Sigma$ for reduced suspension.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: B.suspension()
Sigma(Classifying space of Multiplicative Abelian group isomorphic to C2)
sage: B.suspension().n_skeleton(3).homology()                                  #␣
↪needs sage.modules
{0: 0, 1: 0, 2: C2, 3: 0}
```

If `X` is finite, the suspension comes with a quotient map from the cone:

```
sage: S3 = simplicial_sets.Sphere(3)
sage: S4 = S3.suspension()
sage: S4.quotient_map()
Simplicial set morphism:
  From: Reduced cone of S^3
  To:   Sigma(S^3)
  Defn: [*, sigma_3, (sigma_3,*)] --> [*, s_2 s_1 s_0 *, (sigma_3,*)]
```

**n_skeleton**(*n*)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- `n` – the dimension

In the case when the suspension is infinite, the $n$-skeleton of the suspension is computed as the $n$-skeleton of the suspension of the $n$-skeleton.

EXAMPLES:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: SigmaB = B.suspension()
sage: SigmaB.n_skeleton(4).homology(base_ring=GF(2))                           #␣
↪needs sage.modules
{0: Vector space of dimension 0 over Finite Field of size 2,
 1: Vector space of dimension 0 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2,
 3: Vector space of dimension 1 over Finite Field of size 2,
 4: Vector space of dimension 1 over Finite Field of size 2}
```

**class** sage.topology.simplicial_set_constructions.**SuspensionOfSimplicialSet_finite**(*base*)

Bases: *SuspensionOfSimplicialSet*, *QuotientOfSimplicialSet_finite*

The (reduced) suspension of a finite simplicial set.

See *SuspensionOfSimplicialSet* for more information.

**class** sage.topology.simplicial_set_constructions.**WedgeOfSimplicialSets**(*factors=None*)

Bases: *PushoutOfSimplicialSets*, *Factors*

Return the wedge sum of pointed simplicial sets.

INPUT:

- `factors` – list or tuple of simplicial sets

Return the wedge of the simplicial sets in `factors`: the wedge sum $X \vee Y$ is formed by taking the disjoint union of $X$ and $Y$ and identifying their base points. In this construction, the new base point is renamed '*'.

The wedge comes equipped with maps to and from each factor, or actually, maps from each factor, and maps to simplicial sets isomorphic to each factor. The codomains of the latter maps are quotients of the wedge, not identical to the original factors.

EXAMPLES:

```
sage: CP2 = simplicial_sets.ComplexProjectiveSpace(2)
sage: K = simplicial_sets.KleinBottle()
sage: W = CP2.wedge(K)
sage: W.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: Z x C2, 2: Z, 3: 0, 4: Z}

sage: W.inclusion_map(1)
Simplicial set morphism:
  From: Klein bottle
  To:   Wedge: (CP^2 v Klein bottle)
  Defn: [Delta_{0,0}, Delta_{1,0}, Delta_{1,1}, Delta_{1,2}, Delta_{2,0}, Delta_
↪{2,1}]
        --> [*, Delta_{1,0}, Delta_{1,1}, Delta_{1,2}, Delta_{2,0}, Delta_{2,1}]

sage: W.projection_map(0).domain()
Wedge: (CP^2 v Klein bottle)
sage: W.projection_map(0).codomain() # copy of CP^2
Quotient: (Wedge: (CP^2 v Klein bottle)/Simplicial set with 6 non-degenerate␣
↪simplices)
sage: W.projection_map(0).codomain().homology()                             #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z, 3: 0, 4: Z}
```

An error occurs if any of the factors is not pointed:

```
sage: CP2.wedge(simplicial_sets.Simplex(1))
Traceback (most recent call last):
...
ValueError: the simplicial sets must be pointed
```

**summand**($i$)

Return the $i$-th factor of this construction of simplicial sets.

INPUT:

- `i` – integer; the index of the factor

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: K = S2.disjoint_union(S3)
sage: K.factor(0)
S^2

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
```

(continues on next page)

(continued from previous page)

```
sage: X = B.wedge(S3, B)
sage: X.factor(1)
S^3
sage: X.factor(2)
Classifying space of Multiplicative Abelian group isomorphic to C2
```

**summands**()

Return the factors involved in this construction of simplicial sets.

EXAMPLES:

```
sage: S2 = simplicial_sets.Sphere(2)
sage: S3 = simplicial_sets.Sphere(3)
sage: S2.wedge(S3).factors() == (S2, S3)
True
sage: S2.product(S3).factors()[0]
S^2
```

**class** sage.topology.simplicial_set_constructions.**WedgeOfSimplicialSets_finite**(*factors=None*)

Bases: *WedgeOfSimplicialSets*, *PushoutOfSimplicialSets_finite*

The wedge sum of finite pointed simplicial sets.

**inclusion_map**($i$)

Return the inclusion map of the $i$-th factor.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: S2 = simplicial_sets.Sphere(2)
sage: W = S1.wedge(S2, S1)
sage: W.inclusion_map(1)
Simplicial set morphism:
  From: S^2
  To:   Wedge: (S^1 v S^2 v S^1)
  Defn: [v_0, sigma_2] --> [*, sigma_2]
sage: W.inclusion_map(0).domain()
S^1
sage: W.inclusion_map(2).domain()
S^1
```

**projection_map**($i$)

Return the projection map onto the $i$-th factor.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: S2 = simplicial_sets.Sphere(2)
sage: W = S1.wedge(S2, S1)
sage: W.projection_map(1)
Simplicial set morphism:
  From: Wedge: (S^1 v S^2 v S^1)
  To:   Quotient: (Wedge: (S^1 v S^2 v S^1)/Simplicial set with
                                              3 non-degenerate simplices)
  Defn: [*, sigma_1, sigma_1, sigma_2] --> [*, s_0 *, s_0 *, sigma_2]
sage: W.projection_map(1).image().homology(1)                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
0
sage: W.projection_map(1).image().homology(2)                                  #␣
↪needs sage.modules
Z
```

CHAPTER

NINE

# EXAMPLES OF SIMPLICIAL SETS.

These are accessible via `simplicial_sets.Sphere(3)`, `simplicial_sets.Torus()`, etc. Type `simplicial_sets.[TAB]` to see a complete list.

AUTHORS:

- John H. Palmieri (2016-07)
- Miguel Marco (2022-12)

`sage.topology.simplicial_set_examples.`**`ClassifyingSpace`**(*group*)

Return the classifying space of `group`, as a simplicial set.

INPUT:

- `group` – a finite group or finite monoid

See `sage.categories.finite_monoids.FiniteMonoids.ParentMethods.nerve()` for more details and more examples.

EXAMPLES:

```
sage: # needs sage.groups
sage: C2 = groups.misc.MultiplicativeAbelian([2])
sage: BC2 = simplicial_sets.ClassifyingSpace(C2)
sage: H = BC2.homology(range(9), base_ring=GF(2))                              #␣
↪needs sage.modules
sage: [H[i].dimension() for i in range(9)]                                     #␣
↪needs sage.modules
[0, 1, 1, 1, 1, 1, 1, 1, 1]

sage: Klein4 = groups.misc.MultiplicativeAbelian([2, 2])                       #␣
↪needs sage.groups
sage: BK = simplicial_sets.ClassifyingSpace(Klein4); BK                        #␣
↪needs sage.groups
Classifying space of Multiplicative Abelian group isomorphic to C2 x C2
sage: BK.homology(range(5), base_ring=GF(2))    # long time (1 second)         #␣
↪needs sage.groups sage.modules
{0: Vector space of dimension 0 over Finite Field of size 2,
 1: Vector space of dimension 2 over Finite Field of size 2,
 2: Vector space of dimension 3 over Finite Field of size 2,
 3: Vector space of dimension 4 over Finite Field of size 2,
 4: Vector space of dimension 5 over Finite Field of size 2}
```

`sage.topology.simplicial_set_examples.`**`ComplexProjectiveSpace`**(*n*)

Return complex $n$-dimensional projective space, as a simplicial set.

This is only defined when $n$ is at most 4. It is constructed using the simplicial set decomposition provided by Kenzo, as described by Sergeraert [Ser2010]

EXAMPLES:

```
sage: simplicial_sets.ComplexProjectiveSpace(2).homology(reduced=False)          #␣
↪needs sage.modules
{0: Z, 1: 0, 2: Z, 3: 0, 4: Z}
sage: CP3 = simplicial_sets.ComplexProjectiveSpace(3); CP3                       #␣
↪needs pyparsing
CP^3
sage: latex(CP3)                                                                 #␣
↪needs pyparsing
CP^{3}
sage: CP3.f_vector()                                                             #␣
↪needs pyparsing
[1, 0, 3, 10, 25, 30, 15]

sage: # long time, needs pyparsing sage.modules
sage: K = CP3.suspension()                # long time (1 second)
sage: R = K.cohomology_ring(GF(2))
sage: R.gens()
(h^{0,0}, h^{3,0}, h^{5,0}, h^{7,0})
sage: x = R.gens()[1]
sage: x.Sq(2)
h^{5,0}

sage: simplicial_sets.ComplexProjectiveSpace(4).f_vector()                       #␣
↪needs pyparsing
[1, 0, 4, 22, 97, 255, 390, 315, 105]

sage: simplicial_sets.ComplexProjectiveSpace(5)
Traceback (most recent call last):
...
ValueError: complex projective spaces are only available in dimensions between 0␣
↪and 4
```

sage.topology.simplicial_set_examples.**Empty**()

Return the empty simplicial set.

This should return the same simplicial set each time it is called.

EXAMPLES:

```
sage: from sage.topology.simplicial_set_examples import Empty
sage: E = Empty()
sage: E
Empty simplicial set
sage: E.nondegenerate_simplices()
[]
sage: E is Empty()
True
```

sage.topology.simplicial_set_examples.**HopfMap**()

Return a simplicial model of the Hopf map $S^3 \to S^2$.

This is taken from Exemple II.1.19 in the thesis of Clemens Berger [Ber1991].

The Hopf map is a fibration $S^3 \to S^2$. If it is viewed as attaching a 4-cell to the 2-sphere, the resulting adjunction space is 2-dimensional complex projective space. The resulting model is a bit larger than the one obtained from

simplicial_sets.ComplexProjectiveSpace(2).

EXAMPLES:

```
sage: g = simplicial_sets.HopfMap()
sage: g.domain()
Simplicial set with 20 non-degenerate simplices
sage: g.codomain()
S^2
```

Using the Hopf map to attach a cell:

```
sage: X = g.mapping_cone()
sage: CP2 = simplicial_sets.ComplexProjectiveSpace(2)
sage: X.homology() == CP2.homology()                                            #␣
↪needs sage.modules
True

sage: X.f_vector()
[1, 0, 5, 9, 6]
sage: CP2.f_vector()
[1, 0, 2, 3, 3]
```

sage.topology.simplicial_set_examples.**Horn**($n$, $k$)

Return the horn $\Lambda^n_k$.

This is the subsimplicial set of the $n$-simplex obtained by removing its $k$-th face.

EXAMPLES:

```
sage: L = simplicial_sets.Horn(3, 0)
sage: L
(3, 0)-Horn
sage: L.n_cells(3)
[]
sage: L.n_cells(2)
[(0, 1, 2), (0, 1, 3), (0, 2, 3)]

sage: L20 = simplicial_sets.Horn(2, 0)
sage: latex(L20)
\Lambda^{2}_{0}
sage: L20.inclusion_map()
Simplicial set morphism:
  From: (2, 0)-Horn
  To:   2-simplex
  Defn: [(0,), (1,), (2,), (0, 1), (0, 2)] --> [(0,), (1,), (2,), (0, 1), (0, 2)]
```

sage.topology.simplicial_set_examples.**KleinBottle**()

Return the Klein bottle as a simplicial set.

This converts the $\Delta$-complex version to a simplicial set. It has one 0-simplex, three 1-simplices, and two 2-simplices.

EXAMPLES:

```
sage: K = simplicial_sets.KleinBottle()
sage: K.f_vector()
[1, 3, 2]
sage: K.homology(reduced=False)                                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
{0: Z, 1: Z x C2, 2: 0}
sage: K
Klein bottle
```

**class** sage.topology.simplicial_set_examples.**Nerve**(*monoid*)

Bases: *SimplicialSet_arbitrary*

The nerve of a multiplicative monoid.

INPUT:

- `monoid` – a multiplicative monoid

See `sage.categories.finite_monoids.FiniteMonoids.ParentMethods.nerve()` for full documentation.

EXAMPLES:

```
sage: M = FiniteMonoids().example()
sage: M
An example of a finite multiplicative monoid: the integers modulo 12
sage: X = M.nerve()
sage: list(M)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
sage: X.n_cells(0)
[1]
sage: X.n_cells(1)
[0, 10, 11, 2, 3, 4, 5, 6, 7, 8, 9]
```

**n_skeleton**(*n*)

Return the $n$-skeleton of this simplicial set.

That is, the simplicial set generated by all nondegenerate simplices of dimension at most $n$.

INPUT:

- `n` – the dimension

EXAMPLES:

```
sage: # needs sage.groups
sage: K4 = groups.misc.MultiplicativeAbelian([2,2])
sage: BK4 = simplicial_sets.ClassifyingSpace(K4)
sage: BK4.n_skeleton(3)
Simplicial set with 40 non-degenerate simplices
sage: BK4.n_cells(1) == BK4.n_skeleton(3).n_cells(1)
True
sage: BK4.n_cells(3) == BK4.n_skeleton(1).n_cells(3)
False
```

sage.topology.simplicial_set_examples.**Point**()

Return a single point called "*" as a simplicial set.

This should return the same simplicial set each time it is called.

EXAMPLES:

```
sage: P = simplicial_sets.Point()
sage: P.is_pointed()
True
sage: P.nondegenerate_simplices()
[*]

sage: Q = simplicial_sets.Point()
sage: P is Q
True
sage: P == Q
True
```

sage.topology.simplicial_set_examples.**PresentationComplex**($G$)

Return a simplicial set constructed from a group presentation. The result is a subdivision of the presentation complex.

The presentation complex has a single vertex and it has one edge for each generator. Then triangles (and eventually new edges to glue them) are added to realize the relations.

INPUT:

- G – a finitely presented group

EXAMPLES:

```
sage: # needs sage.groups
sage: G = SymmetricGroup(2).as_finitely_presented_group(); G
Finitely presented group < a | a^2 >
sage: S = simplicial_sets.PresentationComplex(G); S
Simplicial set with 5 non-degenerate simplices
sage: S.face_data()
{Delta^0: None,
 a: (Delta^0, Delta^0),
 a^-1: (Delta^0, Delta^0),
 Ta: (a, s_0 Delta^0, a^-1),
 a^2: (a, s_0 Delta^0, a)}
sage: S.fundamental_group()
Finitely presented group < e0 | e0^2 >
```

sage.topology.simplicial_set_examples.**RealProjectiveSpace**($n$)

Return real $n$-dimensional projective space, as a simplicial set.

This is constructed as the $n$-skeleton of the nerve of the group of order 2, and therefore has a single non-degenerate simplex in each dimension up to $n$.

EXAMPLES:

```
sage: # needs sage.groups
sage: simplicial_sets.RealProjectiveSpace(7)
RP^7
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)
sage: RP5.homology()
{0: 0, 1: C2, 2: 0, 3: C2, 4: 0, 5: Z}
sage: RP5
RP^5
sage: latex(RP5)
RP^{5}

sage: BC2 = simplicial_sets.RealProjectiveSpace(Infinity)                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
sage: latex(BC2)                                                    #␣
↪needs sage.groups
RP^{\infty}
```

sage.topology.simplicial_set_examples.**Simplex**(*n*)

Return the $n$-simplex as a simplicial set.

EXAMPLES:

```
sage: K = simplicial_sets.Simplex(2)
sage: K
2-simplex
sage: latex(K)
\Delta^{2}
sage: K.n_cells(0)
[(0,), (1,), (2,)]
sage: K.n_cells(1)
[(0, 1), (0, 2), (1, 2)]
sage: K.n_cells(2)
[(0, 1, 2)]
```

sage.topology.simplicial_set_examples.**Sphere**(*n*)

Return the $n$-sphere as a simplicial set.

It is constructed with two non-degenerate simplices: a vertex $v_0$ (which is the base point) and an $n$-simplex $\sigma_n$.

INPUT:

- `n` – integer

EXAMPLES:

```
sage: S0 = simplicial_sets.Sphere(0)
sage: S0
S^0
sage: S0.nondegenerate_simplices()
[v_0, w_0]
sage: S0.is_pointed()
True
sage: simplicial_sets.Sphere(4)
S^4
sage: latex(simplicial_sets.Sphere(4))
S^{4}
sage: simplicial_sets.Sphere(4).nondegenerate_simplices()
[v_0, sigma_4]
```

sage.topology.simplicial_set_examples.**Torus**()

Return the torus as a simplicial set.

This computes the product of the circle with itself, where the circle is represented using a single 0-simplex and a single 1-simplex. Thus it has one 0-simplex, three 1-simplices, and two 2-simplices.

EXAMPLES:

```
sage: T = simplicial_sets.Torus()
sage: T.f_vector()
[1, 3, 2]
```

(continues on next page)

(continued from previous page)

```
sage: T.homology(reduced=False)                                                 #␣
↪needs sage.modules
{0: Z, 1: Z x Z, 2: Z}
```

sage.topology.simplicial_set_examples.**simplicial_data_from_kenzo_output**(*filename*)

Return data to construct a simplicial set, given Kenzo output.

INPUT:

- `filename` – name of file containing the output from Kenzo's `show-structure()` function

OUTPUT: data to construct a simplicial set from the Kenzo output

Several files with Kenzo output are in the directory `SAGE_EXTCODE/kenzo/`.

EXAMPLES:

```
sage: from sage.topology.simplicial_set_examples import simplicial_data_from_
↪kenzo_output
sage: from sage.topology.simplicial_set import SimplicialSet
sage: from pathlib import Path
sage: sphere = Path(SAGE_ENV['SAGE_EXTCODE']) / 'kenzo' /'S4.txt'
sage: S4 = SimplicialSet(simplicial_data_from_kenzo_output(sphere))             #␣
↪needs pyparsing
sage: S4.homology(reduced=False)                                                #␣
↪needs pyparsing
{0: Z, 1: 0, 2: 0, 3: 0, 4: Z}
```

CHAPTER

TEN

# CATALOG OF SIMPLICIAL SETS

This provides pre-built simplicial sets:

- the $n$-sphere and $n$-dimensional real projective space, both (in theory) for any positive integer $n$. In practice, as $n$ increases, it takes longer to construct these simplicial sets.
- the $n$-simplex and the horns obtained from it. As $n$ increases, it takes *much* longer to construct these simplicial sets, because the number of nondegenerate simplices increases exponentially in $n$. For example, it is feasible to do `simplicial_sets.RealProjectiveSpace(100)` since it only has 101 nondegenerate simplices, but `simplicial_sets.Simplex(20)` is probably a bad idea.
- $n$-dimensional complex projective space for $n \leq 4$
- the classifying space of a finite multiplicative group or monoid
- the torus and the Klein bottle
- the point
- the Hopf map: this is a pre-built morphism, from which one can extract its domain, codomain, mapping cone, etc.
- the complex of a group presentation.

All of these examples are accessible by typing `simplicial_sets.NAME`, where `NAME` is the name of the example. Type `simplicial_sets.[TAB]` for a complete list.

EXAMPLES:

```
sage: RP10 = simplicial_sets.RealProjectiveSpace(8)                    #␣
↪needs sage.groups
sage: RP10.homology()                                                  #␣
↪needs sage.groups sage.modules
{0: 0, 1: C2, 2: 0, 3: C2, 4: 0, 5: C2, 6: 0, 7: C2, 8: 0}

sage: eta = simplicial_sets.HopfMap()
sage: S3 = eta.domain()
sage: S2 = eta.codomain()
sage: S3.wedge(S2).homology()                                          #␣
↪needs sage.modules
{0: 0, 1: 0, 2: Z, 3: Z}
```

CHAPTER

ELEVEN

# MORPHISMS AND HOMSETS FOR SIMPLICIAL SETS

**Note**

Morphisms with infinite domain are not implemented in general: only constant maps and identity maps are currently implemented.

AUTHORS:

- John H. Palmieri (2016-07)

This module implements morphisms and homsets of simplicial sets.

**class** sage.topology.simplicial_set_morphism.**SimplicialSetHomset**(*X*, *Y*, *category=None*, *base=None*, *check=True*)

Bases: Homset

A set of morphisms between simplicial sets.

Once a homset has been constructed in Sage, typically via `Hom(X,Y)` or `X.Hom(Y)`, one can use it to construct a morphism $f$ by specifying a dictionary, the keys of which are the nondegenerate simplices in the domain, and the value corresponding to $\sigma$ is the simplex $f(\sigma)$ in the codomain.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: f = AbstractSimplex(1, name='f')
sage: X = SimplicialSet({e: (v, w), f: (w, v)})
sage: Y = SimplicialSet({e: (v, v)})
```

Define the homset:

```
sage: H = Hom(X, Y)
```

Now define a morphism by specifying a dictionary:

```
sage: H({v: v, w: v, e: e, f: e})
Simplicial set morphism:
  From: Simplicial set with 4 non-degenerate simplices
  To:   Simplicial set with 2 non-degenerate simplices
  Defn: [v, w, e, f] --> [v, v, e, e]
```

**an_element**()

Return an element of this homset: a constant map.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: S2 = simplicial_sets.Sphere(2)
sage: Hom(S2, S1).an_element()
Simplicial set morphism:
  From: S^2
  To:   S^1
  Defn: Constant map at v_0

sage: K = simplicial_sets.Simplex(3)
sage: L = simplicial_sets.Simplex(4)
sage: d = {K.n_cells(3)[0]: L.n_cells(0)[0].apply_degeneracies(2, 1, 0)}
sage: Hom(K,L)(d) == Hom(K,L).an_element()
True
```

**constant_map**(*point=None*)

Return the constant map in this homset.

INPUT:

- `point` – (default: `None`) if specified, it must be a 0-simplex in the codomain, and it will be the target of the constant map

If `point` is specified, it is the target of the constant map. Otherwise, if the codomain is pointed, the target is its base point. If the codomain is not pointed and `point` is not specified, raise an error.

EXAMPLES:

```
sage: S3 = simplicial_sets.Sphere(3)
sage: T = simplicial_sets.Torus()
sage: T.n_cells(0)[0].rename('w')
sage: Hom(S3,T).constant_map()
Simplicial set morphism:
  From: S^3
  To:   Torus
  Defn: Constant map at w

sage: S0 = simplicial_sets.Sphere(0)
sage: v, w = S0.n_cells(0)
sage: Hom(S3, S0).constant_map(v)
Simplicial set morphism:
  From: S^3
  To:   S^0
  Defn: Constant map at v_0
sage: Hom(S3, S0).constant_map(w)
Simplicial set morphism:
  From: S^3
  To:   S^0
  Defn: Constant map at w_0
```

This constant map is not pointed, since it doesn't send the base point of $S^3$ to the base point of $S^0$:

```
sage: Hom(S3, S0).constant_map(w).is_pointed()
False
```

**diagonal_morphism**()

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

EXAMPLES:

```
sage: RP2 = simplicial_sets.RealProjectiveSpace(2)                    #␣
↪needs sage.groups
sage: Hom(RP2, RP2.product(RP2)).diagonal_morphism()                  #␣
↪needs sage.groups
Simplicial set morphism:
  From: RP^2
  To:   RP^2 x RP^2
  Defn: [1, f, f * f] --> [(1, 1), (f, f), (f * f, f * f)]
```

**identity**()

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

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: Hom(S1, S1).identity()
Simplicial set endomorphism of S^1
  Defn: Identity map
sage: T = simplicial_sets.Torus()
sage: Hom(S1, T).identity()
Traceback (most recent call last):
...
TypeError: identity map is only defined for endomorphism sets
```

**class** sage.topology.simplicial_set_morphism.**SimplicialSetMorphism**(*data=None*, *domain=None*, *codomain=None*, *constant=None*, *identity=False*, *check=True*)

Bases: Morphism

Return a morphism of simplicial sets.

INPUT:

- data – (optional) dictionary defining the map
- domain – simplicial set
- codomain – simplicial set
- constant – (default: None) if not None, then this should be a vertex in the codomain, in which case return the constant map with this vertex as the target
- identity – boolean (default: False); if True, return the identity morphism
- check – boolean (default: True); if True, check that this is actually a morphism: it commutes with the face maps

So to define a map, you must specify domain and codomain. If the map is constant, specify the target (a vertex in the codomain) as constant. If the map is the identity map, specify identity=True. Otherwise, pass a dictionary, data. The keys of the dictionary are the nondegenerate simplices of the domain, the corresponding values are simplices in the codomain.

In fact, the keys in `data` do not need to include all of the nondegenerate simplices, only those which are not faces of other nondegenerate simplices: if $\sigma$ is a face of $\tau$, then the image of $\sigma$ need not be specified.

EXAMPLES:

```
sage: from sage.topology.simplicial_set_morphism import SimplicialSetMorphism
sage: K = simplicial_sets.Simplex(1)
sage: S1 = simplicial_sets.Sphere(1)
sage: v0 = K.n_cells(0)[0]
sage: v1 = K.n_cells(0)[1]
sage: e01 = K.n_cells(1)[0]
sage: w = S1.n_cells(0)[0]
sage: sigma = S1.n_cells(1)[0]

sage: f = {v0: w, v1: w, e01: sigma}
sage: SimplicialSetMorphism(f, K, S1)
Simplicial set morphism:
  From: 1-simplex
  To:   S^1
  Defn: [(0,), (1,), (0, 1)] --> [v_0, v_0, sigma_1]
```

The same map can be defined as follows:

```
sage: H = Hom(K, S1)
sage: H(f)
Simplicial set morphism:
  From: 1-simplex
  To:   S^1
  Defn: [(0,), (1,), (0, 1)] --> [v_0, v_0, sigma_1]
```

Also, this map can be defined by specifying where the 1-simplex goes; the vertices then go where they have to, to satisfy the condition $d_i \circ f = f \circ d_i$:

```
sage: H = Hom(K, S1)
sage: H({e01: sigma})
Simplicial set morphism:
  From: 1-simplex
  To:   S^1
  Defn: [(0,), (1,), (0, 1)] --> [v_0, v_0, sigma_1]
```

A constant map:

```
sage: g = {e01: w.apply_degeneracies(0)}
sage: SimplicialSetMorphism(g, K, S1)
Simplicial set morphism:
  From: 1-simplex
  To:   S^1
  Defn: Constant map at v_0
```

The same constant map:

```
sage: SimplicialSetMorphism(domain=K, codomain=S1, constant=w)
Simplicial set morphism:
  From: 1-simplex
  To:   S^1
  Defn: Constant map at v_0
```

An identity map:

```
sage: SimplicialSetMorphism(domain=K, codomain=K, identity=True)
Simplicial set endomorphism of 1-simplex
  Defn: Identity map
```

Defining a map by specifying it on only some of the simplices in the domain:

```
sage: S5 = simplicial_sets.Sphere(5)
sage: s = S5.n_cells(5)[0]
sage: one = S5.Hom(S5)({s: s})
sage: one
Simplicial set endomorphism of S^5
  Defn: Identity map
```

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

Return the associated chain complex morphism of `self`.

INPUT:

- `base_ring` – default `ZZ`
- `augmented` – boolean (default: `False`); if `True`, return the augmented complex
- `cochain` – boolean (default: `False`); if `True`, return the cochain complex

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: v0 = S1.n_cells(0)[0]
sage: e = S1.n_cells(1)[0]
sage: f = {v0: v0, e: v0.apply_degeneracies(0)} # constant map
sage: g = Hom(S1, S1)(f)
sage: g.associated_chain_complex_morphism().to_matrix()                    #␣
↪needs sage.modules
[1|0]
[-+-]
[0|0]
```

**coequalizer**(*other*)

Return the coequalizer of this map with `other`.

INPUT:

- `other` – a morphism with the same domain and codomain as this map

If the two maps are $f, g : X \to Y$, then the coequalizer $P$ is constructed as the pushout

```
X v Y --> Y
  |       |
  V       V
  Y ----> P
```

where the upper left corner is the coproduct of $X$ and $Y$ (the wedge if they are pointed, the disjoint union otherwise), and the two maps $X \amalg Y \to Y$ are $f \amalg 1$ and $g \amalg 1$.

EXAMPLES:

```
sage: L = simplicial_sets.Simplex(2)
sage: pt = L.n_cells(0)[0]
sage: e = L.n_cells(1)[0]
sage: K = L.subsimplicial_set([e])
```

(continues on next page)

(continued from previous page)

```
sage: f = K.inclusion_map()
sage: v,w = K.n_cells(0)
sage: g = Hom(K,L)({v:pt, w:pt, e:pt.apply_degeneracies(0)})
sage: P = f.coequalizer(g); P
Pushout of maps:
  Simplicial set morphism:
    From: Disjoint union: (Simplicial set with 3 non-degenerate simplices u 2-
↪simplex)
    To:   2-simplex
    Defn: ...
  Simplicial set morphism:
    From: Disjoint union: (Simplicial set with 3 non-degenerate simplices u 2-
↪simplex)
    To:   2-simplex
    Defn: ...
```

**coproduct** (**others*)

Return the coproduct of this map with `others`.

- `others` – morphisms of simplicial sets

If the relevant maps are $f_i : X_i \to Y_i$, this returns the natural map $\amalg X_i \to \amalg Y_i$.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: f = Hom(S1,S1).identity()
sage: f.coproduct(f).is_bijective()
True
sage: g = S1.constant_map(S1)
sage: g.coproduct(g).is_bijective()
False
```

**equalizer** (*other*)

Return the equalizer of this map with `other`.

INPUT:

- `other` – a morphism with the same domain and codomain as this map

If the two maps are $f, g : X \to Y$, then the equalizer $P$ is constructed as the pullback

```
P ----> X
|       |
V       V
X --> X x Y
```

where the two maps $X \to X \times Y$ are $(1, f)$ and $(1, g)$.

EXAMPLES:

```
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: x = AbstractSimplex(0, name='x')
sage: evw = AbstractSimplex(1, name='vw')
sage: evx = AbstractSimplex(1, name='vx')
sage: ewx = AbstractSimplex(1, name='wx')
```

(continues on next page)

(continued from previous page)

```
sage: X = SimplicialSet({evw: (w, v), evx: (x, v)})
sage: Y = SimplicialSet({evw: (w, v), evx: (x, v), ewx: (x, w)})
```

Here $X$ is a wedge of two 1-simplices (a horn, that is), and $Y$ is the boundary of a 2-simplex. The map $f$ includes the two 1-simplices into $Y$, while the map $g$ maps both 1-simplices to the same edge in $Y$.

```
sage: f = Hom(X, Y)({v:v, w:w, x:x, evw:evw, evx:evx})
sage: g = Hom(X, Y)({v:v, w:x, x:x, evw:evx, evx:evx})
sage: P = f.equalizer(g)
sage: P
Pullback of maps:
  Simplicial set morphism:
    From: Simplicial set with 5 non-degenerate simplices
    To:   Simplicial set with 5 non-degenerate simplices x Simplicial set␣
↪with 6 non-degenerate simplices
    Defn: [v, w, x, vw, vx] --> [(v, v), (w, w), (x, x), (vw, vw), (vx, vx)]
  Simplicial set morphism:
    From: Simplicial set with 5 non-degenerate simplices
    To:   Simplicial set with 5 non-degenerate simplices x Simplicial set␣
↪with 6 non-degenerate simplices
    Defn: [v, w, x, vw, vx] --> [(v, v), (w, x), (x, x), (vw, vx), (vx, vx)]
```

**image**()

Return the image of this morphism as a subsimplicial set of the codomain.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: T = S1.product(S1)
sage: K = T.factor(0, as_subset=True)
sage: f = S1.Hom(T)({S1.n_cells(0)[0]: K.n_cells(0)[0],
....:                S1.n_cells(1)[0]: K.n_cells(1)[0]}); f
Simplicial set morphism:
  From: S^1
  To:   S^1 x S^1
  Defn: [v_0, sigma_1] --> [(v_0, v_0), (sigma_1, s_0 v_0)]
sage: f.image()
Simplicial set with 2 non-degenerate simplices
sage: f.image().homology()                                                    #␣
↪needs sage.modules
{0: 0, 1: Z}

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: B.constant_map().image()
Point
sage: Hom(B,B).identity().image() == B
True
```

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

Return the map in (co)homology induced by this map.

INPUT:

- `base_ring` – must be a field (default: `QQ`)

- cohomology – boolean (default: False); if True, the map induced in cohomology rather than homology

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.topology.simplicial_set import AbstractSimplex, SimplicialSet
sage: v = AbstractSimplex(0, name='v')
sage: w = AbstractSimplex(0, name='w')
sage: e = AbstractSimplex(1, name='e')
sage: f = AbstractSimplex(1, name='f')
sage: X = SimplicialSet({e: (v, w), f: (w, v)})
sage: Y = SimplicialSet({e: (v, v)})
sage: H = Hom(X, Y)
sage: f = H({v: v, w: v, e: e, f: e})
sage: g = f.induced_homology_morphism()
sage: g.to_matrix()
[1|0]
[-+-]
[0|2]
sage: g3 = f.induced_homology_morphism(base_ring=GF(3), cohomology=True)
sage: g3.to_matrix()
[1|0]
[-+-]
[0|2]
```

**is_bijective**()

Return True if this map is bijective.

EXAMPLES:

```
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)                    #␣
↪needs sage.groups
sage: RP2 = RP5.n_skeleton(2)                                         #␣
↪needs sage.groups
sage: RP2.inclusion_map().is_bijective()                              #␣
↪needs sage.groups
False

sage: RP5_2 = RP5.quotient(RP2)                                       #␣
↪needs sage.groups
sage: RP5_2.quotient_map().is_bijective()                             #␣
↪needs sage.groups
False

sage: K = RP5_2.pullback(RP5_2.quotient_map(), RP5_2.base_point_map())  #␣
↪needs sage.groups
sage: f = K.universal_property(RP2.inclusion_map(), RP2.constant_map())  #␣
↪needs sage.groups
sage: f.is_bijective()                                                #␣
↪needs sage.groups
True
```

**is_constant**()

Return True if this morphism is a constant map.

EXAMPLES:

```
sage: K = simplicial_sets.KleinBottle()
sage: S4 = simplicial_sets.Sphere(4)
sage: c = Hom(K, S4).constant_map()
sage: c.is_constant()
True
sage: X = S4.n_skeleton(3) # a point
sage: X.inclusion_map().is_constant()
True
sage: eta = simplicial_sets.HopfMap()
sage: eta.is_constant()
False
```

**is_identity**()

Return `True` if this morphism is an identity map.

EXAMPLES:

```
sage: K = simplicial_sets.Simplex(1)
sage: v0 = K.n_cells(0)[0]
sage: v1 = K.n_cells(0)[1]
sage: e01 = K.n_cells(1)[0]
sage: L = simplicial_sets.Simplex(2).n_skeleton(1)
sage: w0 = L.n_cells(0)[0]
sage: w1 = L.n_cells(0)[1]
sage: w2 = L.n_cells(0)[2]
sage: f01 = L.n_cells(1)[0]
sage: f02 = L.n_cells(1)[1]
sage: f12 = L.n_cells(1)[2]

sage: d = {v0:w0, v1:w1, e01:f01}
sage: f = K.Hom(L)(d)
sage: f.is_identity()
False
sage: d = {w0:v0, w1:v1, w2:v1, f01:e01, f02:e01, f12: v1.apply_
↪degeneracies(0,)}
sage: g = L.Hom(K)(d)
sage: (g*f).is_identity()
True
sage: (f*g).is_identity()
False
sage: (f*g).induced_homology_morphism().to_matrix(1)                    #␣
↪needs sage.modules
[0]

sage: RP5 = simplicial_sets.RealProjectiveSpace(5)                     #␣
↪needs sage.groups
sage: RP5.n_skeleton(2).inclusion_map().is_identity()                  #␣
↪needs sage.groups
False
sage: RP5.n_skeleton(5).inclusion_map().is_identity()                  #␣
↪needs sage.groups
True

sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: Hom(B,B).identity().is_identity()
```

(continues on next page)

(continued from previous page)

```
True
sage: Hom(B,B).constant_map().is_identity()
False
```

**is_injective**()

Return `True` if this map is injective.

EXAMPLES:

```
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)                          #␣
↪needs sage.groups
sage: RP2 = RP5.n_skeleton(2)                                               #␣
↪needs sage.groups
sage: RP2.inclusion_map().is_injective()                                    #␣
↪needs sage.groups
True

sage: RP5_2 = RP5.quotient(RP2)                                             #␣
↪needs sage.groups
sage: RP5_2.quotient_map().is_injective()                                   #␣
↪needs sage.groups
False

sage: K = RP5_2.pullback(RP5_2.quotient_map(), RP5_2.base_point_map())      #␣
↪needs sage.groups
sage: f = K.universal_property(RP2.inclusion_map(), RP2.constant_map())     #␣
↪needs sage.groups
sage: f.is_injective()                                                      #␣
↪needs sage.groups
True
```

**is_pointed**()

Return `True` if this is a pointed map.

That is, return `True` if the domain and codomain are pointed and this morphism preserves the base point.

EXAMPLES:

```
sage: S0 = simplicial_sets.Sphere(0)
sage: f = Hom(S0,S0).identity()
sage: f.is_pointed()
True
sage: v = S0.n_cells(0)[0]
sage: w = S0.n_cells(0)[1]
sage: g = Hom(S0,S0)({v:v, w:v})
sage: g.is_pointed()
True
sage: t = Hom(S0,S0)({v:w, w:v})
sage: t.is_pointed()
False
```

**is_surjective**()

Return `True` if this map is surjective.

EXAMPLES:

```
sage: RP5 = simplicial_sets.RealProjectiveSpace(5)                    #␣
↪needs sage.groups
sage: RP2 = RP5.n_skeleton(2)                                         #␣
↪needs sage.groups
sage: RP2.inclusion_map().is_surjective()                             #␣
↪needs sage.groups
False

sage: RP5_2 = RP5.quotient(RP2)                                       #␣
↪needs sage.groups
sage: RP5_2.quotient_map().is_surjective()                            #␣
↪needs sage.groups
True

sage: K = RP5_2.pullback(RP5_2.quotient_map(), RP5_2.base_point_map())  #␣
↪needs sage.groups
sage: f = K.universal_property(RP2.inclusion_map(), RP2.constant_map())  #␣
↪needs sage.groups
sage: f.is_surjective()                                               #␣
↪needs sage.groups
True
```

**mapping_cone**()

Return the mapping cone defined by this map.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: v_0, sigma_1 = S1.nondegenerate_simplices()
sage: K = simplicial_sets.Simplex(2).n_skeleton(1)
```

The mapping cone will be a little smaller if we use only pointed simplicial sets. $S^1$ is already pointed, but not $K$.

```
sage: L = K.set_base_point(K.n_cells(0)[0])
sage: u,v,w = L.n_cells(0)
sage: e,f,g = L.n_cells(1)
sage: h = L.Hom(S1)({u:v_0, v:v_0, w:v_0, e:sigma_1,
....:                f:v_0.apply_degeneracies(0), g:sigma_1})
sage: h
Simplicial set morphism:
  From: Simplicial set with 6 non-degenerate simplices
  To:   S^1
  Defn: [(0,), (1,), (2,), (0, 1), (0, 2), (1, 2)]
        --> [v_0, v_0, v_0, sigma_1, s_0 v_0, sigma_1]
sage: h.induced_homology_morphism().to_matrix()                       #␣
↪needs sage.modules
[1|0]
[-+-]
[0|2]
sage: X = h.mapping_cone()
sage: X.homology() == simplicial_sets.RealProjectiveSpace(2).homology()  #␣
↪needs sage.groups sage.modules
True
```

**n_skeleton**(*n*, *domain=None*, *codomain=None*)

Return the restriction of this morphism to the n-skeleta of the domain and codomain

INPUT:

- `n` – the dimension
- `domain` – (optional) the domain. Specify this to explicitly specify the domain; otherwise, Sage will attempt to compute it. Specifying this can be useful if the domain is built as a pushout or pullback, so trying to compute it may lead to computing the $n$-skeleton of a map, causing an infinite recursion. (Users should not have to specify this, but it may be useful for developers.)
- `codomain` – (optional) the codomain

EXAMPLES:

```
sage: # needs sage.groups
sage: G = groups.misc.MultiplicativeAbelian([2])
sage: B = simplicial_sets.ClassifyingSpace(G)
sage: one = Hom(B,B).identity()
sage: one.n_skeleton(3)
Simplicial set endomorphism of Simplicial set with 4 non-degenerate simplices
  Defn: Identity map
sage: c = Hom(B,B).constant_map()
sage: c.n_skeleton(3)
Simplicial set endomorphism of Simplicial set with 4 non-degenerate simplices
  Defn: Constant map at 1

sage: K = simplicial_sets.Simplex(2)
sage: L = K.subsimplicial_set(K.n_cells(0)[:2])
sage: L.nondegenerate_simplices()
[(0,), (1,)]
sage: L.inclusion_map()
Simplicial set morphism:
  From: Simplicial set with 2 non-degenerate simplices
  To:   2-simplex
  Defn: [(0,), (1,)] --> [(0,), (1,)]
sage: L.inclusion_map().n_skeleton(1)
Simplicial set morphism:
  From: Simplicial set with 2 non-degenerate simplices
  To:   Simplicial set with 6 non-degenerate simplices
  Defn: [(0,), (1,)] --> [(0,), (1,)]
```

**product**(**others*)

Return the product of this map with `others`.

- `others` – morphisms of simplicial sets

If the relevant maps are $f_i : X_i \to Y_i$, this returns the natural map $\prod X_i \to \prod Y_i$.

EXAMPLES:

```
sage: S1 = simplicial_sets.Sphere(1)
sage: f = Hom(S1,S1).identity()
sage: f.product(f).is_bijective()
True
sage: g = S1.constant_map(S1)
sage: g.product(g).is_bijective()
False
```

**pullback**(**others*)

Return the pullback of this morphism along with `others`.

INPUT:

- `others` – morphisms of simplicial sets, the codomains of which must all equal that of `self`

This returns the pullback as a simplicial set. See *sage.topology.simplicial_set_constructions.PullbackOfSimplicialSets* for more documentation and examples.

EXAMPLES:

```
sage: T = simplicial_sets.Torus()
sage: K = simplicial_sets.KleinBottle()
sage: term_T = T.constant_map()
sage: term_K = K.constant_map()
sage: P = term_T.pullback(term_K); P  # the product as a pullback
Pullback of maps:
  Simplicial set morphism:
    From: Torus
    To:   Point
    Defn: Constant map at *
  Simplicial set morphism:
    From: Klein bottle
    To:   Point
    Defn: Constant map at *
```

**pushout** (**others*)

Return the pushout of this morphism along with `others`.

INPUT:

- `others` – morphisms of simplicial sets, the domains of which must all equal that of `self`

This returns the pushout as a simplicial set. See *sage.topology.simplicial_set_constructions.PushoutOfSimplicialSets* for more documentation and examples.

EXAMPLES:

```
sage: T = simplicial_sets.Torus()
sage: K = simplicial_sets.KleinBottle()
sage: init_T = T._map_from_empty_set()
sage: init_K = K._map_from_empty_set()
sage: D = init_T.pushout(init_K); D  # the disjoint union as a pushout
Pushout of maps:
  Simplicial set morphism:
    From: Empty simplicial set
    To:   Torus
    Defn: [] --> []
  Simplicial set morphism:
    From: Empty simplicial set
    To:   Klein bottle
    Defn: [] --> []
```

**suspension** (*n=1*)

Return the $n$-th suspension of this morphism of simplicial sets.

INPUT:

- `n` – nonnegative integer (default: 1)

EXAMPLES:

```
sage: eta = simplicial_sets.HopfMap()
sage: mc_susp_eta = eta.suspension().mapping_cone()
sage: susp_mc_eta = eta.mapping_cone().suspension()
sage: mc_susp_eta.homology() == susp_mc_eta.homology()                    #␣
↪needs sage.modules
True
```

This uses reduced suspensions if the original morphism is pointed, unreduced otherwise. So for example, if a constant map is not pointed, its suspension is not a constant map:

```
sage: L = simplicial_sets.Simplex(1)
sage: L.constant_map().is_pointed()
False
sage: f = L.constant_map().suspension()
sage: f.is_constant()
False

sage: K = simplicial_sets.Sphere(3)
sage: K.constant_map().is_pointed()
True
sage: g = K.constant_map().suspension()
sage: g.is_constant()
True

sage: h = K.identity().suspension()
sage: h.is_identity()
True
```

CHAPTER

# TWELVE

# GENERIC CELL COMPLEXES

AUTHORS:

- John H. Palmieri (2009-08)

This module defines a class of abstract finite cell complexes. This is meant as a base class from which other classes (like `SimplicialComplex`, `CubicalComplex`, and `DeltaComplex`) should derive. As such, most of its properties are not implemented. It is meant for use by developers producing new classes, not casual users.

**Note**

Keywords for *chain_complex()*, *homology()*, etc.: any keywords given to the *homology()* method get passed on to the *chain_complex()* method and also to the constructor for chain complexes in `sage.homology.chain_complex.ChainComplex_class`, as well as its associated `homology()` method. This means that those keywords should have consistent meaning in all of those situations. It also means that it is easy to implement new keywords: for example, if you implement a new keyword for the `sage.homology.chain_complex.ChainComplex_class.homology()` method, then it will be automatically accessible through the *homology()* method for cell complexes – just make sure it gets documented.

**class** sage.topology.cell_complex.**GenericCellComplex**

Bases: `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 $\Delta$-complexes.

EXAMPLES:

It's hard to give informative examples of the base class, since essentially nothing is implemented.

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
```

**alexander_whitney**(*cell*, *dim_left*)

The decomposition of `cell` in this complex into left and right factors, suitable for computing cup products. This should provide a cellular approximation for the diagonal map $K \to K \times K$.

This method is not implemented for generic cell complexes, but must be implemented for any derived class to make cup products work in `self.cohomology_ring()`.

INPUT:

- `cell` – a cell in this complex
- `dim_left` – the dimension of the left-hand factors in the decomposition

OUTPUT: list containing triples `(c, left, right)`. `left` and `right` should be cells in this complex, and `c` an integer. In the cellular approximation of the diagonal map, the chain represented by `cell` should get sent to the sum of terms $c(left \otimes right)$ in the tensor product $C(K) \otimes C(K)$ of the chain complex for this complex with itself.

This gets used in the method `product_on_basis()` for the class of cohomology rings.

For simplicial and cubical complexes, the decomposition can be done at the level of individual cells: see `alexander_whitney()` and `alexander_whitney()`. Then the method for simplicial complexes just calls the method for individual simplices, and similarly for cubical complexes. For $\Delta$-complexes and simplicial sets, the method is instead defined at the level of the cell complex.

EXAMPLES:

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.alexander_whitney(None, 2)
Traceback (most recent call last):
...
NotImplementedError: <abstract method alexander_whitney at ...>
```

**algebraic_topological_model** (*base_ring=Rational Field*)

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

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

This is not implemented for generic cell complexes. For any classes deriving from this one, when this method is implemented, it should essentially just call either `algebraic_topological_model()` or `algebraic_topological_model_delta_complex()`.

EXAMPLES:

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.algebraic_topological_model(QQ)
Traceback (most recent call last):
...
NotImplementedError
```

**betti** (*dim=None*, *subcomplex=None*)

The Betti numbers of this simplicial complex as a dictionary (or a single Betti number, if only one dimension is given): the $i$-th Betti number is the rank of the $i$-th homology group.

INPUT:

- `dim` – integer or list of integers or `None` (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` – a subcomplex (default: `None`) 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()                                                    #␣
↪needs sage.modules
{0: 1, 1: 0, 2: 1}
sage: (S*S*S).betti([1,2])                                               #␣
↪needs sage.modules
{1: 0, 2: 1}
sage: (S*S*S).betti(2)                                                   #␣
↪needs sage.modules
1
```

Or build the two-sphere as a $\Delta$-complex:

```
sage: S2 = delta_complexes.Sphere(2)
sage: S2.betti([1,2])                                                    #␣
↪needs sage.modules
{1: 0, 2: 1}
```

Or as a cubical complex:

```
sage: S2c = cubical_complexes.Sphere(2)
sage: S2c.betti(2)                                                       #␣
↪needs sage.modules
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 cells which are *not* in the subcomplex.

INPUT:

- `subcomplex` – subcomplex (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.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.cells()
Traceback (most recent call last):
...
NotImplementedError: <abstract method cells at ...>
```

**chain_complex** (*subcomplex=None*, *augmented=False*, *verbose=False*, *check=True*, *dimensions=None*, *base_ring='ZZ'*, *cochain=False*)

This is not implemented for general cell complexes.

Some keywords to possibly implement in a derived class:

- `subcomplex` – a subcomplex: compute the relative chain complex

- augmented – a bool: whether to return the augmented complex
- verbose – a bool: whether to print informational messages as the chain complex is being computed
- check – a bool: whether to check that the each composite of two consecutive differentials is zero
- dimensions – if None, compute the chain complex in all dimensions. If a list or tuple of integers, compute the chain complex in those dimensions, setting the chain groups in all other dimensions to zero.

Definitely implement the following:

- base_ring – commutative ring (default: ZZ)
- cochain – a bool: whether to return the cochain complex

EXAMPLES:

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.chain_complex()
Traceback (most recent call last):
...
NotImplementedError: <abstract method chain_complex at ...>
```

**cohomology**(*dim=None*, *base_ring=Integer Ring*, *subcomplex=None*, *generators=False*, *algorithm='pari'*, *verbose=False*, *reduced=True*)

The reduced cohomology of this cell complex.

The arguments are the same as for the *homology()* method, except that *homology()* accepts a cohomology key word, while this function does not: cohomology is automatically true here. Indeed, this function just calls *homology()* with cohomology set to True.

INPUT:

- dim
- base_ring
- subcomplex
- algorithm
- verbose
- reduced

EXAMPLES:

```
sage: circle = SimplicialComplex([[0,1], [1,2], [0, 2]])
sage: circle.cohomology(0)                                                  #␣
↪needs sage.modules
0
sage: circle.cohomology(1)                                                  #␣
↪needs sage.modules
Z
```

Projective plane:

```
sage: # needs sage.modules
sage: P2 = SimplicialComplex([[0,1,2], [0,2,3], [0,1,5], [0,4,5], [0,3,4],
....:                         [1,2,4], [1,3,4], [1,3,5], [2,3,5], [2,4,5]])
sage: P2.cohomology(2)
C2
```

(continues on next page)

(continued from previous page)

```
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)                             #␣
↪needs sage.modules
C2
```

Relative cohomology:

```
sage: T = SimplicialComplex([[0,1]])
sage: U = SimplicialComplex([[0], [1]])
sage: T.cohomology(1, subcomplex=U)                                             #␣
↪needs sage.modules
Z
```

A $\Delta$-complex example:

```
sage: s5 = delta_complexes.Sphere(5)
sage: s5.cohomology(base_ring=GF(7))[5]                                         #␣
↪needs sage.modules
Vector space of dimension 1 over Finite Field of size 7
```

**cohomology_ring** (*base_ring=Rational Field*)

Return the unreduced cohomology with coefficients in `base_ring` with a chosen basis.

This is implemented for simplicial, cubical, and $\Delta$-complexes, not for arbitrary generic cell complexes. The resulting elements are suitable for computing cup products. For simplicial complexes, they should be suitable for computing cohomology operations; so far, only mod 2 cohomology operations have been implemented.

INPUT:

- `base_ring` – coefficient ring (default: `QQ`); must be a field

The basis elements in dimension `dim` are named 'h^{dim,i}' where $i$ ranges between 0 and $r-1$, if $r$ is the rank of the cohomology group.

**Note**

For all but the smallest complexes, this is likely to be slower than *cohomology()* (with field coefficients), possibly by several orders of magnitude. This and its companion *homology_with_basis()* carry extra information which allows computation of cup products, for example, but because of speed issues, you may only wish to use these if you need that extra information.

EXAMPLES:

```
sage: # needs sage.modules
sage: K = simplicial_complexes.KleinBottle()
sage: H = K.cohomology_ring(QQ); H
Cohomology ring of Minimal triangulation of the Klein bottle
 over Rational Field
sage: sorted(H.basis(), key=str)
[h^{0,0}, h^{1,0}]
sage: H = K.cohomology_ring(GF(2)); H
```

(continues on next page)

(continued from previous page)

```
Cohomology ring of Minimal triangulation of the Klein bottle
 over Finite Field of size 2
sage: sorted(H.basis(), key=str)
[h^{0,0}, h^{1,0}, h^{1,1}, h^{2,0}]

sage: X = delta_complexes.SurfaceOfGenus(2)
sage: H = X.cohomology_ring(QQ); H                                          #␣
↪needs sage.modules
Cohomology ring of Delta complex with 3 vertices and 29 simplices
 over Rational Field
sage: sorted(H.basis(1), key=str)                                           #␣
↪needs sage.modules
[h^{1,0}, h^{1,1}, h^{1,2}, h^{1,3}]

sage: H = simplicial_complexes.Torus().cohomology_ring(QQ); H               #␣
↪needs sage.modules
Cohomology ring of Minimal triangulation of the torus
 over Rational Field
sage: x = H.basis()[1,0]; x                                                 #␣
↪needs sage.modules
h^{1,0}
sage: y = H.basis()[1,1]; y                                                 #␣
↪needs sage.modules
h^{1,1}
```

You can compute cup products of cohomology classes:

```
sage: # needs sage.modules
sage: x.cup_product(y)
-h^{2,0}
sage: x * y # alternate notation
-h^{2,0}
sage: y.cup_product(x)
h^{2,0}
sage: x.cup_product(x)
0
```

Cohomology operations:

```
sage: # needs sage.groups
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: K = RP2.suspension()
sage: K.set_immutable()
sage: y = K.cohomology_ring(GF(2)).basis()[2,0]; y                          #␣
↪needs sage.modules
h^{2,0}
sage: y.Sq(1)                                                               #␣
↪needs sage.modules
h^{3,0}
```

To compute the cohomology ring, the complex must be "immutable". This is only relevant for simplicial complexes, and most simplicial complexes are immutable, but certain constructions make them mutable. The suspension is one example, and this is the reason for calling `K.set_immutable()` above. Another example:

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

(continues on next page)

(continued from previous page)

```
sage: T = S1.product(S1)
sage: T.is_immutable()
False
sage: T.cohomology_ring()                                                   #␣
↪needs sage.modules
Traceback (most recent call last):
...
ValueError: this simplicial complex must be immutable; call set_immutable()
sage: T.set_immutable()
sage: T.cohomology_ring()                                                   #␣
↪needs sage.modules
Cohomology ring of Simplicial complex with 9 vertices and
 18 facets over Rational Field
```

**dimension**()

The dimension of this cell complex: the maximum dimension of its cells.

**Warning**

If the *cells()* method calls *dimension()*, then you'll get an infinite loop. So either don't use *dimension()* or override *dimension()*.

EXAMPLES:

```
sage: simplicial_complexes.RandomComplex(d=5, n=8).dimension()
5
sage: delta_complexes.Sphere(3).dimension()
3
sage: T = cubical_complexes.Torus()
sage: T.product(T).dimension()
4
```

**disjoint_union**(*right*)

The disjoint union of this cell complex with another one.

INPUT:

- `right` – the other cell complex (the right-hand factor)

Disjoint unions are not implemented for general cell complexes.

EXAMPLES:

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex(); B = GenericCellComplex()
sage: A.disjoint_union(B)
Traceback (most recent call last):
...
NotImplementedError: <abstract method disjoint_union at ...>
```

**euler_characteristic**()

The Euler characteristic of this cell complex: the alternating sum over $n \geq 0$ of the number of $n$-cells.

EXAMPLES:

```
sage: simplicial_complexes.Simplex(5).euler_characteristic()
1
sage: delta_complexes.Sphere(6).euler_characteristic()
2
sage: cubical_complexes.KleinBottle().euler_characteristic()
0
```

**f_vector**()

The $f$-vector of this cell complex: a list whose $n$-th item is the number of $(n-1)$-cells. Note that, like all lists in Sage, this is indexed starting at 0: the 0th element in this list is the number of $(-1)$-cells (which is 1: the empty cell is the only $(-1)$-cell).

EXAMPLES:

```
sage: simplicial_complexes.KleinBottle().f_vector()
[1, 8, 24, 16]
sage: delta_complexes.KleinBottle().f_vector()
[1, 1, 3, 2]
sage: cubical_complexes.KleinBottle().f_vector()
[1, 42, 84, 42]
```

**face_poset**()

The face poset of this cell complex, the poset of nonempty cells, ordered by inclusion.

This uses the *cells()* method, and also assumes that for each cell `f`, all of `f.faces()`, `tuple(f)`, and `f.dimension()` make sense. (If this is not the case in some derived class, as happens with $\Delta$-complexes, then override this method.)

EXAMPLES:

```
sage: P = SimplicialComplex([[0, 1], [1,2], [2,3]]).face_poset(); P
Finite poset containing 7 elements
sage: sorted(P.list())
[(0,), (0, 1), (1,), (1, 2), (2,), (2, 3), (3,)]

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.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.graph()
Traceback (most recent call last):
...
NotImplementedError
```

**homology**(*dim=None*, *base_ring=Integer Ring*, *subcomplex=None*, *generators=False*, *cohomology=False*, *algorithm='pari'*, *verbose=False*, *reduced=True*, ***kwds*)

The (reduced) homology of this cell complex.

INPUT:

- dim – integer or list of integers or None (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 – commutative ring (default: ZZ); must be **Z** or a field
- subcomplex – (default: empty) a subcomplex of this simplicial complex. Compute the homology relative to this subcomplex.
- generators – boolean (default: False); if True, return generators for the homology groups along with the groups.
- cohomology – boolean (default: False); if True, compute cohomology rather than homology
- algorithm – string (default: 'pari'); the algorithm options are 'auto', 'dhsw', or 'pari'. See below for a description of what they mean.
- verbose – boolean (default: False); if True, print some messages as the homology is computed
- reduced – boolean (default: True); if True, return the reduced homology

ALGORITHM:

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', then for each relatively small matrix, use the standard Sage method, which calls the Pari package. For any large matrix, reduce it using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm [DHSW2003]: see dhsw_snf() for details.

'no_chomp' is a synonym for 'auto', maintained for backward-compatibility.

algorithm may also be 'pari' or 'dhsw', which forces the named algorithm to be used regardless of the size of the matrices.

As of this writing, 'pari' is the fastest standard option.

EXAMPLES:

```
sage: # needs sage.modules
sage: P = delta_complexes.RealProjectivePlane()
sage: P.homology()
{0: 0, 1: C2, 2: 0}
sage: P.homology(reduced=False)
{0: Z, 1: C2, 2: 0}
sage: P.homology(base_ring=GF(2))
{0: Vector space of dimension 0 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: S7 = delta_complexes.Sphere(7)
sage: S7.homology(7)
Z
sage: cubical_complexes.KleinBottle().homology(1, base_ring=GF(2))
Vector space of dimension 2 over Finite Field of size 2
```

Sage can compute generators of homology groups:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: S2.homology(dim=2, generators=True, base_ring=GF(2))                  #␣
↪needs sage.modules
[(Vector space of dimension 1 over Finite Field of size 2,
  (0, 1, 2) + (0, 1, 3) + (0, 2, 3) + (1, 2, 3))]
```

When generators are computed, Sage returns a pair for each dimension: the group and the list of generators. For simplicial complexes, each generator is represented as a linear combination of simplices, as above, and for cubical complexes, each generator is a linear combination of cubes:

```
sage: S2_cub = cubical_complexes.Sphere(2)
sage: S2_cub.homology(dim=2, generators=True)                                  #␣
↪needs sage.modules
[(Z,
 [0,0] x [0,1] x [0,1] - [0,1] x [0,0] x [0,1] + [0,1] x [0,1] x [0,0]
 - [0,1] x [0,1] x [1,1] + [0,1] x [1,1] x [0,1] - [1,1] x [0,1] x [0,1])]
```

Similarly for simpicial sets:

```
sage: S = simplicial_sets.Sphere(2)
sage: S.homology(generators=True)                                              #␣
↪needs sage.modules
{0: [], 1: 0, 2: [(Z, sigma_2)]}
```

**homology_with_basis**(*base_ring=Rational Field*, *cohomology=False*)

Return the unreduced homology of this complex with coefficients in `base_ring` with a chosen basis.

This is implemented for simplicial, cubical, and $\Delta$-complexes, not for arbitrary generic cell complexes.

INPUT:

- `base_ring` – coefficient ring (default: `QQ`); must be a field
- `cohomology` – boolean (default: `False`); if `True`, return cohomology instead of homology

Homology basis elements are named 'h_{dim,i}' where i ranges between 0 and $r-1$, if $r$ is the rank of the homology group. Cohomology basis elements are denoted $h^{dim,i}$ instead.

**See also**

If `cohomology` is `True`, this returns the cohomology as a ring: it calls *cohomology_ring()*.

EXAMPLES:

```
sage: # needs sage.modules
sage: K = simplicial_complexes.KleinBottle()
sage: H = K.homology_with_basis(QQ); H
Homology module of Minimal triangulation of the Klein bottle
 over Rational Field
sage: sorted(H.basis(), key=str)
[h_{0,0}, h_{1,0}]
sage: H = K.homology_with_basis(GF(2)); H
Homology module of Minimal triangulation of the Klein bottle
 over Finite Field of size 2
sage: sorted(H.basis(), key=str)
[h_{0,0}, h_{1,0}, h_{1,1}, h_{2,0}]
```

The homology is constructed as a graded object, so for example, you can ask for the basis in a single degree:

```
sage: H.basis(1)                                                               #␣
↪needs sage.modules
Finite family {(1, 0): h_{1,0}, (1, 1): h_{1,1}}
```

(continues on next page)

(continued from previous page)

```
sage: S3 = delta_complexes.Sphere(3)
sage: H = S3.homology_with_basis(QQ, cohomology=True)                    #␣
↪needs sage.modules
sage: list(H.basis(3))                                                   #␣
↪needs sage.modules
[h^{3,0}]
```

**is_acyclic**(*base_ring=Integer Ring*)

Return `True` if the reduced homology with coefficients in `base_ring` of this cell complex is zero.

INPUT:

- `base_ring` – (default: `ZZ`) compute homology with coefficients in this ring

EXAMPLES:

```
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: RP2.is_acyclic()                                                   #␣
↪needs sage.modules
False
sage: RP2.is_acyclic(QQ)                                                 #␣
↪needs sage.modules
True
```

This first computes the Euler characteristic: if it is not 1, the complex cannot be acyclic. So this should return `False` reasonably quickly on complexes with Euler characteristic not equal to 1:

```
sage: K = cubical_complexes.KleinBottle()
sage: C = cubical_complexes.Cube(2)
sage: P = K.product(C); P
Cubical complex with 168 vertices and 1512 cubes
sage: P.euler_characteristic()
0
sage: P.is_acyclic()
False
```

**is_connected**()

Return `True` if this cell complex is connected.

EXAMPLES:

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

(continues on next page)

(continued from previous page)

```
sage: cubical_complexes.Sphere(0).is_connected()
False
sage: cubical_complexes.Sphere(2).is_connected()
True
```

**join** (*right*)

The join of this cell complex with another one.

INPUT:

- `right` – the other cell complex (the right-hand factor)

Joins are not implemented for general cell complexes. They may be implemented in some derived classes (like simplicial complexes).

EXAMPLES:

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex(); B = GenericCellComplex()
sage: A.join(B)
Traceback (most recent call last):
...
NotImplementedError: <abstract method join at ...>
```

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

List of cells of dimension $n$ of this cell complex. If the optional argument `subcomplex` is present, then return the $n$-dimensional cells which are *not* in the subcomplex.

INPUT:

- `n` – nonnegative integer; the dimension
- `subcomplex` – (optional) a subcomplex of this cell complex; return the cells which are not in this subcomplex

**Note**

The resulting list need not be sorted. If you want a sorted list of $n$-cells, use `_n_cells_sorted()`.

EXAMPLES:

```
sage: delta_complexes.Torus().n_cells(1)
[(0, 0), (0, 0), (0, 0)]
sage: cubical_complexes.Cube(1).n_cells(0)
[[1,1], [0,0]]
```

**n_chains** (*n*, *base_ring=Integer Ring*, *cochains=False*)

Return the free module of chains in degree `n` over `base_ring`.

INPUT:

- `n` – integer
- `base_ring` – ring (default: **Z**)
- `cochains` – boolean (default: `False`); if `True`, return cochains instead

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

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: S2.n_chains(1, QQ)                                                    #␣
↪needs sage.modules
Free module generated by {(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)}
 over Rational Field
sage: list(S2.n_chains(1, QQ, cochains=False).basis())                      #␣
↪needs sage.modules
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: list(S2.n_chains(1, QQ, cochains=True).basis())                       #␣
↪needs sage.modules
[\chi_(0, 1), \chi_(0, 2), \chi_(0, 3), \chi_(1, 2), \chi_(1, 3), \chi_(2, 3)]
```

**n_skeleton**(*n*)

The $n$-skeleton of this cell complex: the cell complex obtained by discarding all of the simplices in dimensions larger than $n$.

INPUT:

- `n` – nonnegative integer

This is not implemented for general cell complexes.

EXAMPLES:

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex()
sage: A.n_skeleton(3)
Traceback (most recent call last):
...
NotImplementedError: <abstract method n_skeleton at ...>
```

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

The (Cartesian) product of this cell complex with another one.

Products are not implemented for general cell complexes. They may be implemented in some derived classes (like simplicial complexes).

EXAMPLES:

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex(); B = GenericCellComplex()
sage: A.product(B)
Traceback (most recent call last):
...
NotImplementedError: <abstract method product at ...>
```

**wedge**(*right*)

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

INPUT:

- `right` – the other cell complex (the right-hand factor)

Wedges are not implemented for general cell complexes.

EXAMPLES:

```
sage: from sage.topology.cell_complex import GenericCellComplex
sage: A = GenericCellComplex(); B = GenericCellComplex()
sage: A.wedge(B)
Traceback (most recent call last):
...
NotImplementedError: <abstract method wedge at ...>
```

CHAPTER

# THIRTEEN

# FINITE FILTERED COMPLEXES

AUTHORS:

- Guillaume Rousseau (2021-05)

This module implements the basic structures of finite filtered complexes. A filtered complex is a simplicial complex, where each simplex is given a weight, or "filtration value", such that the weight of a simplex is greater than the weight of each of its faces.

The algorithm used in this module comes from [ZC2005].

EXAMPLES:

```
sage: FilteredSimplicialComplex([([0], 0), ([1], 0), ([0, 1], 1)])
Filtered complex on vertex set (0, 1) and
 with simplices ((0,) : 0), ((1,) : 0), ((0, 1) : 1)
```

Sage can compute persistent homology of simplicial complexes:

```
sage: X = FilteredSimplicialComplex([([0], 0), ([1], 0), ([0, 1], 1)])
sage: X.persistence_intervals(0)                                            #␣
↪needs sage.modules
[(0, 1), (0, +Infinity)]
```

FilteredSimplicialComplex objects are mutable. Filtration values can be set with the `filtration` method as follows:

```
sage: X = FilteredSimplicialComplex() # returns an empty complex
sage: X.persistence_intervals(1)                                            #␣
↪needs sage.modules
[]
sage: X.filtration(Simplex([0, 2]), 0) # recursively adds faces
sage: X.filtration(Simplex([0, 1]), 0)
sage: X.filtration(Simplex([1, 2]), 0)
sage: X.filtration(Simplex([0, 1, 2]), 1) # closes the circle
sage: X.persistence_intervals(1)                                            #␣
↪needs sage.modules
[(0, 1)]
```

The filtration value of a simplex can be accessed as well with the `filtration` method, by not specifying a filtration value in the arguments. If the simplex is not in the complex, this returns `None`:

```
sage: X = FilteredSimplicialComplex([([0], 0), ([1], 0), ([0,1], 1)])
sage: X.filtration(Simplex([0]))
0
sage: X.filtration(Simplex([1,2])) is None
True
```

Filtration values can be accessed with function call and list syntax as follows:

```
sage: X = FilteredSimplicialComplex([([0], 0), ([1], 0), ([0,1], 1)])
sage: s_1 = Simplex([0])
sage: X[s_1]
0
sage: X(Simplex([0,1]))
1
sage: X(Simplex(['baba']))
```

It is also possible to set the filtration value of a simplex with the `insert` method, which takes as argument a list of vertices rather than a `Simplex`. This can make code more readable / clear:

```
sage: X = FilteredSimplicialComplex()
sage: X.insert(['a'], 0)
sage: X.insert(['b', 'c'], 1)
sage: X
Filtered complex on vertex set ('a', 'b', 'c') and with simplices
 (('a',) : 0), (('c',) : 1), (('b',) : 1), (('b', 'c') : 1)
```

**class** sage.topology.filtered_simplicial_complex.**FilteredSimplicialComplex**(*simplices=[]*, *verbose=False*)

Bases: `SageObject`

Define a filtered complex.

INPUT:

- `simplices` – list of simplices and filtration values
- `verbose` – boolean (default: `False`); if `True`, any change to the filtration value of a simplex will be printed

`simplices` should be a list of tuples `(l, v)`, where `l` is a list of vertices and `v` is the corresponding filtration value.

EXAMPLES:

```
sage: FilteredSimplicialComplex([([0], 0), ([1], 0), ([2], 1), ([0,1], 2.27)])
Filtered complex on vertex set (0, 1, 2) and with simplices
 ((0,) : 0), ((1,) : 0), ((2,) : 1), ((0, 1) : 2.27000000000000)
```

**betti_number**(*k*, *a*, *b*, *field=2*, *strict=True*, *verbose=None*)

Return the `k`-dimensional Betti number from `a` to `a + b`.

INPUT:

- `k` – the dimension for the Betti number
- `a` – the lower filtration value
- `b` – the size of the interval
- `field` – prime number (default: 2); modulo which persistent homology is computed
- `strict` – boolean (default: `True`); if `False`, takes into account intervals of persistence 0
- `verbose` – (optional) if `True`, print the steps of the persistent homology computation; the default is the verbosity of `self`

The Betti number $\beta_k^{a,a+b}$ counts the number of homology elements which are alive throughout the whole duration `[a, a+b]`.

EXAMPLES:

```
sage: X = FilteredSimplicialComplex([([0], 0), ([1], 0), ([0,1], 2)])
sage: X.betti_number(0, 0.5, 1)                                                 #␣
↪needs sage.modules
2
sage: X.betti_number(0, 1.5, 1)                                                 #␣
↪needs sage.modules
1
```

If an element vanishes at time `a + b` exactly, it does not count towards the Betti number:

```
sage: X = FilteredSimplicialComplex([([0], 0), ([1], 0), ([0,1], 2)])
sage: X.betti_number(0, 1.5, 0.5)                                               #␣
↪needs sage.modules
1
```

**filtration**(*s*, *filtration_value=None*)

Set filtration value of a simplex, or return value of existing simplex.

INPUT:

- `s` – *Simplex* for which to set or obtain the value of
- `filtration_value` – (optional) filtration value for the simplex

If no filtration value is specified, this returns the value of the simplex in the complex. If the simplex is not in the complex, this returns `None`.

If `filtration_value` is set, this function inserts the simplex into the complex with the specified value. See documentation of *insert()* for more details.

EXAMPLES:

```
sage: X = FilteredSimplicialComplex([([0], 0), ([1], 1)])
sage: X.filtration(Simplex([0, 1])) is None
True
sage: X.filtration(Simplex([0, 1]), 2)
sage: X.filtration([0, 1])
2
```

**insert**(*vertex_list*, *filtration_value*)

Add a simplex to the complex.

All faces of the simplex are added recursively if they are not already present, with the same value. If the simplex is already present, and the new value is lower than its current value in the complex, the value gets updated, otherwise it does not change. This propagates recursively to faces.

If verbose has been enabled, this method will describe what it is doing during an insertion.

INPUT:

- `vertex_list` – list of vertices
- `filtration_value` – desired value of the simplex to be added

EXAMPLES:

```
sage: X = FilteredSimplicialComplex()
sage: X.insert(Simplex([0]),3)
sage: X
Filtered complex on vertex set (0,) and with simplices ((0,) : 3)
```

If the verbose parameter was set to true, this method will print some info:

```
sage: X = FilteredSimplicialComplex(verbose=True)
sage: X.insert(Simplex([0, 1]), 2)
Also inserting face (1,) with value 2
Also inserting face (0,) with value 2
sage: X.insert(Simplex([0]),1)
Face (0,) is already in the complex.
However its value is 2: updating it to 1
sage: X.insert(Simplex([0]), 77)
Face (0,) is already in the complex.
Its value is 1: keeping it that way
```

**persistence_intervals**(*dimension*, *field=2*, *strict=True*, *verbose=None*)

Return the list of $d$-dimensional homology elements.

INPUT:

- `dimension` – integer; dimension $d$ for which to return intervals
- `field` – prime number (default: 2); modulo which persistent homology is computed
- `strict` – boolean (default: `True`); if `False`, takes into account intervals of persistence 0
- `verbose` – (optional) if `True`, print the steps of the persistent homology computation; the default is the verbosity of `self`

EXAMPLES:

```
sage: X = FilteredSimplicialComplex([([0], 0), ([1], 1), ([0,1], 2)])
sage: X.persistence_intervals(0)                                              #␣
↪needs sage.modules
[(1, 2), (0, +Infinity)]
```

**prune**(*threshold*)

Return a copy of the filtered complex, where simplices above the threshold value have been removed.

INPUT:

- `threshold` – a real value, above which simplices are discarded

Simplices with filtration value exactly equal to `threshold` are kept in the result.

EXAMPLES:

```
sage: a = FilteredSimplicialComplex()
sage: a.insert([0], 0)
sage: a.insert([0, 1], 1)
sage: a.insert([0, 2], 2)
sage: b = a.prune(1)
sage: b
Filtered complex on vertex set (0, 1) and
 with simplices ((0,) : 0), ((1,) : 1), ((0, 1) : 1)
```

CHAPTER

# FOURTEEN

# MOMENT-ANGLE COMPLEXES

AUTHORS:

- Ognjen Petrov (2023-06-25): initial version

**class** sage.topology.moment_angle_complex.**MomentAngleComplex**(*simplicial_complex*)

Bases: UniqueRepresentation, SageObject

A moment-angle complex.

Given a simplicial complex $K$, with a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$, a moment-angle complex over $K$ is a topological space $Z$, which is a union of $X_\sigma$, where $\sigma \in K$, and $X_\sigma = Y_{v_1} \times Y_{v_2} \times \cdots \times Y_{v_n}$ and $Y_{v_i}$ is a 2-disk (a 2-simplex) if $v_i \in \sigma$ , or a 1-sphere otherwise.

$$Y_{v_i} = \begin{cases} D^2, & v_i \in \sigma, \\ S^1, & v_i \notin \sigma. \end{cases}$$

**Note**

The mentioned union is not a disjoint union of topological spaces. The unit disks and the unit spheres are considered subsets of $\mathbf{C}$, so the union is just a normal union of subsets of $\mathbf{C}^n$.

Here we view moment-angle complexes as cubical complexes and try to compute mostly things which would not require computing the moment-angle complex itself, but rather work with the corresponding simplicial complex.

**Note**

One of the more useful properties will be the `bigraded Betti numbers`, and the underlying theorem which makes this possible is Hochter's formula, which can be found on page 104 of [BP2014].

INPUT:

- `simplicial_complex` – an instance of `SimplicialComplex`, or an object from which an instance of `SimplicialComplex` can be created (e.g., list of facets), which represents the associated simplicial complex over which this moment-angle complex is created

EXAMPLES:

```
sage: MomentAngleComplex([[1,2,3], [2,4], [3,4]])
Moment-angle complex of Simplicial complex with vertex set
(1, 2, 3, 4) and facets {(2, 4), (3, 4), (1, 2, 3)}
sage: X = SimplicialComplex([[0,1], [1,2], [1,3], [2,3]])
```

(continues on next page)

(continued from previous page)

```
sage: Z = MomentAngleComplex(X); Z
Moment-angle complex of Simplicial complex with vertex set
(0, 1, 2, 3) and facets {(0, 1), (1, 2), (1, 3), (2, 3)}
sage: M = MomentAngleComplex([[1], [2]]); M
Moment-angle complex of Simplicial complex with vertex set
(1, 2) and facets {(1,), (2,)}
```

We can perform a number of operations, such as find the dimension or compute the homology:

```
sage: M.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: Z}
sage: Z.dimension()
6
sage: Z.homology()                                                          #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: Z x Z, 4: Z, 5: Z, 6: Z}
```

If the associated simplicial complex is an $n$-simplex, then the corresponding moment-angle complex is a polydisc (a complex ball) of complex dimension $n + 1$:

```
sage: Z = MomentAngleComplex([[0, 1, 2]]); Z
Moment-angle complex of Simplicial complex with vertex set (0, 1, 2)
and facets {(0, 1, 2)}
```

This can be seen by viewing the components used in the construction of this moment-angle complex by calling *components()*:

```
sage: Z.components()
{(0, 1, 2): [The 2-simplex, The 2-simplex, The 2-simplex]}
```

If the associated simplicial complex is a disjoint union of 2 points, then the corresponding moment-angle complex is homeomorphic to a boundary of a 3-sphere:

```
sage: Z = MomentAngleComplex([[0], [1]]); Z
Moment-angle complex of Simplicial complex with vertex set
(0, 1) and facets {(0,), (1,)}
sage: dict(sorted(Z.components().items()))
{(0,): [The 2-simplex, Minimal triangulation of the 1-sphere],
 (1,): [Minimal triangulation of the 1-sphere, The 2-simplex]}
```

The moment-angle complex passes all the tests of the test suite relative to its category:

```
sage: TestSuite(Z).run()
```

**betti** (*dim=None*)

Return the Betti number (or numbers) of `self`.

The the $i$-th Betti number is the rank of the $i$-th homology group.

INPUT:

- `dim` – (optional) an integer or a list of integers

OUTPUT:

If `dim` is an integer or a list of integers, then return a dictionary of Betti numbers for each given dimension, indexed by dimension. Otherwise, return all Betti numbers.

EXAMPLES:

```
sage: # needs sage.modules
sage: Z = MomentAngleComplex([[0,1], [1,2], [2,0], [1,2,3]])
sage: Z.betti()
{0: 1, 1: 0, 2: 0, 3: 1, 4: 0, 5: 1, 6: 1, 7: 0}
sage: Z = MomentAngleComplex([[0,1], [1,2], [2,0], [1,2,3], [3,0]])
sage: Z.betti(dim=6)
{6: 2}
```

**cohomology**(*dim=None*, *base_ring=Integer Ring*, *algorithm='pari'*, *verbose=False*, *reduced=True*)

The reduced cohomology of `self`.

This is equivalent to calling the `homology()` method, with `cohomology=True` as an argument.

**See also**

*homology()*.

EXAMPLES:

```
sage: X = SimplicialComplex([[0,1],[1,2],[2,3],[3,0]])
sage: Z = MomentAngleComplex(X)
```

It is known that the previous moment-angle complex is homeomorphic to a product of two 3-spheres (which can be seen by looking at the output of `components()`):

```
sage: # needs sage.modules
sage: S3 = simplicial_complexes.Sphere(3)
sage: product_of_spheres = S3.product(S3)
sage: Z.cohomology()
{0: 0, 1: 0, 2: 0, 3: Z x Z, 4: 0, 5: 0, 6: Z}
sage: Z.cohomology() == product_of_spheres.cohomology()  # long time
True
```

**components**()

Return the dictionary of components of `self`, indexed by facets of the associated simplicial complex.

OUTPUT:

A dictionary, whose values are lists, representing spheres and disks described in the construction of the moment-angle complex. `The 2-simplex` represents a 2-disk, and `Minimal triangulation of the 1-sphere` represents a 1-sphere.

EXAMPLES:

```
sage: M = MomentAngleComplex([[0, 1, 2]]); M
Moment-angle complex of Simplicial complex with vertex set
(0, 1, 2) and facets {(0, 1, 2)}
sage: M.components()
{(0, 1, 2): [The 2-simplex, The 2-simplex, The 2-simplex]}
sage: Z = MomentAngleComplex([[0], [1]]); Z
Moment-angle complex of Simplicial complex with vertex set
(0, 1) and facets {(0,), (1,)}
sage: sorted(Z.components().items())
[((0,), [The 2-simplex, Minimal triangulation of the 1-sphere]),
 ((1,), [Minimal triangulation of the 1-sphere, The 2-simplex])]
```

We interpret the output of this method by taking the product of all the elements in each list, and then taking the union of all products. From the previous example, we have $\mathcal{Z} = S^1 \times D^2 \cup D^2 \times S^1 = \partial(D^2 \times D^2) = \partial D^4 = S^3$:

```
sage: Z = MomentAngleComplex([[0,1], [1,2], [2,3], [3,0]])
sage: sorted(Z.components().items())
[((0, 1),
  [The 2-simplex,
   The 2-simplex,
   Minimal triangulation of the 1-sphere,
   Minimal triangulation of the 1-sphere]),
 ((0, 3),
  [The 2-simplex,
   Minimal triangulation of the 1-sphere,
   Minimal triangulation of the 1-sphere,
   The 2-simplex]),
 ((1, 2),
  [Minimal triangulation of the 1-sphere,
   The 2-simplex,
   The 2-simplex,
   Minimal triangulation of the 1-sphere]),
 ((2, 3),
  [Minimal triangulation of the 1-sphere,
   Minimal triangulation of the 1-sphere,
   The 2-simplex,
   The 2-simplex])]
```

It is not that difficult to prove that the previous moment-angle complex is homeomorphic to a product of two 3-spheres. We can look at the cohomologies to try and validate whether this makes sense:

```
sage: S3 = simplicial_complexes.Sphere(3)
sage: product_of_spheres = S3.product(S3)
sage: Z.cohomology()                                                    #␣
↪needs sage.modules
{0: 0, 1: 0, 2: 0, 3: Z x Z, 4: 0, 5: 0, 6: Z}
sage: Z.cohomology() == product_of_spheres.cohomology()  # long time    #␣
↪needs sage.modules
True
```

**cubical_complex**()

Return the cubical complex that represents `self`.

This method returns returns a cubical complex which is derived by explicitly computing products and unions in the definition of a moment-angle complex.

**Warning**

The construction can be very slow, it is not recommended unless the corresponding simplicial complex has 5 or less vertices.

EXAMPLES:

```
sage: Z = MomentAngleComplex([[0,1,2], [1,3]]); Z
Moment-angle complex of Simplicial complex with vertex set
(0, 1, 2, 3) and facets {(1, 3), (0, 1, 2)}
sage: Z.cubical_complex()
```

(continues on next page)

(continued from previous page)

```
Cubical complex with 256 vertices and 6409 cubes
sage: dim(Z.cubical_complex()) == dim(Z)
True
sage: Z = MomentAngleComplex([[0,1], [1,2], [2,0], [1,3]]); Z
Moment-angle complex of Simplicial complex with vertex set
(0, 1, 2, 3) and facets {(0, 1), (0, 2), (1, 2), (1, 3)}
sage: Z.betti() == Z.cubical_complex().betti()  # long time
True
```

We can now work with moment-angle complexes as concrete cubical complexes. Though, it can be very slow, due to the size of the complex. However, for some smaller moment-angle complexes, this may be possible:

```
sage: Z = MomentAngleComplex([[0], [1]]); Z
Moment-angle complex of Simplicial complex with vertex set
(0, 1) and facets {(0,), (1,)}
sage: Z.cubical_complex().f_vector()
[1, 16, 32, 24, 8]
```

**dimension**()

The dimension of `self`.

The dimension of a moment-angle complex is the dimension of the constructed (cubical) complex. It is not difficult to see that this turns out to be $m + n + 1$, where $m$ is the number of vertices and $n$ is the dimension of the associated simplicial complex.

EXAMPLES:

```
sage: Z = MomentAngleComplex([[0,1], [1,2,3]])
sage: Z.dimension()
7
sage: Z = MomentAngleComplex([[0, 1, 2]])
sage: Z.dimension()
6
sage: dim(Z)
6
```

We can construct the cubical complex and compare whether the dimensions coincide:

```
sage: dim(Z) == dim(Z.cubical_complex())
True
```

**euler_characteristic**()

Return the Euler characteristic of `self`.

The Euler characteristic is defined as the alternating sum of the Betti numbers of `self`.

The Euler characteristic of a moment-angle complex is 0 if the associated simplicial complex is not a simplex.

EXAMPLES:

```
sage: # needs sage.modules
sage: X = SimplicialComplex([[0,1,2,3,4,5], [0,1,2,3,4,6],
....:                        [0,1,2,3,5,7], [0,1,2,3,6,8,9]])
sage: M = MomentAngleComplex(X)
sage: M.euler_characteristic()  # long time
0
sage: Z = MomentAngleComplex([[0,1,2,3,4]])
```

(continues on next page)

(continued from previous page)

```
sage: Z.euler_characteristic()
1
```

**has_trivial_lowest_deg_massey_product**()

Return whether `self` has a non-trivial lowest degree triple Massey product.

This is the Massey product in the cohomology of this moment-angle complex. This relies on the theorem which was proven in [GL2019].

ALGORITHM:

We obtain the one-skeleton from the associated simplicial complex, which we consider to be a graph. We then perform `subgraph_search`, searching for any subgraph isomorphic to one of the 8 obstruction graphs listed in the mentioned paper.

EXAMPLES:

A simplex will not have a trivial triple lowest-degree Massey product, because its one-skeleton certainly does contain a subcomplex isomorphic to one of the 8 mentioned in the paper:

```
sage: Z = MomentAngleComplex([[1,2,3,4,5,6]])
sage: Z.has_trivial_lowest_deg_massey_product()
False
```

The following is one of the 8 obstruction graphs:

```
sage: Z = MomentAngleComplex([[1, 2], [1, 4], [2, 3], [3, 5],
....:                         [5, 6], [4, 5], [1, 6]])
sage: Z.has_trivial_lowest_deg_massey_product()
False
```

A hexagon is not isomorphic to any of the 8 obstruction graphs:

```
sage: Z = MomentAngleComplex([[0,1], [1,2], [2,3],
....:                         [3,4], [4,5], [5,0]])
sage: Z.has_trivial_lowest_deg_massey_product()
True
```

**homology**(*dim=None*, *base_ring=Integer Ring*, *cohomology=False*, *algorithm='pari'*, *verbose=False*, *reduced=True*)

The (reduced) homology of `self`.

INPUT:

- `dim` – integer or a list of integers; represents the homology (or homologies) we want to compute
- `base_ring` – commutative ring (default: `ZZ`); must be `ZZ` or a field
- `cohomology` – boolean (default: `False`); if `True`, compute cohomology rather than homology
- `algorithm` – string (default: `'pari'`); the options are `'auto'`, `'dhsw'`, or `'pari'`; see *cell_complex.GenericCellComplex.homology()* documentation for a description of what they mean
- `verbose` – boolean (default: `False`); if `True`, print some messages as the homology is computed
- `reduced` – boolean (default: `True`); if `True`, return the reduced homology

ALGORITHM:

This algorithm is adopted from Theorem 4.5.8 of [BP2014].

The (co)homology of the moment-angle complex is closely related to the (co)homologies of certain full subcomplexes of the associated simplicial complex. More specifically, we know that:

$$H_l(\mathcal{Z}_\mathcal{K}) \cong \bigoplus_{J \subseteq [m]} \widetilde{H}_{l-|J|-1}(\mathcal{K}_J),$$

where $\mathcal{Z}_\mathcal{K}$ denotes the moment-angle complex associated to a simplicial complex $\mathcal{K}$, on the set of vertices $\{1, 2, 3, \ldots, m\} =: [m]$. $\mathcal{K}_J$ denotes the full subcomplex of $\mathcal{K}$, generated by a set of vertices $J$. The same formula holds true for cohomology groups as well.

**See also**

*cell_complex.GenericCellComplex.homology()*

EXAMPLES:

```
 sage: # needs sage.modules
 sage: Z = MomentAngleComplex([[0,1,2], [1,2,3], [3,0]]); Z
 Moment-angle complex of Simplicial complex with vertex set
 (0, 1, 2, 3) and facets {(0, 3), (0, 1, 2), (1, 2, 3)}
 sage: Z = MomentAngleComplex([[0,1,2], [1,2,3], [3,0]])
 sage: Z.homology()
 {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: Z x Z, 6: Z, 7: 0}
 sage: Z.homology(base_ring=GF(2))
 {0: Vector space of dimension 0 over Finite Field of size 2,
  1: Vector space of dimension 0 over Finite Field of size 2,
  2: Vector space of dimension 0 over Finite Field of size 2,
  3: Vector space of dimension 0 over Finite Field of size 2,
  4: Vector space of dimension 0 over Finite Field of size 2,
  5: Vector space of dimension 2 over Finite Field of size 2,
  6: Vector space of dimension 1 over Finite Field of size 2,
  7: Vector space of dimension 0 over Finite Field of size 2}
sage: RP = simplicial_complexes.RealProjectivePlane()
sage: Z = MomentAngleComplex(RP)
sage: Z.homology()
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: Z^10, 6: Z^15, 7: Z^6, 8: C2, 9: 0}
```

This yields the same result as creating a cubical complex from this moment-angle complex, and then computing its (co)homology, but that is incomparably slower and is really only possible when the associated simplicial complex is very small:

```
sage: Z = MomentAngleComplex([[0,1], [1,2], [2,0]]); Z
Moment-angle complex of Simplicial complex with vertex set
(0, 1, 2) and facets {(0, 1), (0, 2), (1, 2)}
sage: Z.cubical_complex()
Cubical complex with 64 vertices and 729 cubes
sage: Z.cubical_complex().homology() == Z.homology()                    #␣
↪needs sage.modules
True
```

Meanwhile, the homology computation used here is quite efficient and works well even with significantly larger underlying simplicial complexes:

```
sage: # needs sage.modules
sage: Z = MomentAngleComplex([[0,1,2,3,4,5], [0,1,2,3,4,6],
....:                         [0,1,2,3,5,7], [0,1,2,3,6,8,9]])
```

(continues on next page)

(continued from previous page)

```
sage: Z.homology()  # long time
{0: 0,
 1: 0,
 2: 0,
 3: Z^9,
 4: Z^17,
 5: Z^12,
 6: Z x Z x Z,
 7: 0,
 8: 0,
 9: 0,
 10: 0,
 11: 0,
 12: 0,
 13: 0,
 14: 0,
 15: 0,
 16: 0,
 17: 0}
sage: Z = MomentAngleComplex([[0,1,2,3], [0,1,2,4], [0,1,3,5],
....:                         [0,1,4,5], [0,2,3,6], [0,2,4,6]])
sage: Z.homology(dim=range(5), reduced=True)
{0: 0, 1: 0, 2: 0, 3: Z x Z x Z x Z, 4: Z x Z}
sage: Z.homology(dim=range(5), reduced=False)
{0: Z, 1: 0, 2: 0, 3: Z x Z x Z x Z, 4: Z x Z}
sage: all(Z.homology(i,reduced=True) == Z.homology(i,reduced=False)
....:     for i in range(1, dim(Z)))
True
sage: all(Z.homology(i,reduced=True) == Z.homology(i,reduced=False)
....:     for i in range(dim(Z)))
False
```

**product**(*other*)

Return the product of `self` with `other`.

It is known that the product of two moment-angle complexes is a moment-angle complex over the join of the two corresponding simplicial complexes. This result can be found on page 138 of [BP2014].

OUTPUT: a moment-angle complex which is the product of the parsed moment-angle complexes

EXAMPLES:

```
sage: X = SimplicialComplex([[0,1,2,3], [1,4], [3,2,4]])
sage: Y = SimplicialComplex([[1,2,3],[1,2,4],[3,5],[4,5]])
sage: Z = MomentAngleComplex(X)
sage: M = MomentAngleComplex(Y)
sage: Z.product(M)
Moment-angle complex of Simplicial complex with
10 vertices and 12 facets
sage: Z.product(M) == MomentAngleComplex(X*Y)
True
```

**simplicial_complex**()

Return the simplicial complex that defines `self`.

EXAMPLES:

```
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: Z = MomentAngleComplex(RP2)
sage: Z.simplicial_complex()
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 10 facets
sage: Z = MomentAngleComplex([[0], [1], [2]])
sage: Z.simplicial_complex()
Simplicial complex with vertex set (0, 1, 2)
and facets {(0,), (1,), (2,)}
```

CHAPTER

FIFTEEN

# INDICES AND TABLES

- Index
- Module Index
- Search Page

# BIBLIOGRAPHY

[ABS96] Amos Altshule, Jürgen Bokowski and Peter Schuchert, *Neighborly 2-Manifolds with 12 Vertices*, Journal of Combinatorial Theory, Series A, 75, 148-162 (1996), doi:10.1006/jcta.1996.0069

# PYTHON MODULE INDEX

## t


sage.topology.cell_complex, 199
sage.topology.cubical_complex, 93
sage.topology.delta_complex, 77
sage.topology.filtered_simplicial_complex, 213
sage.topology.moment_angle_complex, 217
sage.topology.simplicial_complex, 3
sage.topology.simplicial_complex_examples, 61
sage.topology.simplicial_complex_homset, 57
sage.topology.simplicial_complex_morphism, 47
sage.topology.simplicial_set, 109
sage.topology.simplicial_set_catalog, 183
sage.topology.simplicial_set_constructions, 149
sage.topology.simplicial_set_examples, 175
sage.topology.simplicial_set_morphism, 185

# INDEX

## A


AbstractSimplex() (*in module sage.topology.simplicial_set*), 113
AbstractSimplex_class (*class in sage.topology.simplicial_set*), 114
add_face() (*sage.topology.simplicial_complex.SimplicialComplex method*), 11
alexander_dual() (*sage.topology.simplicial_complex.SimplicialComplex method*), 11
alexander_whitney() (*sage.topology.cell_complex.GenericCellComplex method*), 199
alexander_whitney() (*sage.topology.cubical_complex.Cube method*), 94
alexander_whitney() (*sage.topology.cubical_complex.CubicalComplex method*), 99
alexander_whitney() (*sage.topology.delta_complex.DeltaComplex method*), 80
alexander_whitney() (*sage.topology.simplicial_complex.Simplex method*), 6
alexander_whitney() (*sage.topology.simplicial_complex.SimplicialComplex method*), 12
alexander_whitney() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 118
algebraic_topological_model() (*sage.topology.cell_complex.GenericCellComplex method*), 200
algebraic_topological_model() (*sage.topology.cubical_complex.CubicalComplex method*), 99
algebraic_topological_model() (*sage.topology.delta_complex.DeltaComplex method*), 81
algebraic_topological_model() (*sage.topology.simplicial_complex.SimplicialComplex method*), 12
algebraic_topological_model() (*sage.topology.simplicial_set.SimplicialSet_finite method*), 142
all_degeneracies() (*in module sage.topology.simplicial_set*), 145
all_n_simplices() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 118
ambient() (*sage.topology.simplicial_set_constructions.QuotientOfSimplicialSet method*), 166
ambient_space() (*sage.topology.simplicial_set_constructions.SubSimplicialSet method*), 170
an_element() (*sage.topology.simplicial_complex_homset.SimplicialComplexHomset method*), 57
an_element() (*sage.topology.simplicial_set_morphism.SimplicialSetHomset method*), 185
apply_degeneracies() (*sage.topology.simplicial_set.AbstractSimplex_class method*), 115
associated_chain_complex_morphism() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 48
associated_chain_complex_morphism() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 189
automorphism_group() (*sage.topology.simplicial_complex.SimplicialComplex method*), 13


## B


BarnetteSphere() (*in module sage.topology.simplicial_complex_examples*), 62
barycentric_subdivision() (*sage.topology.delta_complex.DeltaComplex method*), 81
barycentric_subdivision() (*sage.topology.simplicial_complex.SimplicialComplex method*), 14
base_as_subset() (*sage.topology.simplicial_set_constructions.ConeOfSimplicialSet_finite method*), 151
betti() (*sage.topology.cell_complex.GenericCellComplex method*), 200
betti() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 218
betti() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 119
betti_number() (*sage.topology.filtered_simplicial_complex.FilteredSimplicialComplex method*), 214
bigraded_betti_number() (*sage.topology.sim-*

*plicial_complex.SimplicialComplex method*), 14
bigraded_betti_numbers() (*sage.topology.simplicial_complex.SimplicialComplex method*), 15
BrucknerGrunbaumSphere() (*in module sage.topology.simplicial_complex_examples*), 62

## C

cartesian_product() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 119
cells() (*sage.topology.cell_complex.GenericCellComplex method*), 201
cells() (*sage.topology.cubical_complex.CubicalComplex method*), 100
cells() (*sage.topology.delta_complex.DeltaComplex method*), 82
cells() (*sage.topology.simplicial_complex.SimplicialComplex method*), 16
cells() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 121
chain_complex() (*sage.topology.cell_complex.GenericCellComplex method*), 201
chain_complex() (*sage.topology.cubical_complex.CubicalComplex method*), 100
chain_complex() (*sage.topology.delta_complex.DeltaComplex method*), 82
chain_complex() (*sage.topology.simplicial_complex.SimplicialComplex method*), 16
chain_complex() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 122
chain_complex() (*sage.topology.simplicial_set.SimplicialSet_finite method*), 142
ChessboardComplex() (*in module sage.topology.simplicial_complex_examples*), 62
ClassifyingSpace() (*in module sage.topology.simplicial_set_examples*), 175
coequalizer() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 189
cohomology() (*sage.topology.cell_complex.GenericCellComplex method*), 202
cohomology() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 219
cohomology() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 122
cohomology_ring() (*sage.topology.cell_complex.GenericCellComplex method*), 203
ComplexProjectivePlane() (*in module sage.topology.simplicial_complex_examples*), 63
ComplexProjectiveSpace() (*in module sage.topology.simplicial_set_examples*), 175
components() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 219
cone() (*sage.topology.cubical_complex.CubicalComplex method*), 101
cone() (*sage.topology.delta_complex.DeltaComplex method*), 83
cone() (*sage.topology.simplicial_complex.SimplicialComplex method*), 17
cone() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 123
cone_vertices() (*sage.topology.simplicial_complex.SimplicialComplex method*), 17
ConeOfSimplicialSet (*class in sage.topology.simplicial_set_constructions*), 149
ConeOfSimplicialSet_finite (*class in sage.topology.simplicial_set_constructions*), 150
connected_component() (*sage.topology.simplicial_complex.SimplicialComplex method*), 18
connected_sum() (*sage.topology.cubical_complex.CubicalComplex method*), 102
connected_sum() (*sage.topology.delta_complex.DeltaComplex method*), 83
connected_sum() (*sage.topology.simplicial_complex.SimplicialComplex method*), 18
constant_map() (*sage.topology.simplicial_set_morphism.SimplicialSetHomset method*), 186
constant_map() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 124
coproduct() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 190
coproduct() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 124
Cube (*class in sage.topology.cubical_complex*), 94
Cube() (*sage.topology.cubical_complex.CubicalComplexExamples method*), 106
cubical_complex() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 220
CubicalComplex (*class in sage.topology.cubical_complex*), 97
CubicalComplexExamples (*class in sage.topology.cubical_complex*), 106

## D

decone() (*sage.topology.simplicial_complex.SimplicialComplex method*), 19
defining_map() (*sage.topology.simplicial_set_constructions.PullbackOfSimplicialSets method*), 158
defining_map() (*sage.topology.simplicial_set_constructions.PushoutOfSimplicialSets method*), 163
degeneracies() (*sage.topology.simplicial_set.AbstractSimplex_class method*), 115
delta_complex() (*sage.topology.simplicial_complex.SimplicialComplex method*), 19

DeltaComplex (*class in sage.topology.delta_complex*), 77
DeltaComplexExamples (*class in sage.topology.delta_complex*), 89
diagonal_morphism() (*sage.topology.simplicial_complex_homset.SimplicialComplexHomset method*), 58
diagonal_morphism() (*sage.topology.simplicial_set_morphism.SimplicialSetHomset method*), 186
dimension() (*sage.topology.cell_complex.GenericCellComplex method*), 205
dimension() (*sage.topology.cubical_complex.Cube method*), 95
dimension() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 221
dimension() (*sage.topology.simplicial_complex.Simplex method*), 6
dimension() (*sage.topology.simplicial_set.AbstractSimplex_class method*), 115
disjoint_union() (*sage.topology.cell_complex.GenericCellComplex method*), 205
disjoint_union() (*sage.topology.cubical_complex.CubicalComplex method*), 102
disjoint_union() (*sage.topology.delta_complex.DeltaComplex method*), 84
disjoint_union() (*sage.topology.simplicial_complex.SimplicialComplex method*), 20
disjoint_union() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 125
DisjointUnionOfSimplicialSets (*class in sage.topology.simplicial_set_constructions*), 151
DisjointUnionOfSimplicialSets_finite (*class in sage.topology.simplicial_set_constructions*), 153
DunceHat() (*in module sage.topology.simplicial_complex_examples*), 63

## E

elementary_subdivision() (*sage.topology.delta_complex.DeltaComplex method*), 84
Empty() (*in module sage.topology.simplicial_set_examples*), 176
equalizer() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 190
euler_characteristic() (*sage.topology.cell_complex.GenericCellComplex method*), 205
euler_characteristic() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 221
euler_characteristic() (*sage.topology.simplicial_set.SimplicialSet_finite method*), 143

## F

F_triangle() (*sage.topology.simplicial_complex.SimplicialComplex method*), 11
f_triangle() (*sage.topology.simplicial_complex.SimplicialComplex method*), 20
f_vector() (*sage.topology.cell_complex.GenericCellComplex method*), 206
f_vector() (*sage.topology.simplicial_set.SimplicialSet_finite method*), 143
face() (*sage.topology.cubical_complex.Cube method*), 95
face() (*sage.topology.simplicial_complex.Simplex method*), 7
face() (*sage.topology.simplicial_complex.SimplicialComplex method*), 20
face() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 126
face_data() (*sage.topology.simplicial_set.SimplicialSet_finite method*), 144
face_degeneracies() (*in module sage.topology.simplicial_set*), 145
face_iterator() (*sage.topology.simplicial_complex.SimplicialComplex method*), 21
face_poset() (*sage.topology.cell_complex.GenericCellComplex method*), 206
face_poset() (*sage.topology.delta_complex.DeltaComplex method*), 85
faces() (*sage.topology.cubical_complex.Cube method*), 95
faces() (*sage.topology.simplicial_complex.Simplex method*), 7
faces() (*sage.topology.simplicial_complex.SimplicialComplex method*), 21
faces() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 126
faces_as_pairs() (*sage.topology.cubical_complex.Cube method*), 96
facets() (*sage.topology.simplicial_complex.SimplicialComplex method*), 21
facets_for_K3() (*in module sage.topology.simplicial_complex*), 43
facets_for_K3() (*in module sage.topology.simplicial_complex_examples*), 75
facets_for_RP4() (*in module sage.topology.simplicial_complex*), 43
facets_for_RP4() (*in module sage.topology.simplicial_complex_examples*), 75
factor() (*sage.topology.simplicial_set_constructions.Factors method*), 153
factor() (*sage.topology.simplicial_set_constructions.ProductOfSimplicialSets method*), 155
Factors (*class in sage.topology.simplicial_set_constructions*), 153
factors() (*sage.topology.simplicial_set_construc-*

*tions.Factors method*), 154
FareyMap() (*in module sage.topology.simplicial_complex_examples*), 64
fat_wedge_as_subset() (*sage.topology.simplicial_set_constructions.ProductOfSimplicialSets_finite method*), 156
fiber_product() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 50
FilteredSimplicialComplex (*class in sage.topology.filtered_simplicial_complex*), 214
filtration() (*sage.topology.filtered_simplicial_complex.FilteredSimplicialComplex method*), 215
fixed_complex() (*sage.topology.simplicial_complex.SimplicialComplex method*), 22
flip_graph() (*sage.topology.simplicial_complex.SimplicialComplex method*), 22
fundamental_group() (*sage.topology.simplicial_complex.SimplicialComplex method*), 23

## G

g_vector() (*sage.topology.simplicial_complex.SimplicialComplex method*), 24
generated_subcomplex() (*sage.topology.simplicial_complex.SimplicialComplex method*), 25
GenericCellComplex (*class in sage.topology.cell_complex*), 199
GenusSix() (*in module sage.topology.simplicial_complex_examples*), 64
graph() (*sage.topology.cell_complex.GenericCellComplex method*), 206
graph() (*sage.topology.cubical_complex.CubicalComplex method*), 102
graph() (*sage.topology.delta_complex.DeltaComplex method*), 85
graph() (*sage.topology.simplicial_complex.SimplicialComplex method*), 25
graph() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 127

## H

h_triangle() (*sage.topology.simplicial_complex.SimplicialComplex method*), 25
h_vector() (*sage.topology.simplicial_complex.SimplicialComplex method*), 26
has_trivial_lowest_deg_massey_product() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 222
homology() (*sage.topology.cell_complex.GenericCellComplex method*), 206
homology() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 222
homology() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 127
homology_with_basis() (*sage.topology.cell_complex.GenericCellComplex method*), 208
HopfMap() (*in module sage.topology.simplicial_set_examples*), 176
Horn() (*in module sage.topology.simplicial_set_examples*), 177

## I

identity() (*sage.topology.simplicial_complex_homset.SimplicialComplexHomset method*), 58
identity() (*sage.topology.simplicial_set_morphism.SimplicialSetHomset method*), 187
identity() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 128
image() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 50
image() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 191
inclusion_map() (*sage.topology.simplicial_set_constructions.DisjointUnionOfSimplicialSets_finite method*), 153
inclusion_map() (*sage.topology.simplicial_set_constructions.SubSimplicialSet method*), 170
inclusion_map() (*sage.topology.simplicial_set_constructions.WedgeOfSimplicialSets_finite method*), 173
induced_homology_morphism() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 51
induced_homology_morphism() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 191
insert() (*sage.topology.filtered_simplicial_complex.FilteredSimplicialComplex method*), 215
intersection() (*sage.topology.simplicial_complex.SimplicialComplex method*), 26
is_acyclic() (*sage.topology.cell_complex.GenericCellComplex method*), 209
is_balanced() (*sage.topology.simplicial_complex.SimplicialComplex method*), 26
is_bijective() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 192
is_cohen_macaulay() (*sage.topology.simplicial_complex.SimplicialComplex method*), 27
is_connected() (*sage.topology.cell_complex.GenericCellComplex method*), 209
is_connected() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 129
is_constant() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 192

is_contiguous_to() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 52
is_degenerate() (*sage.topology.simplicial_set.AbstractSimplex_class method*), 116
is_empty() (*sage.topology.simplicial_complex.Simplex method*), 7
is_face() (*sage.topology.cubical_complex.Cube method*), 96
is_face() (*sage.topology.simplicial_complex.Simplex method*), 7
is_flag_complex() (*sage.topology.simplicial_complex.SimplicialComplex method*), 28
is_golod() (*sage.topology.simplicial_complex.SimplicialComplex method*), 28
is_identity() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 53
is_identity() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 193
is_immutable() (*sage.topology.simplicial_complex.SimplicialComplex method*), 28
is_injective() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 53
is_injective() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 194
is_isomorphic() (*sage.topology.simplicial_complex.SimplicialComplex method*), 29
is_minimally_non_golod() (*sage.topology.simplicial_complex.SimplicialComplex method*), 29
is_mutable() (*sage.topology.simplicial_complex.SimplicialComplex method*), 30
is_nondegenerate() (*sage.topology.simplicial_set.AbstractSimplex_class method*), 116
is_partitionable() (*sage.topology.simplicial_complex.SimplicialComplex method*), 30
is_pointed() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 194
is_pseudomanifold() (*sage.topology.simplicial_complex.SimplicialComplex method*), 31
is_pure() (*sage.topology.cubical_complex.CubicalComplex method*), 103
is_pure() (*sage.topology.simplicial_complex.SimplicialComplex method*), 32
is_reduced() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 129
is_shellable() (*sage.topology.simplicial_complex.SimplicialComplex method*), 32
is_shelling_order() (*sage.topology.simplicial_complex.SimplicialComplex method*), 33
is_SimplicialComplexHomset() (*in module sage.topology.simplicial_complex_homset*), 58
is_SimplicialComplexMorphism() (*in module sage.topology.simplicial_complex_morphism*), 55
is_subcomplex() (*sage.topology.cubical_complex.CubicalComplex method*), 103
is_subcomplex() (*sage.topology.simplicial_complex.SimplicialComplex method*), 34
is_surjective() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 54
is_surjective() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 194

## J

join() (*sage.topology.cell_complex.GenericCellComplex method*), 210
join() (*sage.topology.cubical_complex.CubicalComplex method*), 104
join() (*sage.topology.delta_complex.DeltaComplex method*), 85
join() (*sage.topology.simplicial_complex.Simplex method*), 7
join() (*sage.topology.simplicial_complex.SimplicialComplex method*), 34
join() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 129

## K

K3Surface() (*in module sage.topology.simplicial_complex_examples*), 64
KleinBottle() (*in module sage.topology.simplicial_complex_examples*), 65
KleinBottle() (*in module sage.topology.simplicial_set_examples*), 177
KleinBottle() (*sage.topology.cubical_complex.CubicalComplexExamples method*), 106
KleinBottle() (*sage.topology.delta_complex.DeltaComplexExamples method*), 89

## L

lattice_paths() (*in module sage.topology.simplicial_complex*), 44
link() (*sage.topology.simplicial_complex.SimplicialComplex method*), 35

## M

map_from_base() (*sage.topology.simplicial_set_constructions.ConeOfSimplicialSet_finite method*), 151
map_from_base() (*sage.topology.simplicial_set_constructions.ReducedConeOfSimplicialSet_finite method*), 168
mapping_cone() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 195

mapping_torus() (*sage.topology.simplicial_complex_morphism.SimplicialComplexMorphism method*), 54
matching() (*in module sage.topology.simplicial_complex_examples*), 75
MatchingComplex() (*in module sage.topology.simplicial_complex_examples*), 65
maximal_cells() (*sage.topology.cubical_complex.CubicalComplex method*), 104
maximal_faces() (*sage.topology.simplicial_complex.SimplicialComplex method*), 35
minimal_nonfaces() (*sage.topology.simplicial_complex.SimplicialComplex method*), 36
module
    sage.topology.cell_complex, 199
    sage.topology.cubical_complex, 93
    sage.topology.delta_complex, 77
    sage.topology.filtered_simplicial_complex, 213
    sage.topology.moment_angle_complex, 217
    sage.topology.simplicial_complex, 3
    sage.topology.simplicial_complex_examples, 61
    sage.topology.simplicial_complex_homset, 57
    sage.topology.simplicial_complex_morphism, 47
    sage.topology.simplicial_set, 109
    sage.topology.simplicial_set_catalog, 183
    sage.topology.simplicial_set_constructions, 149
    sage.topology.simplicial_set_examples, 175
    sage.topology.simplicial_set_morphism, 185
moment_angle_complex() (*sage.topology.simplicial_complex.SimplicialComplex method*), 36
MomentAngleComplex (*class in sage.topology.moment_angle_complex*), 217
MooreSpace() (*in module sage.topology.simplicial_complex_examples*), 65

## N

n_cells() (*sage.topology.cell_complex.GenericCellComplex method*), 210
n_cells() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 129
n_chains() (*sage.topology.cell_complex.GenericCellComplex method*), 210
n_chains() (*sage.topology.delta_complex.DeltaComplex method*), 86
n_chains() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 130
n_cubes() (*sage.topology.cubical_complex.CubicalComplex method*), 104
n_faces() (*sage.topology.simplicial_complex.SimplicialComplex method*), 36
n_skeleton() (*sage.topology.cell_complex.GenericCellComplex method*), 211
n_skeleton() (*sage.topology.cubical_complex.CubicalComplex method*), 105
n_skeleton() (*sage.topology.delta_complex.DeltaComplex method*), 87
n_skeleton() (*sage.topology.simplicial_complex.SimplicialComplex method*), 37
n_skeleton() (*sage.topology.simplicial_set_constructions.ConeOfSimplicialSet method*), 150
n_skeleton() (*sage.topology.simplicial_set_constructions.DisjointUnionOfSimplicialSets method*), 152
n_skeleton() (*sage.topology.simplicial_set_constructions.ProductOfSimplicialSets method*), 156
n_skeleton() (*sage.topology.simplicial_set_constructions.PullbackOfSimplicialSets method*), 158
n_skeleton() (*sage.topology.simplicial_set_constructions.PushoutOfSimplicialSets method*), 163
n_skeleton() (*sage.topology.simplicial_set_constructions.QuotientOfSimplicialSet method*), 166
n_skeleton() (*sage.topology.simplicial_set_constructions.ReducedConeOfSimplicialSet method*), 168
n_skeleton() (*sage.topology.simplicial_set_constructions.SuspensionOfSimplicialSet method*), 171
n_skeleton() (*sage.topology.simplicial_set_examples.Nerve method*), 178
n_skeleton() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 195
n_skeleton() (*sage.topology.simplicial_set.SimplicialSet_finite method*), 144
Nerve (*class in sage.topology.simplicial_set_examples*), 178
nondegenerate() (*sage.topology.simplicial_set.AbstractSimplex_class method*), 116
nondegenerate_intervals() (*sage.topology.cubical_complex.Cube method*), 96
nondegenerate_simplices() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 130
NonDegenerateSimplex (*class in sage.topology.simplicial_set*), 116
NotIConnectedGraphs() (*in module sage.topology.simplicial_complex_examples*), 66

## P


persistence_intervals() (*sage.topology.filtered_simplicial_complex.FilteredSimplicialComplex method*), 216
PoincareHomologyThreeSphere() (*in module sage.topology.simplicial_complex_examples*), 66
Point() (*in module sage.topology.simplicial_set_examples*), 178
PresentationComplex() (*in module sage.topology.simplicial_set_examples*), 179
product() (*sage.topology.cell_complex.GenericCellComplex method*), 211
product() (*sage.topology.cubical_complex.Cube method*), 96
product() (*sage.topology.cubical_complex.CubicalComplex method*), 105
product() (*sage.topology.delta_complex.DeltaComplex method*), 87
product() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 224
product() (*sage.topology.simplicial_complex.Simplex method*), 8
product() (*sage.topology.simplicial_complex.SimplicialComplex method*), 37
product() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 196
product() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 131
ProductOfSimplicialSets (*class in sage.topology.simplicial_set_constructions*), 154
ProductOfSimplicialSets_finite (*class in sage.topology.simplicial_set_constructions*), 156
projection_map() (*sage.topology.simplicial_set_constructions.ProductOfSimplicialSets_finite method*), 157
projection_map() (*sage.topology.simplicial_set_constructions.PullbackOfSimplicialSets_finite method*), 159
projection_map() (*sage.topology.simplicial_set_constructions.WedgeOfSimplicialSets_finite method*), 173
ProjectivePlane() (*in module sage.topology.simplicial_complex_examples*), 67
prune() (*sage.topology.filtered_simplicial_complex.FilteredSimplicialComplex method*), 216
pullback() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 196
pullback() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 133
PullbackOfSimplicialSets (*class in sage.topology.simplicial_set_constructions*), 157
PullbackOfSimplicialSets_finite (*class in sage.topology.simplicial_set_constructions*), 159
pushout() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 197
pushout() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 134
PushoutOfSimplicialSets (*class in sage.topology.simplicial_set_constructions*), 161
PushoutOfSimplicialSets_finite (*class in sage.topology.simplicial_set_constructions*), 164


## Q


QuaternionicProjectivePlane() (*in module sage.topology.simplicial_complex_examples*), 67
quotient() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 136
quotient_map() (*sage.topology.simplicial_set_constructions.QuotientOfSimplicialSet_finite method*), 167
QuotientOfSimplicialSet (*class in sage.topology.simplicial_set_constructions*), 165
QuotientOfSimplicialSet_finite (*class in sage.topology.simplicial_set_constructions*), 167


## R


RandomComplex() (*in module sage.topology.simplicial_complex_examples*), 68
RandomTwoSphere() (*in module sage.topology.simplicial_complex_examples*), 68
RealProjectivePlane() (*in module sage.topology.simplicial_complex_examples*), 69
RealProjectivePlane() (*sage.topology.cubical_complex.CubicalComplexExamples method*), 107
RealProjectivePlane() (*sage.topology.delta_complex.DeltaComplexExamples method*), 90
RealProjectiveSpace() (*in module sage.topology.simplicial_complex_examples*), 69
RealProjectiveSpace() (*in module sage.topology.simplicial_set_examples*), 179
reduce() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 137
ReducedConeOfSimplicialSet (*class in sage.topology.simplicial_set_constructions*), 167
ReducedConeOfSimplicialSet_finite (*class in sage.topology.simplicial_set_constructions*), 168
remove_face() (*sage.topology.simplicial_complex.SimplicialComplex method*), 38
remove_faces() (*sage.topology.simplicial_complex.SimplicialComplex method*), 39
rename_latex() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 138
rename_vertex() (*in module sage.topology.simplicial_complex*), 44

restriction_sets() (*sage.topology.simplicial_complex.SimplicialComplex method*), 39
RudinBall() (*in module sage.topology.simplicial_complex_examples*), 70

## S

sage.topology.cell_complex
module, 199
sage.topology.cubical_complex
module, 93
sage.topology.delta_complex
module, 77
sage.topology.filtered_simplicial_complex
module, 213
sage.topology.moment_angle_complex
module, 217
sage.topology.simplicial_complex
module, 3
sage.topology.simplicial_complex_examples
module, 61
sage.topology.simplicial_complex_homset
module, 57
sage.topology.simplicial_complex_morphism
module, 47
sage.topology.simplicial_set
module, 109
sage.topology.simplicial_set_catalog
module, 183
sage.topology.simplicial_set_constructions
module, 149
sage.topology.simplicial_set_examples
module, 175
sage.topology.simplicial_set_morphism
module, 185
set() (*sage.topology.simplicial_complex.Simplex method*), 8
set_immutable() (*sage.topology.simplicial_complex.SimplicialComplex method*), 40
ShiftedComplex() (*in module sage.topology.simplicial_complex_examples*), 71
shrink_simplicial_complex() (*in module sage.topology.simplicial_set*), 146
Simplex (*class in sage.topology.simplicial_complex*), 5
Simplex() (*in module sage.topology.simplicial_complex_examples*), 71
Simplex() (*in module sage.topology.simplicial_set_examples*), 180
Simplex() (*sage.topology.delta_complex.DeltaComplexExamples method*), 90
simplicial_complex() (*sage.topology.moment_angle_complex.MomentAngleComplex method*), 224
simplicial_data_from_kenzo_output() (*in module sage.topology.simplicial_set_examples*), 181
SimplicialComplex (*class in sage.topology.simplicial_complex*), 9
SimplicialComplexHomset (*class in sage.topology.simplicial_complex_homset*), 57
SimplicialComplexMorphism (*class in sage.topology.simplicial_complex_morphism*), 48
SimplicialSet (*in module sage.topology.simplicial_set*), 117
SimplicialSet_arbitrary (*class in sage.topology.simplicial_set*), 117
SimplicialSet_finite (*class in sage.topology.simplicial_set*), 141
SimplicialSetHomset (*class in sage.topology.simplicial_set_morphism*), 185
SimplicialSetMorphism (*class in sage.topology.simplicial_set_morphism*), 187
SmashProductOfSimplicialSets_finite (*class in sage.topology.simplicial_set_constructions*), 169
Sphere() (*in module sage.topology.simplicial_complex_examples*), 71
Sphere() (*in module sage.topology.simplicial_set_examples*), 180
Sphere() (*sage.topology.cubical_complex.CubicalComplexExamples method*), 107
Sphere() (*sage.topology.delta_complex.DeltaComplexExamples method*), 90
standardize_degeneracies() (*in module sage.topology.simplicial_set*), 146
standardize_face_maps() (*in module sage.topology.simplicial_set*), 147
stanley_reisner_ring() (*sage.topology.simplicial_complex.SimplicialComplex method*), 40
star() (*sage.topology.simplicial_complex.SimplicialComplex method*), 40
stellar_subdivision() (*sage.topology.simplicial_complex.SimplicialComplex method*), 41
structure_map() (*sage.topology.simplicial_set_constructions.PullbackOfSimplicialSets_finite method*), 160
structure_map() (*sage.topology.simplicial_set_constructions.PushoutOfSimplicialSets_finite method*), 164
subcomplex() (*sage.topology.delta_complex.DeltaComplex method*), 88
subcomplex() (*sage.topology.simplicial_set_constructions.QuotientOfSimplicialSet method*), 166

subsimplicial_set() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 138
SubSimplicialSet (*class in sage.topology.simplicial_set_constructions*), 169
SumComplex() (*in module sage.topology.simplicial_complex_examples*), 72
summand() (*sage.topology.simplicial_set_constructions.DisjointUnionOfSimplicialSets method*), 152
summand() (*sage.topology.simplicial_set_constructions.WedgeOfSimplicialSets method*), 172
summands() (*sage.topology.simplicial_set_constructions.DisjointUnionOfSimplicialSets method*), 152
summands() (*sage.topology.simplicial_set_constructions.WedgeOfSimplicialSets method*), 173
SurfaceOfGenus() (*in module sage.topology.simplicial_complex_examples*), 73
SurfaceOfGenus() (*sage.topology.cubical_complex.CubicalComplexExamples method*), 107
SurfaceOfGenus() (*sage.topology.delta_complex.DeltaComplexExamples method*), 90
suspension() (*sage.topology.cubical_complex.CubicalComplex method*), 105
suspension() (*sage.topology.delta_complex.DeltaComplex method*), 88
suspension() (*sage.topology.simplicial_complex.SimplicialComplex method*), 42
suspension() (*sage.topology.simplicial_set_morphism.SimplicialSetMorphism method*), 197
suspension() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 139
SuspensionOfSimplicialSet (*class in sage.topology.simplicial_set_constructions*), 170
SuspensionOfSimplicialSet_finite (*class in sage.topology.simplicial_set_constructions*), 171

## T

Torus() (*in module sage.topology.simplicial_complex_examples*), 74
Torus() (*in module sage.topology.simplicial_set_examples*), 180
Torus() (*sage.topology.cubical_complex.CubicalComplexExamples method*), 107
Torus() (*sage.topology.delta_complex.DeltaComplexExamples method*), 91
tuple() (*sage.topology.cubical_complex.Cube method*), 97
tuple() (*sage.topology.simplicial_complex.Simplex method*), 8

## U

UniqueSimplicialComplex (*class in sage.topology.simplicial_complex_examples*), 74
universal_property() (*sage.topology.simplicial_set_constructions.PullbackOfSimplicialSets_finite method*), 160
universal_property() (*sage.topology.simplicial_set_constructions.PushoutOfSimplicialSets_finite method*), 164

## V

vertices() (*sage.topology.simplicial_complex.SimplicialComplex method*), 42

## W

wedge() (*sage.topology.cell_complex.GenericCellComplex method*), 211
wedge() (*sage.topology.cubical_complex.CubicalComplex method*), 105
wedge() (*sage.topology.delta_complex.DeltaComplex method*), 88
wedge() (*sage.topology.simplicial_complex.SimplicialComplex method*), 43
wedge() (*sage.topology.simplicial_set.SimplicialSet_arbitrary method*), 140
wedge_as_subset() (*sage.topology.simplicial_set_constructions.ProductOfSimplicialSets_finite method*), 157
WedgeOfSimplicialSets (*class in sage.topology.simplicial_set_constructions*), 171
WedgeOfSimplicialSets_finite (*class in sage.topology.simplicial_set_constructions*), 173

## Z

ZieglerBall() (*in module sage.topology.simplicial_complex_examples*), 75