Sage Reference Manual: Groups

Release 6.10

The Sage Development Team

December 21, 2015

CONTENTS

CHAPTER

ONE

EXAMPLES OF GROUPS

The `groups` object may be used to access examples of various groups. Using tab-completion on this object is an easy way to discover and quickly create the groups that are available (as listed here).

Let `<tab>` indicate pressing the tab key. So begin by typing `groups.<tab>` to the see primary divisions, followed by (for example) `groups.matrix.<tab>` to access various groups implemented as sets of matrices.

- Permutation Groups (`groups.permutation.<tab>`)
  - `groups.permutation.Symmetric`
  - `groups.permutation.Alternating`
  - `groups.permutation.KleinFour`
  - `groups.permutation.Quaternion`
  - `groups.permutation.Cyclic`
  - `groups.permutation.Dihedral`
  - `groups.permutation.DiCyclic`
  - `groups.permutation.Mathieu`
  - `groups.permutation.Suzuki`
  - `groups.permutation.PGL`
  - `groups.permutation.PSL`
  - `groups.permutation.PSp`
  - `groups.permutation.PSU`
  - `groups.permutation.PGU`
  - `groups.permutation.Transitive`
  - `groups.permutation.RubiksCube`
- Matrix Groups (`groups.matrix.<tab>`)
  - `groups.matrix.QuaternionGF3`
  - `groups.matrix.GL`
  - `groups.matrix.SL`
  - `groups.matrix.Sp`
  - `groups.matrix.GU`
  - `groups.matrix.SU`

  - `groups.matrix.GO`
  - `groups.matrix.SO`
- Finitely Presented Groups (`groups.presentation.<tab>`)
  - `groups.presentation.Alternating`
  - `groups.presentation.Cyclic`
  - `groups.presentation.Dihedral`
  - `groups.presentation.DiCyclic`
  - `groups.presentation.FGAbelian`
  - `groups.presentation.KleinFour`
  - `groups.presentation.Quaternion`
  - `groups.presentation.Symmetric`
- Affine Groups (`groups.affine.<tab>`)
  - `groups.affine.Affine`
  - `groups.affine.Euclidean`
- Miscellaneous Groups (`groups.misc.<tab>`)
  - `groups.misc.AdditiveAbelian`
  - `groups.misc.AdditiveCyclic`
  - `groups.misc.Braid`
  - `groups.misc.CoxeterGroup`
  - `groups.misc.Free`
  - `groups.misc.RightAngledArtin`
  - `groups.misc.SemimonomialTransformation`
  - `groups.misc.WeylGroup`

CHAPTER

TWO

BASE CLASS FOR GROUPS

class `sage.groups.group.AbelianGroup`

Bases: `sage.groups.group.Group`

Generic abelian group.

is_abelian()

Return True.

EXAMPLES:

```
sage: from sage.groups.group import AbelianGroup
sage: G = AbelianGroup()
sage: G.is_abelian()
True
```

class `sage.groups.group.AlgebraicGroup`

Bases: `sage.groups.group.Group`

class `sage.groups.group.FiniteGroup`

Bases: `sage.groups.group.Group`

Generic finite group.

is_finite()

Return True.

EXAMPLES:

```
sage: from sage.groups.group import FiniteGroup
sage: G = FiniteGroup()
sage: G.is_finite()
True
```

class `sage.groups.group.Group`

Bases: `sage.structure.parent.Parent`

Base class for all groups

TESTS:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: TestSuite(G).run(skip = ["_test_an_element",                    "_test_
```

is_abelian()

Test whether this group is abelian.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.is_abelian()
Traceback (most recent call last):
...
NotImplementedError
```

is_commutative()

Test whether this group is commutative.

This is an alias for is_abelian, largely to make groups work well with the Factorization class.

(Note for developers: Derived classes should override is_abelian, not is_commutative.)

EXAMPLE:

```
sage: SL(2, 7).is_commutative()
False
```

is_finite()

Returns True if this group is finite.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.is_finite()
Traceback (most recent call last):
...
NotImplementedError
```

is_multiplicative()

Returns True if the group operation is given by * (rather than +).

Override for additive groups.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.is_multiplicative()
True
```

order()

Returns the number of elements of this group, which is either a positive integer or infinity.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.order()
Traceback (most recent call last):
...
NotImplementedError
```

quotient(H)

Return the quotient of this group by the normal subgroup H.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
```

```
sage: G.quotient(G)
Traceback (most recent call last):
...
NotImplementedError
```

random_element(*bound=None*)

Return a random element of this group.

EXAMPLES:

```
sage: from sage.groups.group import Group
sage: G = Group()
sage: G.random_element()
Traceback (most recent call last):
...
NotImplementedError
```

sage.groups.group.**is_Group**(*x*)

Return whether x is a group object.

INPUT:

- x – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: from sage.groups.group import is_Group
sage: is_Group(F)
True
sage: is_Group("a string")
False
```

CHAPTER

THREE

SET OF HOMOMORPHISMS BETWEEN TWO GROUPS.

sage.groups.group_homset.**GroupHomset**(G, H)

class sage.groups.group_homset.**GroupHomset_generic**(G, H)

Bases: sage.categories.homset.HomsetWithBase

This class will not work since morphism.GroupHomomorphism_coercion is undefined and morphism.GroupHomomorphism_im_gens is undefined.

natural_map()

sage.groups.group_homset.**is_GroupHomset**(H)

CHAPTER

FOUR

LIBGAP-BASED GROUPS

This module provides helper class for wrapping GAP groups via `libgap`. See `free_group` for an example how they are used.

The parent class keeps track of the libGAP element object, to use it in your Python parent you have to derive both from the suitable group parent and `ParentLibGAP`

```
sage: from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
sage: from sage.groups.group import Group
sage: class FooElement(ElementLibGAP):
...       pass
sage: class FooGroup(Group, ParentLibGAP):
...       Element = FooElement
...       def __init__(self):
...           lg = libgap(libgap.CyclicGroup(3))    # dummy
...           ParentLibGAP.__init__(self, lg)
...           Group.__init__(self)
```

Note how we call the constructor of both superclasses to initialize `Group` and `ParentLibGAP` separately. The parent class implements its output via LibGAP:

```
sage: FooGroup()
<pc group of size 3 with 1 generators>
sage: type(FooGroup().gap())
<type 'sage.libs.gap.element.GapElement'>
```

The element class is a subclass of `MultiplicativeGroupElement`. To use it, you just inherit from `ElementLibGAP`

```
sage: element = FooGroup().an_element()
sage: element
f1
```

The element class implements group operations and printing via LibGAP:

```
sage: element._repr_()
'f1'
sage: element * element
f1^2
```

AUTHORS:

- Volker Braun

class `sage.groups.libgap_wrapper.`**`ElementLibGAP`**

Bases: `sage.structure.element.MultiplicativeGroupElement`

A class for LibGAP-based Sage group elements

INPUT:

- `parent` – the Sage parent
- `libgap_element` – the libgap element that is being wrapped

EXAMPLES:

```
sage: from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
sage: from sage.groups.group import Group
sage: class FooElement(ElementLibGAP):
...       pass
sage: class FooGroup(Group, ParentLibGAP):
...       Element = FooElement
...       def __init__(self):
...           lg = libgap(libgap.CyclicGroup(3))    # dummy
...           ParentLibGAP.__init__(self, lg)
...           Group.__init__(self)
sage: FooGroup()
<pc group of size 3 with 1 generators>
sage: FooGroup().gens()
(f1,)
```

gap()

Returns a LibGAP representation of the element

OUTPUT:

A `GapElement`

EXAMPLES:

```
sage: G.<a,b> = FreeGroup('a, b')
sage: x = G([1, 2, -1, -2])
sage: x
a*b*a^-1*b^-1
sage: xg = x.gap()
sage: xg
a*b*a^-1*b^-1
sage: type(xg)
<type 'sage.libs.gap.element.GapElement'>
```

inverse()

Return the inverse of self.

TESTS:

```
sage: G = FreeGroup('a, b')
sage: x = G([1, 2, -1, -2])
sage: y = G([2, 2, 2, 1, -2, -2, -2])
sage: x.__invert__()
b*a*b^-1*a^-1
sage: y.__invert__()
b^3*a^-1*b^-3
sage: ~x
b*a*b^-1*a^-1
sage: x.inverse()
b*a*b^-1*a^-1
```

is_one()

Test whether the group element is the trivial element.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup('a, b')
sage: x = G([1, 2, -1, -2])
sage: x.is_one()
False
sage: (x * ~x).is_one()
True
```

class `sage.groups.libgap_wrapper.`**`ParentLibGAP`**(*libgap_parent*, *ambient=None*)

Bases: `sage.structure.sage_object.SageObject`

A class for parents to keep track of the GAP parent.

This is not a complete group in Sage, this class is only a base class that you can use to implement your own groups with LibGAP. See `libgap_group` for a minimal example of a group that is actually usable.

Your implementation definitely needs to supply

- `__reduce__()`: serialize the LibGAP group. Since GAP does not support Python pickles natively, you need to figure out yourself how you can recreate the group from a pickle.

INPUT:

- `libgap_parent` – the libgap element that is the parent in GAP.
- `ambient` – A derived class of `ParentLibGAP` or `None` (default). The ambient class if `libgap_parent` has been defined as a subgroup.

EXAMPLES:

```
sage: from sage.groups.libgap_wrapper import ElementLibGAP, ParentLibGAP
sage: from sage.groups.group import Group
sage: class FooElement(ElementLibGAP):
...       pass
sage: class FooGroup(Group, ParentLibGAP):
...       Element = FooElement
...       def __init__(self):
...           lg = libgap(libgap.CyclicGroup(3))    # dummy
...           ParentLibGAP.__init__(self, lg)
...           Group.__init__(self)
sage: FooGroup()
<pc group of size 3 with 1 generators>
```

`ambient`()

Return the ambient group of a subgroup.

OUTPUT:

A group containing `self`. If `self` has not been defined as a subgroup, we just return `self`.

EXAMPLES:

```
sage: G = FreeGroup(3)
sage: G.ambient() is G
True
```

`gap`()

Returns the gap representation of self

OUTPUT:

A `GapElement`

EXAMPLES:

```
sage: G = FreeGroup(3);  G
Free Group on generators {x0, x1, x2}
sage: G.gap()
<free group on the generators [ x0, x1, x2 ]>
sage: G.gap().parent()
C library interface to GAP
sage: type(G.gap())
<type 'sage.libs.gap.element.GapElement'>
```

This can be useful, for example, to call GAP functions that are not wrapped in Sage:

```
sage: G = FreeGroup(3)
sage: H = G.gap()
sage: H.DirectProduct(H)
<fp group on the generators [ f1, f2, f3, f4, f5, f6 ]>
sage: H.DirectProduct(H).RelatorsOfFpGroup()
[ f1^-1*f4^-1*f1*f4, f1^-1*f5^-1*f1*f5, f1^-1*f6^-1*f1*f6, f2^-1*f4^-1*f2*f4,
  f2^-1*f5^-1*f2*f5, f2^-1*f6^-1*f2*f6, f3^-1*f4^-1*f3*f4, f3^-1*f5^-1*f3*f5,
  f3^-1*f6^-1*f3*f6 ]
```

gen(i)

Return the i-th generator of self.

> **Warning:** Indexing starts at 0 as usual in Sage/Python. Not as in GAP, where indexing starts at 1.

INPUT:

- `i` – integer between 0 (inclusive) and `ngens()` (exclusive). The index of the generator.

OUTPUT:

The i-th generator of the group.

EXAMPLES:

```
sage: G = FreeGroup('a, b')
sage: G.gen(0)
a
sage: G.gen(1)
b
```

generators()

Returns the generators of the group.

EXAMPLES:

```
sage: G = FreeGroup(2)
sage: G.gens()
(x0, x1)
sage: H = FreeGroup('a, b, c')
sage: H.gens()
(a, b, c)
```

`generators()` is an alias for `gens()`

```
sage: G = FreeGroup('a, b')
sage: G.generators()
(a, b)
sage: H = FreeGroup(3, 'x')
sage: H.generators()
(x0, x1, x2)
```

gens()

Returns the generators of the group.

EXAMPLES:

```
sage: G = FreeGroup(2)
sage: G.gens()
(x0, x1)
sage: H = FreeGroup('a, b, c')
sage: H.gens()
(a, b, c)
```

`generators()` is an alias for `gens()`

```
sage: G = FreeGroup('a, b')
sage: G.generators()
(a, b)
sage: H = FreeGroup(3, 'x')
sage: H.generators()
(x0, x1, x2)
```

is_subgroup()

Return whether the group was defined as a subgroup of a bigger group.

You can access the contaning group with `ambient()`.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: G = FreeGroup(3)
sage: G.is_subgroup()
False
```

ngens()

Return the number of generators of self.

OUTPUT:

Integer.

EXAMPLES:

```
sage: G = FreeGroup(2)
sage: G.ngens()
2
```

TESTS:

```
sage: type(G.ngens())
<type 'sage.rings.integer.Integer'>
```

one()

Returns the identity element of self

EXAMPLES:

```
sage: G = FreeGroup(3)
sage: G.one()
1
sage: G.one() == G([])
True
sage: G.one().Tietze()
()
```

subgroup (*generators*)

Return the subgroup generated.

INPUT:

- `generators` – a list/tuple/iterable of group elements.

OUTPUT:

The subgroup generated by `generators`.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F.subgroup([a^2*b]);  G
Group([ a^2*b ])
sage: G.gens()
(a^2*b,)
```

CHAPTER

FIVE

GENERIC LIBGAP-BASED GROUP

This is useful if you need to use a GAP group implementation in Sage that does not have a dedicated Sage interface.

If you want to implement your own group class, you should not derive from this but directly from `ParentLibGAP`.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G_gap = libgap.Group([ (a*b^2).gap() ])
sage: from sage.groups.libgap_group import GroupLibGAP
sage: G = GroupLibGAP(G_gap);  G
Group([ a*b^2 ])
sage: type(G)
<class 'sage.groups.libgap_group.GroupLibGAP_with_category'>
sage: G.gens()
(a*b^2,)
```

class `sage.groups.libgap_group.`**`GroupLibGAP`**`(`**args*, ***kwds*`)`

Bases: `sage.groups.group.Group`, `sage.groups.libgap_wrapper.ParentLibGAP`

Group interface for LibGAP-based groups.

INPUT:

Same as `ParentLibGAP`.

TESTS:

```
sage: F.<a,b> = FreeGroup()
sage: G_gap = libgap.Group([ (a*b^2).gap() ])
sage: from sage.groups.libgap_group import GroupLibGAP
sage: G = GroupLibGAP(G_gap);  G
Group([ a*b^2 ])
sage: g = G.gen(0);  g
a*b^2
sage: TestSuite(G).run(skip=['_test_pickling', '_test_elements'])
sage: TestSuite(g).run(skip=['_test_pickling'])
```

`Element`

alias of `ElementLibGAP`

CHAPTER

SIX

MIX-IN CLASS FOR LIBGAP-BASED GROUPS

This class adds access to GAP functionality to groups such that parent and element have a `gap()` method that returns a libGAP object for the parent/element.

If your group implementation uses libgap, then you should add `GroupMixinLibGAP` as the first class that you are deriving from. This ensures that it properly overrides any default methods that just raise `NotImplemented`.

class `sage.groups.libgap_mixin.`**`GroupElementMixinLibGAP`**

Bases: `object`

`order``()`

Return the order of this group element, which is the smallest positive integer n such that $g^n = 1$, or +Infinity if no such integer exists.

EXAMPLES:

```
sage: k = GF(7);
sage: G = MatrixGroup([matrix(k,2,[1,1,0,1]), matrix(k,2,[1,0,0,2])]); G
Matrix group over Finite Field of size 7 with 2 generators (
[1 1]  [1 0]
[0 1], [0 2]
)
sage: G.order()
21
sage: G.gen(0).order(), G.gen(1).order()
(7, 3)

sage: k = QQ;
sage: G = MatrixGroup([matrix(k,2,[1,1,0,1]), matrix(k,2,[1,0,0,2])]); G
Matrix group over Rational Field with 2 generators (
[1 1]  [1 0]
[0 1], [0 2]
)
sage: G.order()
+Infinity
sage: G.gen(0).order(), G.gen(1).order()
(+Infinity, +Infinity)

sage: gl = GL(2, ZZ);  gl
General Linear Group of degree 2 over Integer Ring
sage: g = gl.gen(2);  g
[1 1]
[0 1]
sage: g.order()
+Infinity
```

word_problem(*gens=None*)

Solve the word problem.

This method writes the group element as a product of the elements of the list `gens`, or the standard generators of the parent of self if `gens` is None.

INPUT:

- `gens` – a list/tuple/iterable of elements (or objects that can be converted to group elements), or `None` (default). By default, the generators of the parent group are used.

OUTPUT:

A factorization object that contains information about the order of factors and the exponents. A `ValueError` is raised if the group element cannot be written as a word in `gens`.

ALGORITHM:

Use GAP, which has optimized algorithms for solving the word problem (the GAP functions `EpimorphismFromFreeGroup` and `PreImagesRepresentative`).

EXAMPLE:

```
sage: G = GL(2,5); G
General Linear Group of degree 2 over Finite Field of size 5
sage: G.gens()
(
[2 0]  [4 1]
[0 1], [4 0]
)
sage: G(1).word_problem([G.gen(0)])
1
sage: type(_)
<class 'sage.structure.factorization.Factorization'>

sage: g = G([0,4,1,4])
sage: g.word_problem()
([4 1]
 [4 0])^-1
```

Next we construct a more complicated element of the group from the generators:

```
sage: s,t = G.0, G.1
sage: a = (s * t * s); b = a.word_problem(); b
([2 0]
 [0 1]) *
([4 1]
 [4 0]) *
([2 0]
 [0 1])
sage: flatten(b)
[
[2 0]     [4 1]     [2 0]
[0 1], 1, [4 0], 1, [0 1], 1
]
sage: b.prod() == a
True
```

We solve the word problem using some different generators:

```
sage: s = G([2,0,0,1]); t = G([1,1,0,1]); u = G([0,-1,1,0])
sage: a.word_problem([s,t,u])
([2 0]
```

```
 [0 1])^-1 *
([1 1]
 [0 1])^-1 *
([0 4]
 [1 0]) *
([2 0]
 [0 1])^-1
```

We try some elements that don't actually generate the group:

```
sage: a.word_problem([t,u])
Traceback (most recent call last):
...
ValueError: word problem has no solution
```

AUTHORS:

- David Joyner and William Stein
- David Loeffler (2010): fixed some bugs
- Volker Braun (2013): LibGAP

class `sage.groups.libgap_mixin.`**`GroupMixinLibGAP`**

Bases: `object`

`cardinality``()`

Implements `EnumeratedSets.ParentMethods.cardinality()`.

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.cardinality()
51840

sage: G = SL(4,GF(3))
sage: G.cardinality()
12130560

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
480

sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.cardinality()
+Infinity

sage: G = Sp(4,GF(3))
sage: G.cardinality()
51840

sage: G = SL(4,GF(3))
sage: G.cardinality()
12130560

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
```

```
480

sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.cardinality()
+Infinity
```

center()

Return the center of this linear group as a subgroup.

OUTPUT:

The center as a subgroup.

EXAMPLES:

```
sage: G = SU(3,GF(2))
sage: G.center()
Matrix group over Finite Field in a of size 2^2 with 1 generators (
[a 0 0]
[0 a 0]
[0 0 a]
)
sage: GL(2,GF(3)).center()
Matrix group over Finite Field of size 3 with 1 generators (
[2 0]
[0 2]
)
sage: GL(3,GF(3)).center()
Matrix group over Finite Field of size 3 with 1 generators (
[2 0 0]
[0 2 0]
[0 0 2]
)
sage: GU(3,GF(2)).center()
Matrix group over Finite Field in a of size 2^2 with 1 generators (
[a + 1     0     0]
[    0 a + 1     0]
[    0     0 a + 1]
)

sage: A = Matrix(FiniteField(5), [[2,0,0], [0,3,0], [0,0,1]])
sage: B = Matrix(FiniteField(5), [[1,0,0], [0,1,0], [0,1,1]])
sage: MatrixGroup([A,B]).center()
Matrix group over Finite Field of size 5 with 1 generators (
[1 0 0]
[0 1 0]
[0 0 1]
)
```

class_function(*values*)

Return the class function with given values.

INPUT:

- `values` – list/tuple/iterable of numbers. The values of the class function on the conjugacy classes, in that order.

EXAMPLES:

```
sage: G = GL(2,GF(3))
sage: chi = G.class_function(range(8))
```

```
sage: list(chi)
[0, 1, 2, 3, 4, 5, 6, 7]
```

conjugacy_class(*g*)

Return the conjugacy class of `g`.

OUTPUT:

The conjugacy class of `g` in the group `self`. If `self` is the group denoted by G, this method computes the set $\{x^{-1}gx \mid x \in G\}$.

EXAMPLES:

```
sage: G = SL(2, QQ)
sage: g = G([[1,1],[0,1]])
sage: G.conjugacy_class(g)
Conjugacy class of [1 1]
[0 1] in Special Linear Group of degree 2 over Rational Field
```

conjugacy_class_representatives()

Return a set of representatives for each of the conjugacy classes of the group.

EXAMPLES:

```
sage: G = SU(3,GF(2))
sage: len(G.conjugacy_class_representatives())
16

sage: G = GL(2,GF(3))
sage: G.conjugacy_class_representatives()
(
[1 0]  [0 2]  [2 0]  [0 2]  [0 2]  [0 1]  [0 1]  [2 0]
[0 1], [1 1], [0 2], [1 2], [1 0], [1 2], [1 1], [0 1]
)

sage: len(GU(2,GF(5)).conjugacy_class_representatives())
36
```

conjugacy_classes()

Return a list with all the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = SL(2, GF(2))
sage: G.conjugacy_classes()
(Conjugacy class of [1 0]
 [0 1] in Special Linear Group of degree 2 over Finite Field of size 2,
 Conjugacy class of [0 1]
 [1 0] in Special Linear Group of degree 2 over Finite Field of size 2,
 Conjugacy class of [0 1]
 [1 1] in Special Linear Group of degree 2 over Finite Field of size 2)
```

irreducible_characters()

Returns the irreducible characters of the group.

OUTPUT:

A tuple containing all irreducible characters.

EXAMPLES:

```
sage: G = GL(2,2)
sage: G.irreducible_characters()
(Character of General Linear Group of degree 2 over Finite Field of size 2,
 Character of General Linear Group of degree 2 over Finite Field of size 2,
 Character of General Linear Group of degree 2 over Finite Field of size 2)
```

is_abelian()

Test whether the group is Abelian.

OUTPUT:

Boolean. `True` if this group is an Abelian group.

EXAMPLES:

```
sage: SL(1, 17).is_abelian()
True
sage: SL(2, 17).is_abelian()
False
```

is_finite()

Test whether the matrix group is finite.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: G = GL(2,GF(3))
sage: G.is_finite()
True
sage: SL(2,ZZ).is_finite()
False
```

is_isomorphic(*H*)

Test whether `self` and `H` are isomorphic groups.

INPUT:

- `H` – a group.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: m1 = matrix(GF(3), [[1,1],[0,1]])
sage: m2 = matrix(GF(3), [[1,2],[0,1]])
sage: F = MatrixGroup(m1)
sage: G = MatrixGroup(m1, m2)
sage: H = MatrixGroup(m2)
sage: F.is_isomorphic(G)
True
sage: G.is_isomorphic(H)
True
sage: F.is_isomorphic(H)
True
sage: F==G, G==H, F==H
(False, False, False)
```

list()

List all elements of this group.

OUTPUT:

A tuple containing all group elements in a random but fixed order.

EXAMPLES:

```
sage: F = GF(3)
sage: gens = [matrix(F,2, [1,0, -1,1]), matrix(F, 2, [1,1,0,1])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
24
sage: v = G.list()
sage: len(v)
24
sage: v[:5]
(
[1 0]  [2 0]  [0 1]  [0 2]  [1 2]
[0 1], [0 2], [2 0], [1 0], [2 2]
)
sage: all(g in G for g in G.list())
True
```

An example over a ring (see trac 5241):

```
sage: M1 = matrix(ZZ,2,[[-1,0],[0,1]])
sage: M2 = matrix(ZZ,2,[[1,0],[0,-1]])
sage: M3 = matrix(ZZ,2,[[-1,0],[0,-1]])
sage: MG = MatrixGroup([M1, M2, M3])
sage: MG.list()
(
[1 0]  [ 1  0]  [-1  0]  [-1  0]
[0 1], [ 0 -1], [ 0  1], [ 0 -1]
)
sage: MG.list()[1]
[ 1  0]
[ 0 -1]
sage: MG.list()[1].parent()
Matrix group over Integer Ring with 3 generators (
[-1  0]  [ 1  0]  [-1  0]
[ 0  1], [ 0 -1], [ 0 -1]
)
```

An example over a field (see trac 10515):

```
sage: gens = [matrix(QQ,2,[1,0,0,1])]
sage: MatrixGroup(gens).list()
(
[1 0]
[0 1]
)
```

Another example over a ring (see trac 9437):

```
sage: len(SL(2, Zmod(4)).list())
48
```

An error is raised if the group is not finite:

```
sage: GL(2,ZZ).list()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

order()

Implements EnumeratedSets.ParentMethods.cardinality().

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.cardinality()
51840

sage: G = SL(4,GF(3))
sage: G.cardinality()
12130560

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
480

sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.cardinality()
+Infinity

sage: G = Sp(4,GF(3))
sage: G.cardinality()
51840

sage: G = SL(4,GF(3))
sage: G.cardinality()
12130560

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
480

sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.cardinality()
+Infinity
```

random_element()

Return a random element of this group.

OUTPUT:

A group element.

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.random_element()   # random
[2 1 1 1]
[1 0 2 1]
[0 1 1 0]
```

```
[1 0 0 1]
sage: G.random_element() in G
True

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.random_element()  # random
[1 3]
[0 3]
sage: G.random_element() in G
True
```

CHAPTER SEVEN

PARI GROUPS

class `sage.groups.pari_group.`**`PariGroup`**`(`*x*, *degree=None*`)`

Bases: `sage.groups.old.Group`

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: G = f.galois_group(pari_group=True); G
PARI group [24, -1, 5, "S4"] of degree 4
sage: G.category()
Category of finite groups
```

Caveat: fix those tests and/or document precisely that this is an abstract group without explicit elements:

```
sage: TestSuite(G).run(skip = ["_test_an_element",                                          "_t
```

`degree``()`

`order``()`

`permutation_group``()`

CHAPTER

EIGHT

MISCELLANEOUS GENERIC FUNCTIONS

A collection of functions implementing generic algorithms in arbitrary groups, including additive and multiplicative groups.

In all cases the group operation is specified by a parameter 'operation', which is a string either one of the set of multiplication_names or addition_names specified below, or 'other'. In the latter case, the caller must provide an identity, inverse() and op() functions.

```
multiplication_names = ( 'multiplication', 'times', 'product', '*')
addition_names       = ( 'addition', 'plus', 'sum', '+')
```

Also included are a generic function for computing multiples (or powers), and an iterator for general multiples and powers.

EXAMPLES:

Some examples in the multiplicative group of a finite field:

- Discrete logs:

  ```
  sage: K = GF(3^6,'b')
  sage: b = K.gen()
  sage: a = b^210
  sage: discrete_log(a, b, K.order()-1)
  210
  ```

- Linear relation finder:

  ```
  sage: F.<a>=GF(3^6,'a')
  sage: a.multiplicative_order().factor()
  2^3 * 7 * 13
  sage: b=a^7
  sage: c=a^13
  sage: linear_relation(b,c,'*')
  (13, 7)
  sage: b^13==c^7
  True
  ```

- Orders of elements:

  ```
  sage: k.<a> = GF(5^5)
  sage: b = a^4
  sage: order_from_multiple(b,5^5-1,operation='*')
  781
  sage: order_from_bounds(b,(5^4,5^5),operation='*')
  781
  ```

Some examples in the group of points of an elliptic curve over a finite field:

- Discrete logs:

```
sage: F=GF(37^2,'a')
sage: E=EllipticCurve(F,[1,1])
sage: F.<a>=GF(37^2,'a')
sage: E=EllipticCurve(F,[1,1])
sage: P=E(25*a + 16 , 15*a + 7 )
sage: P.order()
672
sage: Q=39*P; Q
(36*a + 32 : 5*a + 12 : 1)
sage: discrete_log(Q,P,P.order(),operation='+')
39
```

- Linear relation finder:

```
sage: F.<a>=GF(3^6,'a')
sage: E=EllipticCurve([a^5 + 2*a^3 + 2*a^2 + 2*a, a^4 + a^3 + 2*a + 1])
sage: P=E(a^5 + a^4 + a^3 + a^2 + a + 2 , 0)
sage: Q=E(2*a^3 + 2*a^2 + 2*a , a^3 + 2*a^2 + 1)
sage: linear_relation(P,Q,'+')
(1, 2)
sage: P == 2*Q
True
```

- Orders of elements:

```
sage: k.<a> = GF(5^5)
sage: E = EllipticCurve(k,[2,4])
sage: P = E(3*a^4 + 3*a , 2*a + 1 )
sage: M = E.cardinality(); M
3227
sage: plist = M.prime_factors()
sage: order_from_multiple(P, M, plist, operation='+')
3227
sage: Q = E(0,2)
sage: order_from_multiple(Q, M, plist, operation='+')
7
sage: order_from_bounds(Q, Hasse_bounds(5^5), operation='+')
7
```

`sage.groups.generic.`**bsgs**(*a*, *b*, *bounds*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*)

Totally generic discrete baby-step giant-step function.

Solves $na = b$ (or $a^n = b$) with $lb \leq n \leq ub$ where `bounds==(lb,ub)`, raising an error if no such n exists.

a and b must be elements of some group with given identity, inverse of `x` given by `inverse(x)`, and group operation on `x`, `y` by `op(x,y)`.

If operation is '*' or '+' then the other arguments are provided automatically; otherwise they must be provided by the caller.

INPUT:

- `a` - group element
- `b` - group element
- `bounds` - a 2-tuple of integers `(lower,upper)` with `0<=lower<=upper`
- `operation` - string: '*', '+', 'other'
- `identity` - the identity element of the group

- `inverse()` - function of 1 argument `x` returning inverse of `x`
- `op()` - function of 2 arguments `x`, `y` returning `x*y` in group

OUTPUT:

An integer n such that $a^n = b$ (or $na = b$). If no such n exists, this function raises a ValueError exception.

NOTE: This is a generalization of discrete logarithm. One situation where this version is useful is to find the order of an element in a group where we only have bounds on the group order (see the elliptic curve example below).

ALGORITHM: Baby step giant step. Time and space are soft $O(\sqrt{n})$ where n is the difference between upper and lower bounds.

EXAMPLES:

```
sage: b = Mod(2,37);  a = b^20
sage: bsgs(b, a, (0,36))
20

sage: p=next_prime(10^20)
sage: a=Mod(2,p); b=a^(10^25)
sage: bsgs(a, b, (10^25-10^6,10^25+10^6)) == 10^25
True

sage: K = GF(3^6,'b')
sage: a = K.gen()
sage: b = a^210
sage: bsgs(a, b, (0,K.order()-1))
210

sage: K.<z>=CyclotomicField(230)
sage: w=z^500
sage: bsgs(z,w,(0,229))
40
```

An additive example in an elliptic curve group:

```
sage: F.<a> = GF(37^5)
sage: E = EllipticCurve(F, [1,1])
sage: P = E.lift_x(a); P
(a : 28*a^4 + 15*a^3 + 14*a^2 + 7 : 1)
```

This will return a multiple of the order of P:

```
sage: bsgs(P,P.parent()(0),Hasse_bounds(F.order()),operation='+')
69327408
```

AUTHOR:

- John Cremona (2008-03-15)

`sage.groups.generic.`**`discrete_log`**(*a*, *base*, *ord=None*, *bounds=None*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*)

Totally generic discrete log function.

INPUT:

- `a` - group element
- `base` - group element (the base)
- `ord` - integer (multiple of order of base, or `None`)

- `bounds` - a priori bounds on the log
- `operation` - string: '*', '+', 'other'
- `identity` - the group's identity
- `inverse()` - function of 1 argument `x` returning inverse of `x`
- `op()` - function of 2 arguments `x`, `y` returning `x*y` in group

`a` and `base` must be elements of some group with identity given by identity, inverse of `x` by `inverse(x)`, and group operation on `x`, `y` by `op(x,y)`.

If operation is '*' or '+' then the other arguments are provided automatically; otherwise they must be provided by the caller.

OUTPUT: Returns an integer n such that $b^n = a$ (or $nb = a$), assuming that `ord` is a multiple of the order of the base b. If `ord` is not specified, an attempt is made to compute it.

If no such n exists, this function raises a ValueError exception.

> **Warning:** If `x` has a log method, it is likely to be vastly faster than using this function. E.g., if `x` is an integer modulo n, use its log method instead!

ALGORITHM: Pohlig-Hellman and Baby step giant step.

EXAMPLES:

```
sage: b = Mod(2,37);  a = b^20
sage: discrete_log(a, b)
20
sage: b = Mod(2,997);  a = b^20
sage: discrete_log(a, b)
20

sage: K = GF(3^6,'b')
sage: b = K.gen()
sage: a = b^210
sage: discrete_log(a, b, K.order()-1)
210

sage: b = Mod(1,37);  x = Mod(2,37)
sage: discrete_log(x, b)
Traceback (most recent call last):
...
ValueError: No discrete log of 2 found to base 1
sage: b = Mod(1,997);  x = Mod(2,997)
sage: discrete_log(x, b)
Traceback (most recent call last):
...
ValueError: No discrete log of 2 found to base 1

See trac\#2356:
sage: F.<w> = GF(121)
sage: v = w^120
sage: v.log(w)
0

sage: K.<z>=CyclotomicField(230)
sage: w=z^50
sage: discrete_log(w,z)
50
```

An example where the order is infinite: note that we must give an upper bound here:

```
sage: K.<a> = QuadraticField(23)
sage: eps = 5*a-24        # a fundamental unit
sage: eps.multiplicative_order()
+Infinity
sage: eta = eps^100
sage: discrete_log(eta,eps,bounds=(0,1000))
100
```

In this case we cannot detect negative powers:

```
sage: eta = eps^(-3)
sage: discrete_log(eta,eps,bounds=(0,100))
Traceback (most recent call last):
...
ValueError: No discrete log of -11515*a - 55224 found to base 5*a - 24
```

But we can invert the base (and negate the result) instead:

```
sage: - discrete_log(eta^-1,eps,bounds=(0,100))
-3
```

An additive example: elliptic curve DLOG:

```
sage: F=GF(37^2,'a')
sage: E=EllipticCurve(F,[1,1])
sage: F.<a>=GF(37^2,'a')
sage: E=EllipticCurve(F,[1,1])
sage: P=E(25*a + 16 , 15*a + 7 )
sage: P.order()
672
sage: Q=39*P; Q
(36*a + 32 : 5*a + 12 : 1)
sage: discrete_log(Q,P,P.order(),operation='+')
39
```

An example of big smooth group:

```
sage: F.<a>=GF(2^63)
sage: g=F.gen()
sage: u=g**123456789
sage: discrete_log(u,g)
123456789
```

AUTHORS:

- William Stein and David Joyner (2005-01-05)
- John Cremona (2008-02-29) rewrite using `dict()` and make generic

`sage.groups.generic.`**`discrete_log_generic`**(*a*, *base*, *ord=None*, *bounds=None*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*)

Alias for `discrete_log`.

`sage.groups.generic.`**`discrete_log_lambda`**(*a*, *base*, *bounds*, *operation='*'*, *hash_function=<built-in function hash>*)

Pollard Lambda algorithm for computing discrete logarithms. It uses only a logarithmic amount of memory. It's useful if you have bounds on the logarithm. If you are computing logarithms in a whole finite group, you should use Pollard Rho algorithm.

INPUT:

- a - a group element
- base - a group element
- bounds - a couple (lb,ub) representing the range where we look for a logarithm
- operation - string: '+', '*' or 'other'
- hash_function – having an efficient hash function is critical for this algorithm

OUTPUT: Returns an integer n such that $a = base^n$ (or $a = n * base$)

ALGORITHM: Pollard Lambda, if bounds are (lb,ub) it has time complexity O(sqrt(ub-lb)) and space complexity O(log(ub-lb))

EXAMPLES:

```
sage: F.<a> = GF(2^63)
sage: discrete_log_lambda(a^1234567, a, (1200000,1250000))
1234567

sage: F.<a> = GF(37^5)
sage: E = EllipticCurve(F, [1,1])
sage: P = E.lift_x(a); P
(a : 9*a^4 + 22*a^3 + 23*a^2 + 30 : 1)
```

This will return a multiple of the order of P:

```
sage: discrete_log_lambda(P.parent()(0), P, Hasse_bounds(F.order()), operation='+')
69327408

sage: K.<a> = GF(89**5)
sage: hs = lambda x: hash(x) + 15
sage: discrete_log_lambda(a**(89**3 - 3), a, (89**2, 89**4), operation = '*', hash_function = hs
704966
```

AUTHOR:

– Yann Laigle-Chapuy (2009-01-25)

`sage.groups.generic.`**`discrete_log_rho`**(*a*, *base*, *ord=None*, *operation='*'*, *hash_function=<built-in function hash>*)

Pollard Rho algorithm for computing discrete logarithm in cyclic group of prime order. If the group order is very small it falls back to the baby step giant step algorithm.

INPUT:

- `a` – a group element
- `base` – a group element
- `ord` – the order of `base` or `None`, in this case we try to compute it
- `operation` – a string (default: `'*'`) denoting whether we are in an additive group or a multiplicative one
- `hash_function` – having an efficient hash function is critical for this algorithm (see examples)

OUTPUT: an integer n such that $a = base^n$ (or $a = n * base$)

ALGORITHM: Pollard rho for discrete logarithm, adapted from the article of Edlyn Teske, 'A space efficient algorithm for group structure computation'.

EXAMPLES:

```
sage: F.<a> = GF(2^13)
sage: g = F.gen()
sage: discrete_log_rho(g^1234, g)
1234

sage: F.<a> = GF(37^5)
sage: E = EllipticCurve(F, [1,1])
sage: G = (3*31*2^4)*E.lift_x(a)
sage: discrete_log_rho(12345*G, G, ord=46591, operation='+')
12345
```

It also works with matrices:

```
sage: A = matrix(GF(50021),[[10577,23999,28893],[14601,41019,30188],[3081,736,27092]])
sage: discrete_log_rho(A^1234567, A)
1234567
```

Beware, the order must be prime:

```
sage: I = IntegerModRing(171980)
sage: discrete_log_rho(I(2), I(3))
Traceback (most recent call last):
...
ValueError: for Pollard rho algorithm the order of the group must be prime
```

If it fails to find a suitable logarithm, it raises a `ValueError`:

```
sage: I = IntegerModRing(171980)
sage: discrete_log_rho(I(31002),I(15501))
Traceback (most recent call last):
...
ValueError: Pollard rho algorithm failed to find a logarithm
```

The main limitation on the hash function is that we don't want to have $hash(x * y) = hash(x) + hash(y)$:

```
sage: I = IntegerModRing(next_prime(2^23))
sage: def test():
....:     try:
....:          discrete_log_rho(I(123456),I(1),operation='+')
....:     except Exception:
....:          print "FAILURE"
sage: test()  # random failure
FAILURE
```

If this happens, we can provide a better hash function:

```
sage: discrete_log_rho(I(123456),I(1),operation='+', hash_function=lambda x: hash(x*x))
123456
```

AUTHOR:

- Yann Laigle-Chapuy (2009-09-05)

`sage.groups.generic.`**`linear_relation`**(*P*, *Q*, *operation='+'*, *identity=None*, *inverse=None*, *op=None*)

Function which solves the equation `a*P=m*Q` or `P^a=Q^m`.

Additive version: returns (a, m) with minimal $m > 0$ such that $aP = mQ$. Special case: if $\langle P\rangle$ and $\langle Q\rangle$ intersect only in $\{0\}$ then $(a, m) = (0, n)$ where n is `Q.additive_order()`.

Multiplicative version: returns (a, m) with minimal $m > 0$ such that $P^a = Q^m$. Special case: if $\langle P\rangle$ and $\langle Q\rangle$ intersect only in $\{1\}$ then $(a, m) = (0, n)$ where n is `Q.multiplicative_order()`.

ALGORITHM:

Uses the generic `bsgs()` function, and so works in general finite abelian groups.

EXAMPLES:

An additive example (in an elliptic curve group):

```
sage: F.<a>=GF(3^6,'a')
sage: E=EllipticCurve([a^5 + 2*a^3 + 2*a^2 + 2*a, a^4 + a^3 + 2*a + 1])
sage: P=E(a^5 + a^4 + a^3 + a^2 + a + 2 , 0)
sage: Q=E(2*a^3 + 2*a^2 + 2*a , a^3 + 2*a^2 + 1)
sage: linear_relation(P,Q,'+')
(1, 2)
sage: P == 2*Q
True
```

A multiplicative example (in a finite field's multiplicative group):

```
sage: F.<a>=GF(3^6,'a')
sage: a.multiplicative_order().factor()
2^3 * 7 * 13
sage: b=a^7
sage: c=a^13
sage: linear_relation(b,c,'*')
(13, 7)
sage: b^13==c^7
True
```

`sage.groups.generic.`**`merge_points`**(*P1*, *P2*, *operation='+'*, *identity=None*, *inverse=None*, *op=None*, *check=True*)

Returns a group element whose order is the lcm of the given elements.

INPUT:

- `P1` – a pair (g_1, n_1) where g_1 is a group element of order n_1
- `P2` – a pair (g_2, n_2) where g_2 is a group element of order n_2
- `operation` – string: '+' (default) or '*' or other. If other, the following must be supplied:
  - `identity`: the identity element for the group;
  - `inverse()`: a function of one argument giving the inverse of a group element;
  - **op(): a function of 2 arguments defining the group** binary operation.

OUTPUT:

A pair (g_3, n_3) where g_3 has order $n_3 = \text{lcm}(n_1, n_2)$.

EXAMPLES:

```
sage: F.<a>=GF(3^6,'a')
sage: b = a^7
sage: c = a^13
sage: ob = (3^6-1)//7
sage: oc = (3^6-1)//13
sage: merge_points((b,ob),(c,oc),operation='*')
```

```
(a^4 + 2*a^3 + 2*a^2, 728)
sage: d,od = merge_points((b,ob),(c,oc),operation='*')
sage: od == d.multiplicative_order()
True
sage: od == lcm(ob,oc)
True

sage: E=EllipticCurve([a^5 + 2*a^3 + 2*a^2 + 2*a, a^4 + a^3 + 2*a + 1])
sage: P=E(2*a^5 + 2*a^4 + a^3 + 2 , a^4 + a^3 + a^2 + 2*a + 2)
sage: P.order()
7
sage: Q=E(2*a^5 + 2*a^4 + 1 , a^5 + 2*a^3 + 2*a + 2 )
sage: Q.order()
4
sage: R,m = merge_points((P,7),(Q,4), operation='+')
sage: R.order() == m
True
sage: m == lcm(7,4)
True
```

sage.groups.generic.**multiple**(*a*, *n*, *operation='*'*, *identity=None*, *inverse=None*, *op=None*)

Returns either na or a^n, where n is any integer and a is a Python object on which a group operation such as addition or multiplication is defined. Uses the standard binary algorithm.

INPUT: See the documentation for `discrete_logarithm()`.

EXAMPLES:

```
sage: multiple(2,5)
32
sage: multiple(RealField()('2.5'),4)
39.0625000000000
sage: multiple(2,-3)
1/8
sage: multiple(2,100,'+') == 100*2
True
sage: multiple(2,100) == 2**100
True
sage: multiple(2,-100,) == 2**-100
True
sage: R.<x>=ZZ[]
sage: multiple(x,100)
x^100
sage: multiple(x,100,'+')
100*x
sage: multiple(x,-10)
1/x^10
```

Idempotence is detected, making the following fast:

```
sage: multiple(1,10^1000)
1

sage: E=EllipticCurve('389a1')
sage: P=E(-1,1)
sage: multiple(P,10,'+')
(645656132358737542773209599489/22817025904944891235367494656 : 525532176124281192881231818644174
sage: multiple(P,-10,'+')
(645656132358737542773209599489/22817025904944891235367494656 : -528978757629498440949529703029
```

class `sage.groups.generic.`**`multiples`**(*P*, *n*, *P0=None*, *indexed=False*, *operation='+'*, *op=None*)

Return an iterator which runs through `P0+i*P` for `i` in `range(n)`.

`P` and `P0` must be Sage objects in some group; if the operation is multiplication then the returned values are instead `P0*P**i`.

EXAMPLES:

```
sage: list(multiples(1,10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list(multiples(1,10,100))
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

sage: E=EllipticCurve('389a1')
sage: P=E(-1,1)
sage: for Q in multiples(P,5): print Q, Q.height()/P.height()
(0 : 1 : 0) 0.000000000000000
(-1 : 1 : 1) 1.00000000000000
(10/9 : -35/27 : 1) 4.00000000000000
(26/361 : -5720/6859 : 1) 9.00000000000000
(47503/16641 : 9862190/2146689 : 1) 16.0000000000000

sage: R.<x>=ZZ[]
sage: list(multiples(x,5))
[0, x, 2*x, 3*x, 4*x]
sage: list(multiples(x,5,operation='*'))
[1, x, x^2, x^3, x^4]
sage: list(multiples(x,5,indexed=True))
[(0, 0), (1, x), (2, 2*x), (3, 3*x), (4, 4*x)]
sage: list(multiples(x,5,indexed=True,operation='*'))
[(0, 1), (1, x), (2, x^2), (3, x^3), (4, x^4)]
sage: for i,y in multiples(x,5,indexed=True): print "%s  times %s = %s"%(i,x,y)
0  times x = 0
1  times x = x
2  times x = 2*x
3  times x = 3*x
4  times x = 4*x

sage: for i,n in multiples(3,5,indexed=True,operation='*'):  print "3 to the power %s = %s"%(i,n
3 to the power 0 = 1
3 to the power 1 = 3
3 to the power 2 = 9
3 to the power 3 = 27
3 to the power 4 = 81
```

`next`()

Returns the next item in this multiples iterator.

`sage.groups.generic.`**`order_from_bounds`**(*P*, *bounds*, *d=None*, *operation='+'*, *identity=None*, *inverse=None*, *op=None*)

Generic function to find order of a group element, given only upper and lower bounds for a multiple of the order (e.g. bounds on the order of the group of which it is an element)

INPUT:

- `P` - a Sage object which is a group element
- `bounds` - a 2-tuple `(lb,ub)` such that `m*P=0` (or `P**m=1`) for some `m` with `lb<=m<=ub`.
- `d` - (optional) a positive integer; only `m` which are multiples of this will be considered.
- `operation` - string: '+' (default) or '*' or other. If other, the following must be supplied:

–`identity`: the identity element for the group;

–`inverse()`: a function of one argument giving the inverse of a group element;

–`op()`: a function of 2 arguments defining the group binary operation.

Note: Typically `lb` and `ub` will be bounds on the group order, and from previous calculation we know that the group order is divisible by `d`.

EXAMPLES:

```
sage: k.<a> = GF(5^5)
sage: b = a^4
sage: order_from_bounds(b,(5^4,5^5),operation='*')
781
sage: E = EllipticCurve(k,[2,4])
sage: P = E(3*a^4 + 3*a , 2*a + 1 )
sage: bounds = Hasse_bounds(5^5)
sage: Q = E(0,2)
sage: order_from_bounds(Q, bounds, operation='+')
7
sage: order_from_bounds(P, bounds, 7, operation='+')
3227

sage: K.<z>=CyclotomicField(230)
sage: w=z^50
sage: order_from_bounds(w,(200,250),operation='*')
23
```

`sage.groups.generic.`**`order_from_multiple`**(*P*, *m*, *plist=None*, *factorization=None*, *check=True*, *operation='+'*)

Generic function to find order of a group element given a multiple of its order.

INPUT:

- `P` - a Sage object which is a group element;
- `m` - a Sage integer which is a multiple of the order of `P`, i.e. we require that `m*P=0` (or `P**m=1`);
- `check` - a Boolean (default:True), indicating whether we check if `m` really is a multiple of the order;
- `factorization` - the factorization of `m`, or `None` in which case this function will need to factor `m`;
- `plist` - a list of the prime factors of `m`, or `None` - kept for compatibility only, prefer the use of `factorization`;
- `operation` - string: '+' (default) or '*'.

Note: It is more efficient for the caller to factor `m` and cache the factors for subsequent calls.

EXAMPLES:

```
sage: k.<a> = GF(5^5)
sage: b = a^4
sage: order_from_multiple(b,5^5-1,operation='*')
781
sage: E = EllipticCurve(k,[2,4])
sage: P = E(3*a^4 + 3*a , 2*a + 1 )
sage: M = E.cardinality(); M
3227
sage: F = M.factor()
sage: order_from_multiple(P, M, factorization=F, operation='+')
```

```
3227
sage: Q = E(0,2)
sage: order_from_multiple(Q, M, factorization=F, operation='+')
7

sage: K.<z>=CyclotomicField(230)
sage: w=z^50
sage: order_from_multiple(w,230,operation='*')
23

sage: F=GF(2^1279,'a')
sage: n=F.cardinality()-1 # Mersenne prime
sage: order_from_multiple(F.random_element(),n,factorization=[(n,1)],operation='*')==n
True

sage: K.<a> = GF(3^60)
sage: order_from_multiple(a, 3^60-1, operation='*', check=False)
42391158275216203514294433200
```

sage.groups.generic.**structure_description**(*G*, *latex=False*)

Return a string that tries to describe the structure of `G`.

This methods wraps GAP's `StructureDescription` method.

Requires the *optional* `database_gap` package.

For full details, including the form of the returned string and the algorithm to build it, see GAP's documentation.

INPUT:

- `latex` – a boolean (default: `False`). If `True` return a LaTeX formatted string.

OUTPUT:

- string

> **Warning:** From GAP's documentation: The string returned by `StructureDescription` is **not** an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()            # optional - database_gap
'C6'
sage: G.structure_description(latex=True)   # optional - database_gap
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True))    # optional - database_gap
C_{6} \times C_{6}
```

This method is mainly intended for small groups or groups with few normal subgroups. Even then there are some surprises:

```
sage: D3 = DihedralGroup(3)
sage: D3.structure_description()     # optional - database_gap
'S3'
```

We use the Sage notation for the degree of dihedral groups:

```
sage: D4 = DihedralGroup(4)
sage: D4.structure_description()    # optional - database_gap
'D4'
```

Works for finitely presented groups (trac ticket #17573):

```
sage: F.<x, y> = FreeGroup()
sage: G=F / [x^2*y^-1, x^3*y^2, x*y*x^-1*y^-1]
sage: G.structure_description()     # optional - database_gap
'C7'
```

And matrix groups (trac ticket #17573):

```
sage: groups.matrix.GL(4,2).structure_description() # optional - database_gap
'A8'
```

CHAPTER

NINE

FREE GROUPS

Free groups and finitely presented groups are implemented as a wrapper over the corresponding GAP objects.

A free group can be created by giving the number of generators, or their names. It is also possible to create indexed generators:

```
sage: G.<x,y,z> = FreeGroup();  G
Free Group on generators {x, y, z}
sage: FreeGroup(3)
Free Group on generators {x0, x1, x2}
sage: FreeGroup('a,b,c')
Free Group on generators {a, b, c}
sage: FreeGroup(3,'t')
Free Group on generators {t0, t1, t2}
```

The elements can be created by operating with the generators, or by passing a list with the indices of the letters to the group:

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: a*b*c*a
a*b*c*a
sage: G([1,2,3,1])
a*b*c*a
sage: a * b / c * b^2
a*b*c^-1*b^2
sage: G([1,1,2,-1,-3,2])
a^2*b*a^-1*c^-1*b
```

You can use call syntax to replace the generators with a set of arbitrary ring elements:

```
sage: g =  a * b / c * b^2
sage: g(1,2,3)
8/3
sage: M1 = identity_matrix(2)
sage: M2 = matrix([[1,1],[0,1]])
sage: M3 = matrix([[0,1],[1,0]])
sage: g([M1, M2, M3])
[1 3]
[1 2]
```

AUTHORS:

- Miguel Angel Marco Buzunariz
- Volker Braun

sage.groups.free_group.**FreeGroup**(*n=None*, *names='x'*, *index_set=None*, *abelian=False*, ***kwds*)

Construct a Free Group.

INPUT:

- n – integer or None (default). The nnumber of generators. If not specified the names are counted.
- names – string or list/tuple/iterable of strings (default: 'x'). The generator names or name prefix.
- index_set – (optional) an index set for the generators; if specified then the optional keyword abelian can be used
- abelian – (default: False) whether to construct a free abelian group or a free group

Note: If you want to create a free group, it is currently preferential to use Groups().free(...) as that does not load GAP.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup();  G
Free Group on generators {a, b}
sage: H = FreeGroup('a, b')
sage: G is H
True
sage: FreeGroup(0)
Free Group on generators {}
```

The entry can be either a string with the names of the generators, or the number of generators and the prefix of the names to be given. The default prefix is 'x'

```
sage: FreeGroup(3)
Free Group on generators {x0, x1, x2}
sage: FreeGroup(3, 'g')
Free Group on generators {g0, g1, g2}
sage: FreeGroup()
Free Group on generators {x}
```

We give two examples using the index_set option:

```
sage: FreeGroup(index_set=ZZ)
Free group indexed by Integer Ring
sage: FreeGroup(index_set=ZZ, abelian=True)
Free abelian group indexed by Integer Ring
```

TESTS:

```
sage: G1 = FreeGroup(2, 'a,b')
sage: G2 = FreeGroup('a,b')
sage: G3.<a,b> = FreeGroup()
sage: G1 is G2, G2 is G3
(True, True)
```

class sage.groups.free_group.**FreeGroupElement**(*parent*, *x*)

Bases: sage.groups.libgap_wrapper.ElementLibGAP

A wrapper of GAP's Free Group elements.

INPUT:

- x – something that determines the group element. Either a GapElement or the Tietze list (see Tietze()) of the group element.

- `parent` – the parent `FreeGroup`.

EXAMPLES:

```
sage: G = FreeGroup('a, b')
sage: x = G([1, 2, -1, -2])
sage: x
a*b*a^-1*b^-1
sage: y = G([2, 2, 2, 1, -2, -2, -2])
sage: y
b^3*a*b^-3
sage: x*y
a*b*a^-1*b^2*a*b^-3
sage: y*x
b^3*a*b^-3*a*b*a^-1*b^-1
sage: x^(-1)
b*a*b^-1*a^-1
sage: x == x*y*y^(-1)
True
```

Tietze()

Return the Tietze list of the element.

The Tietze list of a word is a list of integers that represent the letters in the word. A positive integer i represents the letter corresponding to the i-th generator of the group. Negative integers represent the inverses of generators.

OUTPUT:

A tuple of integers.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: a.Tietze()
(1,)
sage: x = a^2 * b^(-3) * a^(-2)
sage: x.Tietze()
(1, 1, -2, -2, -2, -1, -1)
```

TESTS:

```
sage: type(a.Tietze())
<type 'tuple'>
sage: type(a.Tietze()[0])
<type 'sage.rings.integer.Integer'>
```

fox_derivative(*gen*, *im_gens=None*, *ring=None*)

Return the Fox derivative of `self` with respect to a given generator `gen` of the free group.

Let F be a free group with free generators $x_1, x_2, \ldots, x_n$. Let $j \in \{1, 2, \ldots, n\}$. Let $a_1, a_2, \ldots, a_n$ be n invertible elements of a ring A. Let $a : F \to A^\times$ be the (unique) homomorphism from F to the multiplicative group of invertible elements of A which sends each x_i to a_i. Then, we can define a map $\partial_j : F \to A$ by the requirements that

$$\partial_j(x_i) = \delta_{i,j} \qquad \text{for all indices } i \text{ and } j$$

and

$$\partial_j(uv) = \partial_j(u) + a(u)\partial_j(v) \qquad \text{for all } u, v \in F.$$

This map ∂_j is called the j-th Fox derivative on F induced by $(a_1, a_2, \ldots, a_n)$.

The most well-known case is when A is the group ring $\mathbf{Z}[F]$ of F over $\mathbf{Z}$, and when $a_i = x_i \in A$. In this case, ∂_j is simply called the j-th Fox derivative on F.

INPUT:

- `gen` – the generator with respect to which the derivative will be computed. If this is x_j, then the method will return ∂_j.
- `im_gens` (optional) – the images of the generators (given as a list or iterable). This is the list $(a_1, a_2, \ldots, a_n)$. If not provided, it defaults to $(x_1, x_2, \ldots, x_n)$ in the group ring $\mathbf{Z}[F]$.
- `ring` (optional) – the ring in which the elements of the list $(a_1, a_2, \ldots, a_n)$ lie. If not provided, this ring is inferred from these elements.

OUTPUT:

The fox derivative of `self` with respect to `gen` (induced by `im_gens`). By default, it is an element of the group algebra with integer coefficients. If `im_gens` are provided, the result lives in the algebra where `im_gens` live.

EXAMPLES:

```
sage: G = FreeGroup(5)
sage: G.inject_variables()
Defining x0, x1, x2, x3, x4
sage: (~x0*x1*x0*x2*~x0).fox_derivative(x0)
-B[x0^-1] + B[x0^-1*x1] - B[x0^-1*x1*x0*x2*x0^-1]
sage: (~x0*x1*x0*x2*~x0).fox_derivative(x1)
B[x0^-1]
sage: (~x0*x1*x0*x2*~x0).fox_derivative(x2)
B[x0^-1*x1*x0]
sage: (~x0*x1*x0*x2*~x0).fox_derivative(x3)
0
```

If `im_gens` is given, the images of the generators are mapped to them:

```
sage: F=FreeGroup(3)
sage: a=F([2,1,3,-1,2])
sage: a.fox_derivative(F([1]))
B[x1] - B[x1*x0*x2*x0^-1]
sage: R.<t>=LaurentPolynomialRing(ZZ)
sage: a.fox_derivative(F([1]),[t,t,t])
t - t^2
sage: S.<t1,t2,t3>=LaurentPolynomialRing(ZZ)
sage: a.fox_derivative(F([1]),[t1,t2,t3])
-t2*t3 + t2
sage: R.<x,y,z>=QQ[]
sage: a.fox_derivative(F([1]),[x,y,z])
-y*z + y
sage: a.inverse().fox_derivative(F([1]),[x,y,z])
(z - 1)/(y*z)
```

The optional parameter `ring` determines the ring A:

```
sage: u = a.fox_derivative(F([1]), [1,2,3], ring=QQ)
sage: u
-4
sage: parent(u)
Rational Field
sage: u = a.fox_derivative(F([1]), [1,2,3], ring=R)
sage: u
-4
```

```
sage: parent(u)
Multivariate Polynomial Ring in x, y, z over Rational Field
```

TESTS:

```
sage: F=FreeGroup(3)
sage: a=F([])
sage: a.fox_derivative(F([1]))
0
sage: R.<t>=LaurentPolynomialRing(ZZ)
sage: a.fox_derivative(F([1]),[t,t,t])
0
```

syllables()

Return the syllables of the word.

Consider a free group element $g = x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k}$. The uniquely-determined subwords $x_i^{e_i}$ consisting only of powers of a single generator are called the syllables of g.

OUTPUT:

The tuple of syllables. Each syllable is given as a pair (x_i, e_i) consisting of a generator and a non-zero integer.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: w = a^2 * b^-1 * a^3
sage: w.syllables()
((a, 2), (b, -1), (a, 3))
```

class sage.groups.free_group.**FreeGroup_class**(*generator_names*, *libgap_free_group=None*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.groups.group.Group, sage.groups.libgap_wrapper.ParentLibGAP

A class that wraps GAP's FreeGroup

See FreeGroup() for details.

TESTS:

```
sage: G = FreeGroup('a, b')
sage: TestSuite(G).run()
```

Element

alias of FreeGroupElement

abelian_invariants()

Return the Abelian invariants of self.

The Abelian invariants are given by a list of integers $i_1 \dots i_j$, such that the abelianization of the group is isomorphic to

$$\mathbf{Z}/(i_1) \times \cdots \times \mathbf{Z}/(i_j)$$

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: F.abelian_invariants()
(0, 0)
```

quotient (*relations*)

Return the quotient of `self` by the normal subgroup generated by the given elements.

This quotient is a finitely presented groups with the same generators as `self`, and relations given by the elements of `relations`.

INPUT:

- `relations` – A list/tuple/iterable with the elements of the free group.

OUTPUT:

A finitely presented group, with generators corresponding to the generators of the free group, and relations corresponding to the elements in `relations`.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: F.quotient([a*b^2*a, b^3])
Finitely presented group < a, b | a*b^2*a, b^3 >
```

Division is shorthand for `quotient()`

```
sage: F /  [a*b^2*a, b^3]
Finitely presented group < a, b | a*b^2*a, b^3 >
```

Relations are converted to the free group, even if they are not elements of it (if possible)

```
sage: F1.<a,b,c,d>=FreeGroup()
sage: F2.<a,b>=FreeGroup()
sage: r=a*b/a
sage: r.parent()
Free Group on generators {a, b}
sage: F1/[r]
Finitely presented group < a, b, c, d | a*b*a^-1 >
```

rank ()

Return the number of generators of self.

Alias for `ngens()`.

OUTPUT:

Integer.

EXAMPLES:

```
sage: G = FreeGroup('a, b');  G
Free Group on generators {a, b}
sage: G.rank()
2
sage: H = FreeGroup(3, 'x')
sage: H
Free Group on generators {x0, x1, x2}
sage: H.rank()
3
```

`sage.groups.free_group.`**is_FreeGroup** (*x*)

Test whether x is a `FreeGroup_class`.

INPUT:

- x – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.groups.free_group import is_FreeGroup
sage: is_FreeGroup('a string')
False
sage: is_FreeGroup(FreeGroup(0))
True
sage: is_FreeGroup(FreeGroup(index_set=ZZ))
True
```

`sage.groups.free_group.`**`wrap_FreeGroup`**`(`*libgap_free_group*`)`

Wrap a LibGAP free group.

This function changes the comparison method of `libgap_free_group` to comparison by Python `id`. If you want to put the LibGAP free group into a container (set, dict) then you should understand the implications of `_set_compare_by_id()`. To be safe, it is recommended that you just work with the resulting Sage `FreeGroup_class`.

INPUT:

- `libgap_free_group` – a LibGAP free group.

OUTPUT:

A Sage `FreeGroup_class`.

EXAMPLES:

First construct a LibGAP free group:

```
sage: F = libgap.FreeGroup(['a', 'b'])
sage: type(F)
<type 'sage.libs.gap.element.GapElement'>
```

Now wrap it:

```
sage: from sage.groups.free_group import wrap_FreeGroup
sage: wrap_FreeGroup(F)
Free Group on generators {a, b}
```

TESTS:

Check that we can do it twice (see trac ticket #12339)

```
sage: G = libgap.FreeGroup(['a', 'b'])
sage: wrap_FreeGroup(G)
Free Group on generators {a, b}
```

CHAPTER

TEN

FINITELY PRESENTED GROUPS

Finitely presented groups are constructed as quotients of `free_group`:

```
sage: F.<a,b,c> = FreeGroup()
sage: G = F / [a^2, b^2, c^2, a*b*c*a*b*c]
sage: G
Finitely presented group < a, b, c | a^2, b^2, c^2, (a*b*c)^2 >
```

One can create their elements by mutiplying the generators or by specifying a Tietze list (see `Tietze()`) as in the case of free groups:

```
sage: G.gen(0) * G.gen(1)
a*b
sage: G([1,2,-1])
a*b*a^-1
sage: a.parent()
Free Group on generators {a, b, c}
sage: G.inject_variables()
Defining a, b, c
sage: a.parent()
Finitely presented group < a, b, c | a^2, b^2, c^2, (a*b*c)^2 >
```

Notice that, even if they are represented in the same way, the elements of a finitely presented group and the elements of the corresponding free group are not the same thing. However, they can be converted from one parent to the other:

```
sage: F.<a,b,c> = FreeGroup()
sage: G = F / [a^2,b^2,c^2,a*b*c*a*b*c]
sage: F([1])
a
sage: G([1])
a
sage: F([1]) is G([1])
False
sage: F([1]) == G([1])
False
sage: G(a*b/c)
a*b*c^-1
sage: F(G(a*b/c))
a*b*c^-1
```

Finitely presented groups are implemented via GAP. You can use the `gap()` method to access the underlying LibGAP object:

```
sage: G = FreeGroup(2)
sage: G.inject_variables()
```

```
Defining x0, x1
sage: H = G / (x0^2, (x0*x1)^2, x1^2)
sage: H.gap()
<fp group on the generators [ x0, x1 ]>
```

This can be useful, for example, to use GAP functions that are not yet wrapped in Sage:

```
sage: H.gap().LowerCentralSeries()
[ Group(<fp, no generators known>), Group(<fp, no generators known>) ]
```

The same holds for the group elements:

```
sage: G = FreeGroup(2)
sage: H = G / (G([1, 1]), G([2, 2, 2]), G([1, 2, -1, -2])); H
Finitely presented group < x0, x1 | x0^2, x1^3, x0*x1*x0^-1*x1^-1 >
sage: a = H([1])
sage: a
x0
sage: a.gap()
x0
sage: a.gap().Order()
2
sage: type(_)    # note that the above output is not a Sage integer
<type 'sage.libs.gap.element.GapElement_Integer'>
```

You can use call syntax to replace the generators with a set of arbitrary ring elements. For example, take the free abelian group obtained by modding out the commutator subgroup of the free group:

```
sage: G = FreeGroup(2)
sage: G_ab = G / [G([1, 2, -1, -2])]; G_ab
Finitely presented group < x0, x1 | x0*x1*x0^-1*x1^-1 >
sage: a,b = G_ab.gens()
sage: g = a * b
sage: M1 = matrix([[1,0],[0,2]])
sage: M2 = matrix([[0,1],[1,0]])
sage: g(3, 5)
15
sage: g(M1, M1)
[1 0]
[0 4]
sage: M1*M2 == M2*M1   # matrices do not commute
False
sage: g(M1, M2)
Traceback (most recent call last):
...
ValueError: the values do not satisfy all relations of the group
```

> **Warning:** Some methods are not guaranteed to finish since the word problem for finitely presented groups is, in general, undecidable. In those cases the process may run unil the available memory is exhausted.

REFERENCES:

- Wikipedia article Presentation_of_a_group
- Wikipedia article Word_problem_for_groups

AUTHOR:

- Miguel Angel Marco Buzunariz

class `sage.groups.finitely_presented.FinitelyPresentedGroup`(*free_group*, *relations*)

Bases: `sage.groups.libgap_mixin.GroupMixinLibGAP`, `sage.structure.unique_representation.Uni` `sage.groups.group.Group`, `sage.groups.libgap_wrapper.ParentLibGAP`

A class that wraps GAP's Finitely Presented Groups.

> **Warning:** You should use `quotient()` to construct finitely presented groups as quotients of free groups.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: H = G / [a, b^3]
sage: H
Finitely presented group < a, b | a, b^3 >
sage: H.gens()
(a, b)

sage: F.<a,b> = FreeGroup('a, b')
sage: J = F / (F([1]), F([2, 2, 2]))
sage: J is H
True

sage: G = FreeGroup(2)
sage: H = G / (G([1, 1]), G([2, 2, 2]))
sage: H.gens()
(x0, x1)
sage: H.gen(0)
x0
sage: H.ngens()
2
sage: H.gap()
<fp group on the generators [ x0, x1 ]>
sage: type(_)
<type 'sage.libs.gap.element.GapElement'>
```

Element

alias of `FinitelyPresentedGroupElement`

abelian_invariants()

Return the abelian invariants of `self`.

The abelian invariants are given by a list of integers $(i_1, \ldots, i_j)$, such that the abelianization of the group is isomorphic to $\mathbf{Z}/(i_1) \times \cdots \times \mathbf{Z}/(i_j)$.

EXAMPLES:

```
sage: G = FreeGroup(4, 'g')
sage: G.inject_variables()
Defining g0, g1, g2, g3
sage: H = G.quotient([g1^2, g2*g1*g2^(-1)*g1^(-1), g1*g3^(-2), g0^4])
sage: H.abelian_invariants()
(0, 4, 4)
```

ALGORITHM:

Uses GAP.

alexander_matrix(*im_gens=None*)

Return the Alexander matrix of the group.

This matrix is given by the fox derivatives of the relations with respect to the generators.

- `im_gens` – (optional) the images of the generators.

OUTPUT:

A matrix with coefficients in the group algebra. If `im_gens` is given, the coefficients will live in the same algebra as the given values. The result depends on the (fixed) choice of presentation.

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: H = G.quotient([a*b/a/b, a*c/a/c, c*b/c/b])
sage: H.alexander_matrix()
[     B[1] - B[a*b*a^-1] B[a] - B[a*b*a^-1*b^-1]                       0]
[     B[1] - B[a*c*a^-1]                       0 B[a] - B[a*c*a^-1*c^-1]]
[                      0 B[c] - B[c*b*c^-1*b^-1]      B[1] - B[c*b*c^-1]]
```

If we introduce the images of the generators, we obtain the result in the corresponding algebra.

```
sage: G.<a,b,c,d,e> = FreeGroup()
sage: H = G.quotient([a*b/a/b, a*c/a/c, a*d/a/d, b*c*d/(c*d*b), b*c*d/(d*b*c)])
sage: H.alexander_matrix()
[              B[1] - B[a*b*a^-1]          B[a] - B[a*b*a^-1*b^-1]
[              B[1] - B[a*c*a^-1]                                0           B[a] - B[a*c*a^-
[              B[1] - B[a*d*a^-1]                                0
[                                0             B[1] - B[b*c*d*b^-1]   B[b] - B[b*c*d*b^-1*d^-
[                                0        B[1] - B[b*c*d*c^-1*b^-1]              B[b] - B[b*c*
sage: R.<t1,t2,t3,t4> = LaurentPolynomialRing(ZZ)
sage: H.alexander_matrix([t1,t2,t3,t4])
[    -t2 + 1      t1 - 1           0           0           0]
[    -t3 + 1           0      t1 - 1           0           0]
[    -t4 + 1           0           0      t1 - 1           0]
[          0  -t3*t4 + 1      t2 - 1  t2*t3 - t3           0]
[          0     -t4 + 1 -t2*t4 + t2   t2*t3 - 1           0]
```

as_permutation_group(*limit=4096000*)

Return an isomorphic permutation group.

The generators of the resulting group correspond to the images by the isomorphism of the generators of the given group.

INPUT:

- `limit` – integer (default: 4096000). The maximal number of cosets before the computation is aborted.

OUTPUT:

A Sage `PermutationGroup()`. If the number of cosets exceeds the given `limit`, a `ValueError` is returned.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup()
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.as_permutation_group()
Permutation Group with generators [(1,2)(3,5)(4,6), (1,3,4)(2,5,6)]
```

```
sage: G.<a,b> = FreeGroup()
sage: H = G / [a^3*b]
sage: H.as_permutation_group(limit=1000)
Traceback (most recent call last):
```

```
...
ValueError: Coset enumeration exceeded limit, is the group finite?
```

ALGORITHM:

Uses GAP's coset enumeration on the trivial subgroup.

> **Warning:** This is in general not a decidable problem (in fact, it is not even posible to check if the group is finite or not). If the group is infinite, or too big, you should be prepared for a long computation that consumes all the memory without finishing if you do not set a sensible `limit`.

cardinality (*limit=4096000*)

Compute the cardinality of `self`.

INPUT:

- `limit` – integer (default: 4096000). The maximal number of cosets before the computation is aborted.

OUTPUT:

Integer or `Infinity`. The number of elements in the group.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup('a, b')
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.cardinality()
6

sage: F.<a,b,c> = FreeGroup()
sage: J = F / (F([1]), F([2, 2, 2]))
sage: J.cardinality()
+Infinity
```

ALGORITHM:

Uses GAP.

> **Warning:** This is in general not a decidable problem, so it is not guaranteed to give an answer. If the group is infinite, or too big, you should be prepared for a long computation that consumes all the memory without finishing if you do not set a sensible `limit`.

direct_product (*H*, *reduced=False*, *new_names=True*)

Return the direct product of `self` with finitely presented group `H`.

Calls GAP function `DirectProduct`, which returns the direct product of a list of groups of any representation.

From [JohnsonPG90] (pg 45, proposition 4): If G, H are groups presented by $\langle X \mid R\rangle$ and $\langle Y \mid S\rangle$ respectively, then their direct product has the presentation $\langle X, Y \mid R, S, [X, Y]\rangle$ where $[X, Y]$ denotes the set of commutators $\{x^{-1}y^{-1}xy \mid x \in X, y \in Y\}$.

INPUT:

- `H` – a finitely presented group
- `reduced` – (default: `False`) boolean; if `True`, then attempt to reduce the presentation of the product group

- `new_names` – (default: `True`) boolean; If `True`, then lexicographical variable names are assigned to the generators of the group to be returned. If `False`, the group to be returned keeps the generator names of the two groups forming the direct product. Note that one cannot ask to reduce the output and ask to keep the old variable names, as they they may change meaning in the output group if its presentation is reduced.

OUTPUT:

The direct product of `self` with `H` as a finitely presented group.

EXAMPLES:

```
sage: G = FreeGroup()
sage: C12 =  ( G / [G([1,1,1,1])] ).direct_product( G / [G([1,1,1])]); C12
Finitely presented group < a, b | a^4, b^3, a^-1*b^-1*a*b >
sage: C12.order(), C12.as_permutation_group().is_cyclic()
(12, True)
sage: klein = ( G / [G([1,1])] ).direct_product( G / [G([1,1])]); klein
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
sage: klein.order(), klein.as_permutation_group().is_cyclic()
(4, False)
```

We can keep the variable names from `self` and `H` to examine how new relations are formed:

```
sage: F = FreeGroup("a"); G = FreeGroup("g")
sage: X = G / [G.0^12]; A = F / [F.0^6]
sage: X.direct_product(A, new_names=False)
Finitely presented group < g, a | g^12, a^6, g^-1*a^-1*g*a >
sage: A.direct_product(X, new_names=False)
Finitely presented group < a, g | a^6, g^12, a^-1*g^-1*a*g >
```

Or we can attempt to reduce the output group presentation:

```
sage: F = FreeGroup("a"); G = FreeGroup("g")
sage: X = G / [G.0]; A = F / [F.0]
sage: X.direct_product(A, new_names=True)
Finitely presented group < a, b | a, b, a^-1*b^-1*a*b >
sage: X.direct_product(A, reduced=True, new_names=True)
Finitely presented group <  |  >
```

But we cannot do both:

```
sage: K = FreeGroup(['a','b'])
sage: D = K / [K.0^5, K.1^8]
sage: D.direct_product(D, reduced=True, new_names=False)
Traceback (most recent call last):
...
ValueError: cannot reduce output and keep old variable names
```

TESTS:

```
sage: G = FreeGroup()
sage: Dp = (G / [G([1,1])]).direct_product( G / [G([1,1,1,1,1,1])] )
sage: Dp.as_permutation_group().is_isomorphic(PermutationGroup(['(1,2)','(3,4,5,6,7,8)']))
True
sage: C7 = G / [G.0**7]; C6 =  G / [G.0**6]
sage: C14 = G / [G.0**14]; C3 =  G / [G.0**3]
sage: C7.direct_product(C6).is_isomorphic(C14.direct_product(C3))
True
sage: F = FreeGroup(2); D = F / [F([1,1,1,1,1]),F([2,2]),F([1,2])**2]
sage: D.direct_product(D).as_permutation_group().is_isomorphic(
```

```
....: direct_product_permgroups([DihedralGroup(5),DihedralGroup(5)]))
True
```

AUTHORS:

- Davis Shurbert (2013-07-20): initial version

REFERENCES:

free_group()

Return the free group (without relations).

OUTPUT:

A `FreeGroup()`.

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.free_group()
Free Group on generators {a, b, c}
sage: H.free_group() is G
True
```

order(*limit=4096000*)

Compute the cardinality of `self`.

INPUT:

- `limit` – integer (default: 4096000). The maximal number of cosets before the computation is aborted.

OUTPUT:

Integer or `Infinity`. The number of elements in the group.

EXAMPLES:

```
sage: G.<a,b> = FreeGroup('a, b')
sage: H = G / (a^2, b^3, a*b*~a*~b)
sage: H.cardinality()
6

sage: F.<a,b,c> = FreeGroup()
sage: J = F / (F([1]), F([2, 2, 2]))
sage: J.cardinality()
+Infinity
```

ALGORITHM:

Uses GAP.

> **Warning:** This is in general not a decidable problem, so it is not guaranteed to give an answer. If the group is infinite, or too big, you should be prepared for a long computation that consumes all the memory without finishing if you do not set a sensible `limit`.

relations()

Return the relations of the group.

OUTPUT:

The relations as a tuple of elements of `free_group()`.

EXAMPLES:

```
sage: F = FreeGroup(5, 'x')
sage: F.inject_variables()
Defining x0, x1, x2, x3, x4
sage: G = F.quotient([x0*x2, x3*x1*x3, x2*x1*x2])
sage: G.relations()
(x0*x2, x3*x1*x3, x2*x1*x2)
sage: all(rel in F for rel in G.relations())
True
```

rewriting_system()

Return the rewriting system corresponding to the finitely presented group. This rewriting system can be used to reduce words with respect to the relations.

If the rewriting system is transformed into a confluent one, the reduction process will give as a result the (unique) reduced form of an element.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F / [a^2,b^3,(a*b/a)^3,b*a*b*a]
sage: k = G.rewriting_system()
sage: k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1, (b*a)^2 >
with rules:
    a^2    --->    1
    b^3    --->    1
    (b*a)^2    --->    1
    a*b^3*a^-1    --->    1

sage: G([1,1,2,2,2])
a^2*b^3
sage: k.reduce(G([1,1,2,2,2]))
1
sage: k.reduce(G([2,2,1]))
b^2*a
sage: k.make_confluent()
sage: k.reduce(G([2,2,1]))
a*b
```

semidirect_product(*H*, *hom*, *check=True*, *reduced=False*)

The semidirect product of `self` with `H` via `hom`.

If there exists a homomorphism ϕ from a group G to the automorphism group of a group H, then we can define the semidirect product of G with H via ϕ as the cartesian product of G and H with the operation

$$(g_1, h_1)(g_2, h_2) = (g_1 g_2, \phi(g_2)(h_1) h_2).$$

INPUT:

- `H` – Finitely presented group which is implicitly acted on by `self` and can be naturally embedded as a normal subgroup of the semidirect product.
- `hom` – Homomorphism from `self` to the automorphism group of `H`. Given as a pair, with generators of `self` in the first slot and the images of the corresponding generators in the second. These images must be automorphisms of `H`, given again as a pair of generators and images.
- `check` – Boolean (default `True`). If `False` the defining homomorphism and automorphism images are not tested for validity. This test can be costly with large groups, so it can be bypassed if the user is confident that his morphisms are valid.

- `reduced` – Boolean (default `False`). If `True` then the method attempts to reduce the presentation of the output group.

OUTPUT:

The semidirect product of `self` with `H` via `hom` as a finitely presented group. See `PermutationGroup_generic.semidirect_product` for a more in depth explanation of a semidirect product.

AUTHORS:

- Davis Shurbert (8-1-2013)

EXAMPLES:

Group of order 12 as two isomorphic semidirect products:

```
sage: D4 = groups.presentation.Dihedral(4)
sage: C3 = groups.presentation.Cyclic(3)
sage: alpha1 = ([C3.gen(0)],[C3.gen(0)])
sage: alpha2 = ([C3.gen(0)],[C3([1,1])])
sage: S1 = D4.semidirect_product(C3, ([D4.gen(1), D4.gen(0)],[alpha1,alpha2]))
sage: C2 = groups.presentation.Cyclic(2)
sage: Q = groups.presentation.DiCyclic(3)
sage: a = Q([1]); b = Q([-2])
sage: alpha = (Q.gens(), [a,b])
sage: S2 = C2.semidirect_product(Q, ([C2.0],[alpha]))
sage: S1.is_isomorphic(S2)
True
```

Dihedral groups can be constructed as semidirect products of cyclic groups:

```
sage: C2 = groups.presentation.Cyclic(2)
sage: C8 = groups.presentation.Cyclic(8)
sage: hom = (C2.gens(), [ ([C8([1])], [C8([-1])]) ])
sage: D = C2.semidirect_product(C8, hom)
sage: D.as_permutation_group().is_isomorphic(DihedralGroup(8))
True
```

You can attempt to reduce the presentation of the output group:

```
sage: D = C2.semidirect_product(C8, hom); D
Finitely presented group < a, b, c, d |
 a^2, b^-1*a^-1*b*a*d^-1*c^-1, c^-1*a^-1*c*a*d^-1, d^-1*a^-1*d*a,
 b^2*c^-1, c^-1*b^-1*c*b, d^-1*b^-1*d*b, c^2*d^-1, d^-1*c^-1*d*c, d^2 >
sage: D = C2.semidirect_product(C8, hom, reduced=True); D
Finitely presented group < a, b | a^2, (a*b)^2, b^8 >

sage: C3 = groups.presentation.Cyclic(3)
sage: C4 = groups.presentation.Cyclic(4)
sage: hom = (C3.gens(), [(C4.gens(), C4.gens())])
sage: C3.semidirect_product(C4, hom)
Finitely presented group < a, b, c |
 a^3, b^-1*a^-1*b*a, c^-1*a^-1*c*a, b^2*c^-1, c^-1*b^-1*c*b, c^2 >
sage: D = C3.semidirect_product(C4, hom, reduced=True); D
Finitely presented group < a, b | a^3, b^4, b^-1*a^-1*b*a >
sage: D.as_permutation_group().is_cyclic()
True
```

You can turn off the checks for the validity of the input morphisms. This check is expensive but behavior is unpredictable if inputs are invalid and are not caught by these tests:

```
sage: C5 = groups.presentation.Cyclic(5)
sage: C12 = groups.presentation.Cyclic(12)
sage: hom = (C5.gens(), [(C12.gens(), C12.gens())])
sage: sp = C5.semidirect_product(C12, hom, check=False); sp
Finitely presented group < a, b, c, d |
 a^5, b^-1*a^-1*b*a, c^-1*a^-1*c*a, d^-1*a^-1*d*a, b^2*d^-1,
 c^-1*b^-1*c*b, d^-1*b^-1*d*b, c^3, d^-1*c^-1*d*c, d^2 >
sage: sp.as_permutation_group().is_cyclic(), sp.order()
(True, 60)
```

TESTS:

The following was fixed in Gap-4.7.2:

```
sage: C5.semidirect_product(C12, hom) == sp
True
```

A more complicated semidirect product:

```
sage: C = groups.presentation.Cyclic(7)
sage: D = groups.presentation.Dihedral(5)
sage: id1 = ([C.0], [(D.gens(),D.gens())])
sage: Se1 =  C.semidirect_product(D, id1)
sage: id2 = (D.gens(), [(C.gens(),C.gens()),(C.gens(),C.gens())])
sage: Se2 =  D.semidirect_product(C ,id2)
sage: Dp1 = C.direct_product(D);
sage: Dp1.is_isomorphic(Se1), Dp1.is_isomorphic(Se2)
(True, True)
```

Most checks for validity of input are left to GAP to handle:

```
sage: bad_aut = ([C.0], [(D.gens(),[D.0, D.0])])
sage: C.semidirect_product(D, bad_aut)
Traceback (most recent call last):
...
ValueError: images of input homomorphism must be automorphisms
sage: bad_hom = ([D.0, D.1], [(C.gens(),C.gens())])
sage: D.semidirect_product(C, bad_hom)
Traceback (most recent call last):
...
ValueError: libGAP: Error, <gens> and <imgs> must be lists of same length
```

simplification_isomorphism()

Return an isomorphism from `self` to a finitely presented group with a (hopefully) simpler presentation.

EXAMPLES:

```
sage: G.<a,b,c> = FreeGroup()
sage: H = G / [a*b*c, a*b^2, c*b/c^2]
sage: I = H.simplification_isomorphism()
sage: I
Generic morphism:
  From: Finitely presented group < a, b, c | a*b*c, a*b^2, c*b*c^-2 >
  To:   Finitely presented group < b |  >
sage: I(a)
b^-2
sage: I(b)
b
sage: I(c)
b
```

TESTS:

```
sage: F = FreeGroup(1)
sage: G = F.quotient([F.0])
sage: G.simplification_isomorphism()
Generic morphism:
  From: Finitely presented group < x | x >
  To:   Finitely presented group <  |  >
```

ALGORITHM:

Uses GAP.

simplified()

Return an isomorphic group with a (hopefully) simpler presentation.

OUTPUT:

A new finitely presented group. Use `simplification_isomorphism()` if you want to know the isomorphism.

EXAMPLES:

```
sage: G.<x,y> = FreeGroup()
sage: H = G /  [x ^5, y ^4, y*x*y^3*x ^3]
sage: H
Finitely presented group < x, y | x^5, y^4, y*x*y^3*x^3 >
sage: H.simplified()
Finitely presented group < x, y | y^4, y*x*y^-1*x^-2, x^5 >
```

A more complicate example:

```
sage: G.<e0, e1, e2, e3, e4, e5, e6, e7, e8, e9> = FreeGroup()
sage: rels = [e6, e5, e3, e9, e4*e7^-1*e6, e9*e7^-1*e0,
...           e0*e1^-1*e2, e5*e1^-1*e8, e4*e3^-1*e8, e2]
sage: H = G.quotient(rels);  H
Finitely presented group < e0, e1, e2, e3, e4, e5, e6, e7, e8, e9 |
e6, e5, e3, e9, e4*e7^-1*e6, e9*e7^-1*e0, e0*e1^-1*e2, e5*e1^-1*e8, e4*e3^-1*e8, e2 >
sage: H.simplified()
Finitely presented group < e0 | e0^2 >
```

structure_description(*G*, *latex=False*)

Return a string that tries to describe the structure of `G`.

This methods wraps GAP's `StructureDescription` method.

Requires the *optional* `database_gap` package.

For full details, including the form of the returned string and the algorithm to build it, see GAP's documentation.

INPUT:

- `latex` – a boolean (default: `False`). If `True` return a LaTeX formatted string.

OUTPUT:

- string

Warning: From GAP's documentation: The string returned by `StructureDescription` is **not** an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()             # optional - database_gap
'C6'
sage: G.structure_description(latex=True)   # optional - database_gap
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True))    # optional - database_gap
C_{6} \times C_{6}
```

This method is mainly intended for small groups or groups with few normal subgroups. Even then there are some surprises:

```
sage: D3 = DihedralGroup(3)
sage: D3.structure_description()     # optional - database_gap
'S3'
```

We use the Sage notation for the degree of dihedral groups:

```
sage: D4 = DihedralGroup(4)
sage: D4.structure_description()     # optional - database_gap
'D4'
```

Works for finitely presented groups (trac ticket #17573):

```
sage: F.<x, y> = FreeGroup()
sage: G=F / [x^2*y^-1, x^3*y^2, x*y*x^-1*y^-1]
sage: G.structure_description()      # optional - database_gap
'C7'
```

And matrix groups (trac ticket #17573):

```
sage: groups.matrix.GL(4,2).structure_description() # optional - database_gap
'A8'
```

class `sage.groups.finitely_presented.`**`FinitelyPresentedGroupElement`**(*parent*, *x*, *check=True*)

Bases: `sage.groups.free_group.FreeGroupElement`

A wrapper of GAP's Finitely Presented Group elements.

The elements are created by passing the Tietze list that determines them.

EXAMPLES:

```
sage: G = FreeGroup('a, b')
sage: H = G / [G([1]), G([2, 2, 2])]
sage: H([1, 2, 1, -1])
a*b
sage: H([1, 2, 1, -2])
a*b*a*b^-1
sage: x = H([1, 2, -1, -2])
sage: x
a*b*a^-1*b^-1
sage: y = H([2, 2, 2, 1, -2, -2, -2])
sage: y
b^3*a*b^-3
sage: x*y
a*b*a^-1*b^2*a*b^-3
sage: x^(-1)
b*a*b^-1*a^-1
```

Tietze()

Return the Tietze list of the element.

The Tietze list of a word is a list of integers that represent the letters in the word. A positive integer i represents the letter corresponding to the i-th generator of the group. Negative integers represent the inverses of generators.

OUTPUT:

A tuple of integers.

EXAMPLES:

```
sage: G = FreeGroup('a, b')
sage: H = G / (G([1]), G([2, 2, 2]))
sage: H.inject_variables()
Defining a, b
sage: a.Tietze()
(1,)
sage: x = a^2*b^(-3)*a^(-2)
sage: x.Tietze()
(1, 1, -2, -2, -2, -1, -1)
```

class sage.groups.finitely_presented.**RewritingSystem**(G)

Bases: object

A class that wraps GAP's rewriting systems.

A rewriting system is a set of rules that allow to transform one word in the group to an equivalent one.

If the rewriting system is confluent, then the transformated word is a unique reduced form of the element of the group.

Warning: Note that the process of making a rewriting system confluent might not end.

INPUT:

- G – a group

REFERENCES:

- Wikipedia article Knuth-Bendix_completion_algorithm

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F / [a*b/a/b]
sage: k = G.rewriting_system()
sage: k
Rewriting system of Finitely presented group < a, b | a*b*a^-1*b^-1 >
with rules:
    a*b*a^-1*b^-1    --->    1

sage: k.reduce(a*b*a*b)
(a*b)^2
sage: k.make_confluent()
sage: k
Rewriting system of Finitely presented group < a, b | a*b*a^-1*b^-1 >
with rules:
    b^-1*a^-1    --->    a^-1*b^-1
    b^-1*a    --->    a*b^-1
    b*a^-1    --->    a^-1*b
```

```
    b*a    --->    a*b

sage: k.reduce(a*b*a*b)
a^2*b^2
```

Todo

- Include support for different orderings (currently only shortlex is used).
- Include the GAP package kbmag for more functionalities, including automatic structures and faster compiled functions.

AUTHORS:

- Miguel Angel Marco Buzunariz (2013-12-16)

finitely_presented_group()

The finitely presented group where the rewriting system is defined.

EXAMPLES:

```
sage: F = FreeGroup(3)
sage: G = F / [ [1,2,3], [-1,-2,-3], [1,1], [2,2] ]
sage: k = G.rewriting_system()
sage: k.make_confluent()
sage: k
Rewriting system of Finitely presented group < x0, x1, x2 | x0*x1*x2, x0^-1*x1^-1*x2^-1, x0^
with rules:
    x0^-1    --->    x0
    x1^-1    --->    x1
    x2^-1    --->    x2
    x0^2    --->    1
    x0*x1    --->    x2
    x0*x2    --->    x1
    x1*x0    --->    x2
    x1^2    --->    1
    x1*x2    --->    x0
    x2*x0    --->    x1
    x2*x1    --->    x0
    x2^2    --->    1
sage: k.finitely_presented_group()
Finitely presented group < x0, x1, x2 | x0*x1*x2, x0^-1*x1^-1*x2^-1, x0^2, x1^2 >
```

free_group()

The free group after which the rewriting system is defined

EXAMPLES:

```
sage: F = FreeGroup(3)
sage: G = F / [ [1,2,3], [-1,-2,-3] ]
sage: k = G.rewriting_system()
sage: k.free_group()
Free Group on generators {x0, x1, x2}
```

gap()

The gap representation of the rewriting system.

EXAMPLES:

```
sage: F.<a,b>=FreeGroup()
sage: G=F/[a*a,b*b]
sage: k=G.rewriting_system()
sage: k.gap()
Knuth Bendix Rewriting System for Monoid( [ a, A, b, B ] ) with rules
[ [ a^2, <identity ...> ], [ a*A, <identity ...> ],
  [ A*a, <identity ...> ], [ b^2, <identity ...> ],
  [ b*B, <identity ...> ], [ B*b, <identity ...> ] ]
```

is_confluent()

Return `True` if the system is confluent and `False` otherwise.

EXAMPLES:

```
sage: F = FreeGroup(3)
sage: G = F / [F([1,2,1,2,1,3,-1]),F([2,2,2,1,1,2]),F([1,2,3])]
sage: k = G.rewriting_system()
sage: k.is_confluent()
False
sage: k
Rewriting system of Finitely presented group < x0, x1, x2 | (x0*x1)^2*x0*x2*x0^-1, x1^3*x0^2
with rules:
    x0*x1*x2    --->    1
    x1^3*x0^2*x1    --->    1
    (x0*x1)^2*x0*x2*x0^-1    --->    1

sage: k.make_confluent()
sage: k.is_confluent()
True
sage: k
Rewriting system of Finitely presented group < x0, x1, x2 | (x0*x1)^2*x0*x2*x0^-1, x1^3*x0^2
with rules:
    x0^-1    --->    x0
    x1^-1    --->    x1
    x0^2    --->    1
    x0*x1    --->    x2^-1
    x0*x2^-1    --->    x1
    x1*x0    --->    x2
    x1^2    --->    1
    x1*x2^-1    --->    x0*x2
    x1*x2    --->    x0
    x2^-1*x0    --->    x0*x2
    x2^-1*x1    --->    x0
    x2^-2    --->    x2
    x2*x0    --->    x1
    x2*x1    --->    x0*x2
    x2^2    --->    x2^-1
```

make_confluent()

Applies Knuth-Bendix algorithm to try to transform the rewriting system into a confluent one.

Note that this method does not return any object, just changes the rewriting sytem internally.

ALGORITHM:

Uses GAP's `MakeConfluent`.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F / [a^2,b^3,(a*b/a)^3,b*a*b*a]
sage: k = G.rewriting_system()
sage: k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1, (b*a)^2 >
with rules:
    a^2    --->    1
    b^3    --->    1
    (b*a)^2    --->    1
    a*b^3*a^-1    --->    1

sage: k.make_confluent()
sage: k
Rewriting system of Finitely presented group < a, b | a^2, b^3, a*b^3*a^-1, (b*a)^2 >
with rules:
    a^-1    --->    a
    a^2    --->    1
    b^-1*a    --->    a*b
    b^-2    --->    b
    b*a    --->    a*b^-1
    b^2    --->    b^-1
```

reduce(*element*)

Applies the rules in the rewriting system to the element, to obtain a reduced form.

If the rewriting system is confluent, this reduced form is unique for all words representing the same element.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F/[a^2, b^3, (a*b/a)^3, b*a*b*a]
sage: k = G.rewriting_system()
sage: k.reduce(b^4)
b
sage: k.reduce(a*b*a)
a*b*a
```

rules()

Return the rules that form the rewritig system.

OUTPUT:

A dictionary containing the rules of the rewriting system. Each key is a word in the free group, and its corresponding value is the word to which it is reduced.

EXAMPLES:

```
sage: F.<a,b> = FreeGroup()
sage: G = F / [a*a*a,b*b*a*a]
sage: k = G.rewriting_system()
sage: k
Rewriting system of Finitely presented group < a, b | a^3, b^2*a^2 >
with rules:
    a^3    --->    1
    b^2*a^2    --->    1

sage: k.rules()
{a^3: 1, b^2*a^2: 1}
sage: k.make_confluent()
```

```
sage: sorted(k.rules().items())
[(a^-2, a), (a^-1*b^-1, a*b), (a^-1*b, b^-1), (a^2, a^-1),
 (a*b^-1, b), (b^-1*a^-1, a*b), (b^-1*a, b), (b^-2, a^-1),
 (b*a^-1, b^-1), (b*a, a*b), (b^2, a)]
```

`sage.groups.finitely_presented.`**`wrap_FpGroup`**`(`*libgap_fpgroup*`)`

Wrap a GAP finitely presented group.

This function changes the comparison method of `libgap_free_group` to comparison by Python `id`. If you want to put the LibGAP free group into a container `(set, dict)` then you should understand the implications of `_set_compare_by_id()`. To be safe, it is recommended that you just work with the resulting Sage `FinitelyPresentedGroup`.

INPUT:

- `libgap_fpgroup` – a LibGAP finitely presented group

OUTPUT:

A Sage `FinitelyPresentedGroup`.

EXAMPLES:

First construct a LibGAP finitely presented group:

```
sage: F = libgap.FreeGroup(['a', 'b'])
sage: a_cubed = F.GeneratorsOfGroup()[0] ^ 3
sage: P = F / libgap([ a_cubed ]);   P
<fp group of size infinity on the generators [ a, b ]>
sage: type(P)
<type 'sage.libs.gap.element.GapElement'>
```

Now wrap it:

```
sage: from sage.groups.finitely_presented import wrap_FpGroup
sage: wrap_FpGroup(P)
Finitely presented group < a, b | a^3 >
```

CHAPTER

ELEVEN

NAMED FINITELY PRESENTED GROUPS

Construct groups of small order and “named” groups as quotients of free groups. These groups are available through tab completion by typing `groups.presentation.<tab>` or by importing the required methods. Tab completion is made available through Sage’s *group catalog*. Some examples are engineered from entries in [THOMAS-WOODS].

Groups available as finite presentations:

- Alternating group, A_n of order $n!/2$ – `groups.presentation.Alternating`
- Cyclic group, C_n of order n – `groups.presentation.Cyclic`
- Dicyclic group, nonabelian groups of order $4n$ with a unique element of order 2 – `groups.presentation.DiCyclic`
- Dihedral group, D_n of order $2n$ – `groups.presentation.Dihedral`
- Finitely generated abelian group, $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$ – `groups.presentation.FGAbelian`
- Klein four group, $C_2 \times C_2$ – `groups.presentation.KleinFour`
- Quaternion group of order 8 – `groups.presentation.Quaternion`
- Symmetric group, S_n of order $n!$ – `groups.presentation.Symmetric`

AUTHORS:

- Davis Shurbert (2013-06-21): initial version

EXAMPLES:

```
sage: groups.presentation.Cyclic(4)
Finitely presented group < a | a^4 >
```

You can also import the desired functions:

```
sage: from sage.groups.finitely_presented_named import CyclicPresentation
sage: CyclicPresentation(4)
Finitely presented group < a | a^4 >
```

`sage.groups.finitely_presented_named.`**`AlternatingPresentation`**`(n)`

Build the Alternating group of order $n!/2$ as a finitely presented group.

INPUT:

- `n` – The size of the underlying set of arbitrary symbols being acted on by the Alternating group of order $n!/2$.

OUTPUT:

Alternating group as a finite presentation, implementation uses GAP to find an isomorphism from a permutation representation to a finitely presented group representation. Due to this fact, the exact output presentation may not be the same for every method call on a constant `n`.

EXAMPLES:

```
sage: A6 = groups.presentation.Alternating(6)
sage: A6.as_permutation_group().is_isomorphic(AlternatingGroup(6)), A6.order()
(True, 360)
```

TESTS:

```
sage: #even permutation test..
sage: A1 = groups.presentation.Alternating(1); A2 = groups.presentation.Alternating(2)
sage: A1.is_isomorphic(A2), A1.order()
(True, 1)
sage: A3 = groups.presentation.Alternating(3); A3.order(), A3.as_permutation_group().is_cyclic()
(3, True)
sage: A8 = groups.presentation.Alternating(8); A8.order()
20160
```

`sage.groups.finitely_presented_named.`**`CyclicPresentation`**`(n)`

Build cyclic group of order n as a finitely presented group.

INPUT:

- `n` – The order of the cyclic presentation to be returned.

OUTPUT:

The cyclic group of order n as finite presentation.

EXAMPLES:

```
sage: groups.presentation.Cyclic(10)
Finitely presented group < a | a^10 >
sage: n = 8; C = groups.presentation.Cyclic(n)
sage: C.as_permutation_group().is_isomorphic(CyclicPermutationGroup(n))
True
```

TESTS:

```
sage: groups.presentation.Cyclic(0)
Traceback (most recent call last):
...
ValueError: finitely presented group order must be positive
```

`sage.groups.finitely_presented_named.`**`DiCyclicPresentation`**`(n)`

Build the dicyclic group of order $4n$, for $n \geq 2$, as a finitely presented group.

INPUT:

- `n` – positive integer, 2 or greater, determining the order of the group ($4n$).

OUTPUT:

The dicyclic group of order $4n$ is defined by the presentation

$$\langle a, x \mid a^{2n} = 1, x^2 = a^n, x^{-1}ax = a^{-1} \rangle$$

Note: This group is also available as a permutation group via `groups.permutation.DiCyclic`.

EXAMPLES:

```
sage: D = groups.presentation.DiCyclic(9); D
Finitely presented group < a, b | a^18, b^2*a^-9, b^-1*a*b*a >
sage: D.as_permutation_group().is_isomorphic(groups.permutation.DiCyclic(9))
True
```

TESTS:

```
sage: Q = groups.presentation.DiCyclic(2)
sage: Q.as_permutation_group().is_isomorphic(QuaternionGroup())
True
sage: all([groups.presentation.DiCyclic(i).as_permutation_group(
....: ).is_isomorphic(groups.permutation.DiCyclic(i)) for i in [5,8,12,2^5]])
True
sage: groups.presentation.DiCyclic(1)
Traceback (most recent call last):
...
ValueError: input integer must be greater than 1
```

`sage.groups.finitely_presented_named.`**`DihedralPresentation`**(*n*)

Build the Dihedral group of order $2n$ as a finitely presented group.

INPUT:

- `n` – The size of the set that D_n is acting on.

OUTPUT:

Dihedral group of order $2n$.

EXAMPLES:

```
sage: D = groups.presentation.Dihedral(7); D
Finitely presented group < a, b | a^7, b^2, (a*b)^2 >
sage: D.as_permutation_group().is_isomorphic(DihedralGroup(7))
True
```

TESTS:

```
sage: n = 9
sage: D = groups.presentation.Dihedral(n)
sage: D.ngens() == 2
True
sage: groups.presentation.Dihedral(0)
Traceback (most recent call last):
...
ValueError: finitely presented group order must be positive
```

`sage.groups.finitely_presented_named.`**`FinitelyGeneratedAbelianPresentation`**(*int_list*)

Return canonical presentation of finitely generated abelian group.

INPUT:

- `int_list` – List of integers defining the group to be returned, the defining list is reduced to the invariants of the input list before generating the corresponding group.

OUTPUT:

Finitely generated abelian group, $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \cdots \times \mathbf{Z}_{n_k}$ as a finite presentation, where n_i forms the invariants of the input list.

EXAMPLES:

```
sage: groups.presentation.FGAbelian([2,2])
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
sage: groups.presentation.FGAbelian([2,3])
Finitely presented group < a | a^6 >
sage: groups.presentation.FGAbelian([2,4])
Finitely presented group < a, b | a^2, b^4, a^-1*b^-1*a*b >
```

You can create free abelian groups:

```
sage: groups.presentation.FGAbelian([0])
Finitely presented group < a |  >
sage: groups.presentation.FGAbelian([0,0])
Finitely presented group < a, b | a^-1*b^-1*a*b >
sage: groups.presentation.FGAbelian([0,0,0])
Finitely presented group < a, b, c | a^-1*b^-1*a*b, a^-1*c^-1*a*c, b^-1*c^-1*b*c >
```

And various infinite abelian groups:

```
sage: groups.presentation.FGAbelian([0,2])
Finitely presented group < a, b | a^2, a^-1*b^-1*a*b >
sage: groups.presentation.FGAbelian([0,2,2])
Finitely presented group < a, b, c | a^2, b^2, a^-1*b^-1*a*b, a^-1*c^-1*a*c, b^-1*c^-1*b*c >
```

Outputs are reduced to minimal generators and relations:

```
sage: groups.presentation.FGAbelian([3,5,2,7,3])
Finitely presented group < a, b | a^3, b^210, a^-1*b^-1*a*b >
sage: groups.presentation.FGAbelian([3,210])
Finitely presented group < a, b | a^3, b^210, a^-1*b^-1*a*b >
```

The trivial group is an acceptable output:

```
sage: groups.presentation.FGAbelian([])
Finitely presented group <  |  >
sage: groups.presentation.FGAbelian([1])
Finitely presented group <  |  >
sage: groups.presentation.FGAbelian([1,1,1,1,1,1,1,1,1,1])
Finitely presented group <  |  >
```

Input list must consist of positive integers:

```
sage: groups.presentation.FGAbelian([2,6,3,9,-4])
Traceback (most recent call last):
...
ValueError: input list must contain nonnegative entries
sage: groups.presentation.FGAbelian([2,'a',4])
Traceback (most recent call last):
...
TypeError: unable to convert 'a' to an integer
```

TESTS:

```
sage: ag = groups.presentation.FGAbelian([2,2])
sage: ag.as_permutation_group().is_isomorphic(groups.permutation.KleinFour())
True
sage: G = groups.presentation.FGAbelian([2,4,8])
sage: C2 = CyclicPermutationGroup(2)
sage: C4 = CyclicPermutationGroup(4)
sage: C8 = CyclicPermutationGroup(8)
sage: gg = (C2.direct_product(C4)[0]).direct_product(C8)[0]
```

```
sage: gg.is_isomorphic(G.as_permutation_group())
True
sage: all([groups.presentation.FGAbelian([i]).as_permutation_group().is_isomorphic(groups.presen
True
```

sage.groups.finitely_presented_named.**KleinFourPresentation**()

Build the Klein group of order 4 as a finitely presented group.

OUTPUT:

Klein four group ($C_2 \times C_2$) as a finitely presented group.

EXAMPLES:

```
sage: K = groups.presentation.KleinFour(); K
Finitely presented group < a, b | a^2, b^2, a^-1*b^-1*a*b >
```

sage.groups.finitely_presented_named.**QuaternionPresentation**()

Build the Quaternion group of order 8 as a finitely presented group.

OUTPUT:

Quaternion group as a finite presentation.

EXAMPLES:

```
sage: Q = groups.presentation.Quaternion(); Q
Finitely presented group < a, b | a^4, b^2*a^-2, a*b*a*b^-1 >
sage: Q.as_permutation_group().is_isomorphic(QuaternionGroup())
True
```

TESTS:

```
sage: Q = groups.presentation.Quaternion()
sage: Q.order(), Q.is_abelian()
(8, False)
sage: Q.is_isomorphic(groups.presentation.DiCyclic(2))
True
```

sage.groups.finitely_presented_named.**SymmetricPresentation**(n)

Build the Symmetric group of order $n!$ as a finitely presented group.

INPUT:

- n – The size of the underlying set of arbitrary symbols being acted on by the Symmetric group of order $n!$.

OUTPUT:

Symmetric group as a finite presentation, implementation uses GAP to find an isomorphism from a permutation representation to a finitely presented group representation. Due to this fact, the exact output presentation may not be the same for every method call on a constant n.

EXAMPLES:

```
sage: S4 = groups.presentation.Symmetric(4)
sage: S4.as_permutation_group().is_isomorphic(SymmetricGroup(4))
True
```

TESTS:

```
sage: S = [groups.presentation.Symmetric(i) for i in range(1,4)]; S[0].order()
1
sage: S[1].order(), S[2].as_permutation_group().is_isomorphic(DihedralGroup(3))
(2, True)
```

```
sage: S5 = groups.presentation.Symmetric(5)
sage: perm_S5 = S5.as_permutation_group(); perm_S5.is_isomorphic(SymmetricGroup(5))
True
sage: groups.presentation.Symmetric(8).order()
40320
```

CHAPTER

TWELVE

BRAID GROUPS

Braid groups are implemented as a particular case of finitely presented groups, but with a lot of specific methods for braids.

A braid group can be created by giving the number of strands, and the name of the generators:

```
sage: BraidGroup(3)
Braid group on 3 strands
sage: BraidGroup(3,'a')
Braid group on 3 strands
sage: BraidGroup(3,'a').gens()
(a0, a1)
sage: BraidGroup(3,'a,b').gens()
(a, b)
```

The elements can be created by operating with the generators, or by passing a list with the indices of the letters to the group:

```
sage: B.<s0,s1,s2> = BraidGroup(4)
sage: s0*s1*s0
s0*s1*s0
sage: B([1,2,1])
s0*s1*s0
```

The mapping class action of the braid group over the free group is also implemented, see `MappingClassGroupAction` for an explanation. This action is left multiplication of a free group element by a braid:

```
sage: B.<b0,b1,b2> = BraidGroup()
sage: F.<f0,f1,f2,f3> = FreeGroup()
sage: B.strands() == F.rank()   # necessary for the action to be defined
True
sage: f1 * b1
f1*f2*f1^-1
sage: f0 * b1
f0
sage: f1 * b1
f1*f2*f1^-1
sage: f1^-1 * b1
f1*f2^-1*f1^-1
```

AUTHORS:

- Miguel Angel Marco Buzunariz
- Volker Braun

- Søren Fuglede Jørgensen
- Robert Lipshitz
- Thierry Monteil: add a `__hash__` method consistent with the word problem to ensure correct Cayley graph computations.

class `sage.groups.braid.`**`Braid`**(*parent*, *x*, *check=True*)

Bases: `sage.groups.finitely_presented.FinitelyPresentedGroupElement`

Class that models elements of the braid group.

It is a particular case of element of a finitely presented group.

EXAMPLES:

```
sage: B.<s0,s1,s2> = BraidGroup(4)
sage: B
Braid group on 4 strands
sage: s0*s1/s2/s1
s0*s1*s2^-1*s1^-1
sage: B((1, 2, -3, -2))
s0*s1*s2^-1*s1^-1
```

`LKB_matrix`(*variables='x, y'*)

Return the Lawrence-Krammer-Bigelow representation matrix.

The matrix is expressed in the basis $\{e_{i,j} \mid 1 \leq i < j \leq n\}$, where the indices are ordered lexicographically. It is a matrix whose entries are in the ring of Laurent polynomials on the given variables. By default, the variables are `'x'` and `'y'`.

INPUT:

- `variables` – string (default: `'x,y'`). A string containing the names of the variables, separated by a comma.

OUTPUT:

The matrix corresponding to the Lawrence-Krammer-Bigelow representation of the braid.

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1, 2, 1])
sage: b.LKB_matrix()
[             0 -x^4*y + x^3*y         -x^4*y]
[             0         -x^3*y              0]
[        -x^2*y  x^3*y - x^2*y              0]
sage: c = B([2, 1, 2])
sage: c.LKB_matrix()
[             0 -x^4*y + x^3*y         -x^4*y]
[             0         -x^3*y              0]
[        -x^2*y  x^3*y - x^2*y              0]
```

REFERENCES:

`TL_matrix`(*drain_size*, *variab=None*, *sparse=True*)

Return the matrix representation of the Temperley–Lieb–Jones representation of the braid in a certain basis.

The basis is given by non-intersecting pairings of $(n + d)$ points, where n is the number of strands, d is given by `drain_size`, and the pairings satisfy certain rules. See `TL_basis_with_drain()` for details.

We use the convention that the eigenvalues of the standard generators are 1 and $-A^4$, where A is a variable of a Laurent polynomial ring.

When $d = n - 2$ and the variables are picked appropriately, the resulting representation is equivalent to the reduced Burau representation.

INPUT:

- `drain_size` – integer between 0 and the number of strands (both inclusive)
- `variab` – variable (default: `None`); the variable in the entries of the matrices; if `None`, then use a default variable in $\mathbf{Z}[A, A^{-1}]$
- `sparse` – boolean (default: `True`); whether or not the result should be given as a sparse matrix

OUTPUT:

The matrix of the TL representation of the braid.

The parameter `sparse` can be set to `False` if it is expected that the resulting matrix will not be sparse. We currently make no attempt at guessing this.

EXAMPLES:

Let us calculate a few examples for B_4 with $d = 0$:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, -3])
sage: b.TL_matrix(0)
[1 - A^4    -A^-2]
[   -A^6        0]
sage: R.<x> = LaurentPolynomialRing(GF(2))
sage: b.TL_matrix(0, variab=x)
[1 + x^4     x^-2]
[    x^6        0]
sage: b = B([])
sage: b.TL_matrix(0)
[1 0]
[0 1]
```

Test of one of the relations in B_8:

```
sage: B = BraidGroup(8)
sage: d = 0
sage: B([4,5,4]).TL_matrix(d) == B([5,4,5]).TL_matrix(d)
True
```

An element of the kernel of the Burau representation, following [Big99]:

```
sage: B = BraidGroup(6)
sage: psi1 = B([4, -5, -2, 1])
sage: psi2 = B([-4, 5, 5, 2, -1, -1])
sage: w1 = psi1^(-1) * B([3]) * psi1
sage: w2 = psi2^(-1) * B([3]) * psi2
sage: (w1 * w2 * w1^(-1) * w2^(-1)).TL_matrix(4)
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

REFERENCES:

alexander_polynomial (*var='t'*, *normalized=True*)

Return the Alexander polynomial of the closure of the braid.

INPUT:

- `var` – string (default: `'t'`); the name of the variable in the entries of the matrix
- `normalized` – boolean (default: `True`); whether to return the normalized Alexander polynomial

OUTPUT:

The Alexander polynomial of the braid closure of the braid.

This is computed using the reduced Burau representation. The unnormalized Alexander polynomial is a Laurent polynomial, which is only well-defined up to multiplication by plus or minus times a power of t.

We normalize the polynomial by dividing by the largest power of t and then if the resulting constant coefficient is negative, we multiply by -1.

EXAMPLES:

We first construct the trefoil:

```
sage: B = BraidGroup(3)
sage: b = B([1,2,1,2])
sage: b.alexander_polynomial(normalized=False)
1 - t + t^2
sage: b.alexander_polynomial()
t^-2 - t^-1 + 1
```

Next we construct the figure 8 knot:

```
sage: b = B([-1,2,-1,2])
sage: b.alexander_polynomial(normalized=False)
-t^-2 + 3*t^-1 - 1
sage: b.alexander_polynomial()
t^-2 - 3*t^-1 + 1
```

Our last example is the Kinoshita-Terasaka knot:

```
sage: B = BraidGroup(4)
sage: b = B([1,1,1,3,3,2,-3,-1,-1,2,-1,-3,-2])
sage: b.alexander_polynomial(normalized=False)
-t^-1
sage: b.alexander_polynomial()
1
```

REFERENCES:

- Wikipedia article Alexander_polynomial

burau_matrix (*var='t'*, *reduced=False*)

Return the Burau matrix of the braid.

INPUT:

- `var` – string (default: `'t'`); the name of the variable in the entries of the matrix
- `reduced` – boolean (default: `False`); whether to return the reduced or unreduced Burau representation

OUTPUT:

The Burau matrix of the braid. It is a matrix whose entries are Laurent polynomials in the variable `var`. If `reduced` is `True`, return the matrix for the reduced Burau representation instead.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: B.inject_variables()
Defining s0, s1, s2
sage: b = s0*s1/s2/s1
sage: b.burau_matrix()
[       1 - t            0      t - t^2          t^2]
[           1            0            0            0]
[           0            0            1            0]
[           0         t^-2 -t^-2 + t^-1    -t^-1 + 1]
sage: s2.burau_matrix('x')
[    1     0     0     0]
[    0     1     0     0]
[    0     0 1 - x     x]
[    0     0     1     0]
sage: s0.burau_matrix(reduced=True)
[-t  0  0]
[-t  1  0]
[-t  0  1]
```

REFERENCES:

- Wikipedia article Burau_representation

components_in_closure()

Return the number of components of the trace closure of the braid.

OUTPUT:

Positive integer.

EXAMPLES:

```
sage: B = BraidGroup(5)
sage: b = B([1, -3])  # Three disjoint unknots
sage: b.components_in_closure()
3
sage: b = B([1, 2, 3, 4])  # The unknot
sage: b.components_in_closure()
1
sage: B = BraidGroup(4)
sage: K11n42 = B([1, -2, 3, -2, 3, -2, -2, -1, 2, -3, -3, 2, 2])
sage: K11n42.components_in_closure()
1
```

exponent_sum()

Return the exponent sum of the braid.

OUTPUT:

Integer.

EXAMPLES:

```
sage: B = BraidGroup(5)
sage: b = B([1, 4, -3, 2])
sage: b.exponent_sum()
2
sage: b = B([])
sage: b.exponent_sum()
0
```

jones_polynomial(*variab=None*, *skein_normalization=False*)

Return the Jones polynomial of the trace closure of the braid.

The normalization is so that the unknot has Jones polynomial 1. If `skein_normalization` is `True`, the variable of the result is replaced by a itself to the power of 4, so that the result agrees with the conventions of [Lic] (which in particular differs slightly from the conventions used otherwise in this class), had one used the conventional Kauffman bracket variable notation directly.

If `variab` is `None` return a polynomial in the variable A or t, depending on the value `skein_normalization`. In particular, if `skein_normalization` is `False`, return the result in terms of the variable t, also used in [Lic].

INPUT:

- `variab` – variable (default: `None`); the variable in the resulting polynomial; if unspecified, use either a default variable in $ZZ[A, A^{-1}]$ or the variable t in the symbolic ring
- `skein_normalization` – boolean (default: `False`); determines the variable of the resulting polynomial

OUTPUT:

If `skein_normalization` if `False`, this returns an element in the symbolic ring as the Jones polynomial of the closure might have fractional powers when the closure of the braid is not a knot. Otherwise the result is a Laurent polynomial in `variab`.

EXAMPLES:

The unknot:

```
sage: B = BraidGroup(9)
sage: b = B([1, 2, 3, 4, 5, 6, 7, 8])
sage: b.jones_polynomial()
1
```

Two unlinked unknots:

```
sage: B = BraidGroup(2)
sage: b = B([])
sage: b.jones_polynomial()
-sqrt(t) - 1/sqrt(t)
```

The Hopf link:

```
sage: B = BraidGroup(2)
sage: b = B([-1,-1])
sage: b.jones_polynomial()
-1/sqrt(t) - 1/t^(5/2)
```

Different representations of the trefoil and one of its mirror:

```
sage: B = BraidGroup(2)
sage: b = B([-1, -1, -1])
sage: b.jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
sage: b.jones_polynomial()
1/t + 1/t^3 - 1/t^4
sage: B = BraidGroup(3)
sage: b = B([-1, -2, -1, -2])
sage: b.jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
sage: R.<x> = LaurentPolynomialRing(GF(2))
sage: b.jones_polynomial(skein_normalization=True, variab=x)
```

```
x^-16 + x^-12 + x^-4
sage: B = BraidGroup(3)
sage: b = B([1, 2, 1, 2])
sage: b.jones_polynomial(skein_normalization=True)
A^4 + A^12 - A^16
```

K11n42 (the mirror of the “Kinoshita-Terasaka” knot) and K11n34 (the mirror of the “Conway” knot):

```
sage: B = BraidGroup(4)
sage: b11n42 = B([1, -2, 3, -2, 3, -2, -2, -1, 2, -3, -3, 2, 2])
sage: b11n34 = B([1, 1, 2, -3, 2, -3, 1, -2, -2, -3, -3])
sage: cmp(b11n42.jones_polynomial(), b11n34.jones_polynomial())
0
```

REFERENCES:

left_normal_form()

Return the left normal form of the braid.

OUTPUT:

A tuple of braid generators in the left normal form. The first element is a power of Δ, and the rest are permutation braids.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 3, -1, 2, -3])
sage: b.left_normal_form()
(s0^-1*s1^-1*s2^-1*s0^-1*s1^-1*s0^-1, s0*s1*s2*s1*s0, s0*s2*s1)
sage: c = B([1])
sage: c.left_normal_form()
(1, s0)
```

markov_trace(*variab=None*, *normalized=True*)

Return the Markov trace of the braid.

The normalization is so that in the underlying braid group representation, the eigenvalues of the standard generators of the braid group are 1 and $-A^4$.

INPUT:

- `variab` – variable (default: `None`); the variable in the resulting polynomial; if `None`, then use the variable A in $\mathbf{Z}[A, A^{-1}]$
- `normalized` - boolean (default: `True`); if specified to be `False`, return instead a rescaled Laurent polynomial version of the Markov trace

OUTPUT:

If `normalized` is `False`, return instead the Markov trace of the braid, normalized by a factor of $(A^2 + A^{-2})^n$. The result is then a Laurent polynomial in `variab`. Otherwise it is a quotient of Laurent polynomials in `variab`.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, -3])
sage: mt = b.markov_trace(); mt
A^4/(A^12 + 3*A^8 + 3*A^4 + 1)
sage: mt.factor()
A^4 * (A^4 + 1)^-3
```

We now give the non-normalized Markov trace:

```
sage: mt = b.markov_trace(normalized=False); mt
A^-4 + 1
sage: mt.parent()
Univariate Laurent Polynomial Ring in A over Integer Ring
```

permutation()

Return the permutation induced by the braid in its strands.

OUTPUT:

A permutation.

EXAMPLES:

```
sage: B.<s0,s1,s2> = BraidGroup()
sage: b = s0*s1/s2/s1
sage: b.permutation()
[4, 1, 3, 2]
sage: b.permutation().cycle_string()
'(1,4,2)'
```

plot(*color='rainbow'*, *orientation='bottom-top'*, *gap=0.05*, *aspect_ratio=1*, *axes=False*, ***kwds*)

Plot the braid

The following options are available:

- `color` – (default: `'rainbow'`) the color of the strands. Possible values are:
  - `'rainbow'`, uses `rainbow()` according to the number of strands.
  - a valid color name for `bezier_path()` and `line()`. Used for all strands.
  - a list or a tuple of colors for each individual strand.
- `orientation` – (default: `'bottom-top'`) determines how the braid is printed. The possible values are:
  - `'bottom-top'`, the braid is printed from bottom to top
  - `'top-bottom'`, the braid is printed from top to bottom
  - `'left-right'`, the braid is printed from left to right
- `gap` – floating point number (default: 0.05). determines the size of the gap left when a strand goes under another.
- `aspect_ratio` – floating point number (default: `1`). The aspect ratio.
- `**kwds` – other keyword options that are passed to `bezier_path()` and `line()`.

EXAMPLES:

```
sage: B = BraidGroup(4, 's')
sage: b = B([1, 2, 3, 1, 2, 1])
sage: b.plot()
Graphics object consisting of 30 graphics primitives
sage: b.plot(color=["red", "blue", "red", "blue"])
Graphics object consisting of 30 graphics primitives

sage: B.<s,t> = BraidGroup(3)
sage: b = t^-1*s^2
sage: b.plot(orientation="left-right", color="red")
Graphics object consisting of 12 graphics primitives
```

plot3d(*color='rainbow'*)
Plots the braid in 3d.

The following option is available:

- color – (default: 'rainbow') the color of the strands. Possible values are:
  - 'rainbow', uses rainbow() according to the number of strands.
  - a valid color name for bezier3d(). Used for all strands.
  - a list or a tuple of colors for each individual strand.

EXAMPLES:

```
sage: B = BraidGroup(4, 's')
sage: b = B([1, 2, 3, 1, 2, 1])
sage: b.plot3d()
Graphics3d Object
sage: b.plot3d(color="red")
Graphics3d Object
sage: b.plot3d(color=["red", "blue", "red", "blue"])
Graphics3d Object
```

strands()
Return the number of strands in the braid.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, -1, 3, -2])
sage: b.strands()
4
```

tropical_coordinates()
Return the tropical coordinates of self in the braid group B_n.

OUTPUT:

- a list of $2n$ tropical integers

EXAMPLES:

```
sage: B = BraidGroup(3)
sage: b = B([1])
sage: tc = b.tropical_coordinates(); tc
[1, 0, 0, 2, 0, 1]
sage: tc[0].parent()
Tropical semiring over Integer Ring

sage: b = B([-2, -2, -1, -1, 2, 2, 1, 1])
sage: b.tropical_coordinates()
[1, -19, -12, 9, 0, 13]
```

REFERENCES:

sage.groups.braid.**BraidGroup**(*n=None*, *names='s'*)
Construct a Braid Group

INPUT:

- n – integer or None (default). The number of strands. If not specified the names are counted and the group is assumed to have one more strand than generators.
- names – string or list/tuple/iterable of strings (default: 'x'). The generator names or name prefix.

EXAMPLES:

```
sage: B.<a,b> = BraidGroup();  B
Braid group on 3 strands
sage: H = BraidGroup('a, b')
sage: B is H
True
sage: BraidGroup(3)
Braid group on 3 strands
```

The entry can be either a string with the names of the generators, or the number of generators and the prefix of the names to be given. The default prefix is `'s'`

```
sage: B=BraidGroup(3); B.generators()
(s0, s1)
sage: BraidGroup(3, 'g').generators()
(g0, g1)
```

Since the word problem for the braid groups is solvable, their Cayley graph can be localy obtained as follows (see trac ticket #16059):

```
sage: def ball(group, radius):
....:     ret = set()
....:     ret.add(group.one())
....:     for length in range(1, radius):
....:         for w in Words(alphabet=group.gens(), length=length):
....:              ret.add(prod(w))
....:     return ret
sage: B = BraidGroup(4)
sage: GB = B.cayley_graph(elements=ball(B, 4), generators=B.gens()); GB
Digraph on 31 vertices
```

Since the braid group has nontrivial relations, this graph contains less vertices than the one associated to the free group (which is a tree):

```
sage: F = FreeGroup(3)
sage: GF = F.cayley_graph(elements=ball(F, 4), generators=F.gens()); GF
Digraph on 40 vertices
```

TESTS:

```
sage: G1 = BraidGroup(3, 'a,b')
sage: G2 = BraidGroup('a,b')
sage: G3.<a,b> = BraidGroup()
sage: G1 is G2, G2 is G3
(True, True)
```

class `sage.groups.braid.`**`BraidGroup_class`**(*names*)

Bases: `sage.groups.finitely_presented.FinitelyPresentedGroup`

The braid group on n strands.

EXAMPLES:

```
sage: B1 = BraidGroup(5)
sage: B1
Braid group on 5 strands
sage: B2 = BraidGroup(3)
sage: B1==B2
False
sage: B2 is BraidGroup(3)
True
```

Element
alias of Braid

TL_basis_with_drain (*drain_size*)
Return a basis of a summand of the Temperley–Lieb–Jones representation of self.

The basis elements are given by non-intersecting pairings of $n+d$ points in a square with n points marked 'on the top' and d points 'on the bottom' so that every bottom point is paired with a top point. Here, n is the number of strands of the braid group, and d is specified by drain_size.

A basis element is specified as a list of integers obtained by considering the pairings as obtained as the 'highest term' of trivalent trees marked by Jones–Wenzl projectors (see e.g. [Wan10]). In practice, this is a list of non-negative integers whose first element is drain_size, whose last element is 0, and satisfying that consecutive integers have difference 1. Moreover, the length of each basis element is $n+1$.

Given these rules, the list of lists is constructed recursively in the natural way.

INPUT:

- drain_size – integer between 0 and the number of strands (both inclusive)

OUTPUT:

A list of basis elements, each of which is a list of integers.

EXAMPLES:

We calculate the basis for the appropriate vector space for B_5 when $d = 3$:

```
sage: B = BraidGroup(5)
sage: B.TL_basis_with_drain(3)
[[3, 4, 3, 2, 1, 0],
 [3, 2, 3, 2, 1, 0],
 [3, 2, 1, 2, 1, 0],
 [3, 2, 1, 0, 1, 0]]
```

The number of basis elements hopefully correponds to the general formula for the dimension of the representation spaces:

```
sage: B = BraidGroup(10)
sage: d = 2
sage: B.dimension_of_TL_space(d) == len(B.TL_basis_with_drain(d))
True
```

REFERENCES:

TL_representation (*drain_size*, *variab=None*)
Return representation matrices of the Temperley–Lieb–Jones representation of standard braid group generators and inverses of self.

The basis is given by non-intersecting pairings of $(n+d)$ points, where n is the number of strands, and d is given by drain_size, and the pairings satisfy certain rules. See TL_basis_with_drain() for details. This basis has the useful property that all resulting entries can be regarded as Laurent polynomials.

We use the convention that the eigenvalues of the standard generators are 1 and $-A^4$, where A is the generator of the Laurent polynomial ring.

When $d = n - 2$ and the variables are picked appropriately, the resulting representation is equivalent to the reduced Burau representation. When $d = n$, the resulting representation is trivial and 1-dimensional.

INPUT:

- `drain_size` – integer between 0 and the number of strands (both inclusive)
- `variab` – variable (default: `None`); the variable in the entries of the matrices; if `None`, then use a default variable in $\mathbf{Z}[A, A^{-1}]$

OUTPUT:

A list of matrices corresponding to the representations of each of the standard generators and their inverses.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: B.TL_representation(0)
[(
  [   1    0]  [    1     0]
  [ A^2 -A^4], [ A^-2 -A^-4]
),
 (
  [-A^4  A^2]  [-A^-4  A^-2]
  [   0    1], [    0     1]
),
 (
  [   1    0]  [    1     0]
  [ A^2 -A^4], [ A^-2 -A^-4]
)]
sage: R.<A> = LaurentPolynomialRing(GF(2))
sage: B.TL_representation(0, variab=A)
[(
  [  1   0]  [   1    0]
  [A^2 A^4], [A^-2 A^-4]
),
 (
  [A^4 A^2]  [A^-4 A^-2]
  [  0   1], [   0    1]
),
 (
  [  1   0]  [   1    0]
  [A^2 A^4], [A^-2 A^-4]
)]
sage: B = BraidGroup(8)
sage: B.TL_representation(8)
[([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1]),
 ([1], [1])]
```

an_element()

Return an element of the braid group.

This is used both for illustration and testing purposes.

EXAMPLES:

```
sage: B=BraidGroup(2)
sage: B.an_element()
s
```

as_permutation_group()

Return an isomorphic permutation group.

OUTPUT:

Raises a `ValueError` error since braid groups are infinite.

TESTS:

```
sage: B = BraidGroup(4, 'g')
sage: B.as_permutation_group()
Traceback (most recent call last):
...
ValueError: the group is infinite
```

cardinality()

Return the number of group elements.

OUTPUT:

Infinity.

TESTS:

```
sage: B1 = BraidGroup(5)
sage: B1.cardinality()
+Infinity
```

dimension_of_TL_space(*drain_size*)

Return the dimension of a particular Templerley–Lieb representation summand of `self`.

Following the notation of `TL_basis_with_drain()`, the summand is the one corresponding to the number of drains being fixed to be `drain_size`.

INPUT:

- `drain_size` – integer between 0 and the number of strands (both inclusive)

EXAMPLES:

Calculation of the dimension of the representation of B_8 corresponding to having 2 drains:

```
sage: B = BraidGroup(8)
sage: B.dimension_of_TL_space(2)
28
```

The direct sum of endomorphism spaces of these vector spaces make up the entire Temperley–Lieb algebra:

```
sage: import sage.combinat.diagram_algebras as da
sage: B = BraidGroup(6)
sage: dimensions = [B.dimension_of_TL_space(d)**2 for d in [0, 2, 4, 6]]
sage: total_dim = sum(dimensions)
sage: total_dim == len(list(da.temperley_lieb_diagrams(6)))
True
```

mapping_class_action(*F*)

Return the action of self in the free group F as mapping class group.

This action corresponds to the action of the braid over the punctured disk, whose fundamental group is the free group on as many generators as strands.

In Sage, this action is the result of multiplying a free group element with a braid. So you generally do not have to construct this action yourself.

OUTPUT:

A `MappingClassGroupAction`.

EXAMPLES

```
sage: B = BraidGroup(3)
sage: B.inject_variables()
Defining s0, s1
sage: F.<a,b,c> = FreeGroup(3)
sage: A = B.mapping_class_action(F)
sage: A(a,s0)
a*b*a^-1
sage: a * s0    # simpler notation
a*b*a^-1
```

order()

Return the number of group elements.

OUTPUT:

Infinity.

TESTS:

```
sage: B1 = BraidGroup(5)
sage: B1.cardinality()
+Infinity
```

some_elements()

Return a list of some elements of the braid group.

This is used both for illustration and testing purposes.

EXAMPLES:

```
sage: B=BraidGroup(3)
sage: B.some_elements()
[s0, s0*s1, (s0*s1)^3]
```

strands()

Return the number of strands.

OUTPUT:

Integer.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: B.strands()
4
```

class sage.groups.braid.**MappingClassGroupAction**(*G*, *M*, *is_left=0*)

Bases: sage.categories.action.Action

The action of the braid group the free group as the mapping class group of the punctured disk.

That is, this action is the action of the braid over the punctured disk, whose fundamental group is the free group on as many generators as strands.

This action is defined as follows:

$$x_j \cdot \sigma_i = \begin{cases} x_j \cdot x_{j+1} \cdot {x_j}^{-1} & \text{if } i = j \\ x_{j-1} & \text{if } i = j-1 \\ x_j & \text{otherwise} \end{cases},$$

where σ_i are the generators of the braid group on n strands, and x_j the generators of the free group of rank n.

You should left multiplication of the free group element by the braid to compute the action. Alternatively, use the `mapping_class_action()` method of the braid group to constuct this action.

EXAMPLES:

```
sage: B.<s0,s1,s2> = BraidGroup(4)
sage: F.<x0,x1,x2,x3> = FreeGroup(4)
sage: x0 * s1
x0
sage: x1 * s1
x1*x2*x1^-1
sage: x1^-1 * s1
x1*x2^-1*x1^-1

sage: A = B.mapping_class_action(F)
sage: A
Right action by Braid group on 4 strands on Free Group on generators {x0, x1, x2, x3}
sage: A(x0, s1)
x0
sage: A(x1, s1)
x1*x2*x1^-1
sage: A(x1^-1, s1)
x1*x2^-1*x1^-1
```

CHAPTER

THIRTEEN

INDEXED FREE GROUPS

Free groups and free abelian groups implemented using an indexed set of generators.

AUTHORS:

- Travis Scrimshaw (2013-10-16): Initial version

class `sage.groups.indexed_free_group.`**`IndexedFreeAbelianGroup`**(*indices*, *prefix*, *category=None*, ***kwds*)

Bases: `sage.groups.indexed_free_group.IndexedGroup`, `sage.groups.group.AbelianGroup`

An indexed free abelian group.

EXAMPLES:

```
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G
Free abelian group indexed by Integer Ring
sage: G = Groups().Commutative().free(index_set='abcde')
sage: G
Free abelian group indexed by {'a', 'b', 'c', 'd', 'e'}
```

class **`Element`**(*F*, *x*)

Bases: `sage.monoids.indexed_free_monoid.IndexedFreeAbelianMonoidElement`, `sage.groups.indexed_free_group.IndexedFreeGroup.Element`

Create the element `x` of an indexed free abelian monoid `F`.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(index_set=ZZ)
sage: x = F([(0, 1), (2, 2), (-1, 2)])
sage: y = F({0:1, 2:2, -1:2})
sage: z = F(reversed([(0, 1), (2, 2), (-1, 2)]))
sage: x == y and y == z
True
sage: TestSuite(x).run()
```

`IndexedFreeAbelianGroup.`**`gen`**(*x*)

The generator indexed by `x` of `self`.

EXAMPLES:

```
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.gen(0)
F[0]
sage: G.gen(2)
F[2]
```

IndexedFreeAbelianGroup.**one**()

Return the identity element of self.

EXAMPLES:

```
sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.one()
1
```

class sage.groups.indexed_free_group.**IndexedFreeGroup**(*indices*, *prefix*, *category=None*, ***kwds*)

Bases: sage.groups.indexed_free_group.IndexedGroup, sage.groups.group.Group

An indexed free group.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: G
Free group indexed by Integer Ring
sage: G = Groups().free(index_set='abcde')
sage: G
Free group indexed by {'a', 'b', 'c', 'd', 'e'}
```

class Element(*F*, *x*)

Bases: sage.monoids.indexed_free_monoid.IndexedFreeMonoidElement

Create the element x of an indexed free abelian monoid F.

EXAMPLES:

```
sage: F = FreeMonoid(index_set=tuple('abcde'))
sage: x = F( [(1, 2), (0, 1), (3, 2), (0, 1)] )
sage: y = F( ((1, 2), (0, 1), [3, 2], [0, 1]) )
sage: z = F( reversed([(0, 1), (3, 2), (0, 1), (1, 2)]) )
sage: x == y and y == z
True
sage: TestSuite(x).run()
```

length()

Return the length of self.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: a,b,c,d,e = [G.gen(i) for i in range(5)]
sage: elt = a*c^-3*b^-2*a
sage: elt.length()
7
sage: len(elt)
7
```

```
sage: G = Groups().free(index_set=ZZ)
sage: a,b,c,d,e = [G.gen(i) for i in range(5)]
sage: elt = a*c^-3*b^-2*a
sage: elt.length()
7
sage: len(elt)
7
```

to_word_list()

Return self as a word represented as a list whose entries are the pairs (i, s) where i is the index and s is the sign.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: a,b,c,d,e = [G.gen(i) for i in range(5)]
sage: x = a*b^2*e*a^-1
sage: x.to_word_list()
[(0, 1), (1, 1), (1, 1), (4, 1), (0, -1)]
```

`IndexedFreeGroup.gen(x)`

The generator indexed by `x` of `self`.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: G.gen(0)
F[0]
sage: G.gen(2)
F[2]
```

`IndexedFreeGroup.one()`

Return the identity element of `self`.

EXAMPLES:

```
sage: G = Groups().free(ZZ)
sage: G.one()
1
```

class `sage.groups.indexed_free_group.IndexedGroup`(*indices*, *prefix*, *category=None*, *names=None*, ***kwds*)

Bases: `sage.monoids.indexed_free_monoid.IndexedMonoid`

Base class for free (abelian) groups whose generators are indexed by a set.

TESTS:

We check finite properties:

```
sage: G = Groups().free(index_set=ZZ)
sage: G.is_finite()
False
sage: G = Groups().free(index_set='abc')
sage: G.is_finite()
False
sage: G = Groups().free(index_set=[])
sage: G.is_finite()
True

sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.is_finite()
False
sage: G = Groups().Commutative().free(index_set='abc')
sage: G.is_finite()
False
sage: G = Groups().Commutative().free(index_set=[])
sage: G.is_finite()
True
```

`gens()`

Return the group generators of `self`.

EXAMPLES:

```
sage: G = Groups.free(index_set=ZZ)
sage: G.group_generators()
Lazy family (Generator map from Integer Ring to
 Free group indexed by Integer Ring(i))_{i in Integer Ring}
sage: G = Groups().free(index_set='abcde')
sage: sorted(G.group_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

group_generators()

Return the group generators of `self`.

EXAMPLES:

```
sage: G = Groups.free(index_set=ZZ)
sage: G.group_generators()
Lazy family (Generator map from Integer Ring to
 Free group indexed by Integer Ring(i))_{i in Integer Ring}
sage: G = Groups().free(index_set='abcde')
sage: sorted(G.group_generators())
[F['a'], F['b'], F['c'], F['d'], F['e']]
```

order()

Return the number of elements of `self`, which is ∞ unless this is the trivial group.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: G.order()
+Infinity
sage: G = Groups().Commutative().free(index_set='abc')
sage: G.order()
+Infinity
sage: G = Groups().Commutative().free(index_set=[])
sage: G.order()
1
```

rank()

Return the rank of `self`.

This is the number of generators of `self`.

EXAMPLES:

```
sage: G = Groups().free(index_set=ZZ)
sage: G.rank()
+Infinity
sage: G = Groups().free(index_set='abc')
sage: G.rank()
3
sage: G = Groups().free(index_set=[])
sage: G.rank()
0

sage: G = Groups().Commutative().free(index_set=ZZ)
sage: G.rank()
+Infinity
sage: G = Groups().Commutative().free(index_set='abc')
sage: G.rank()
3
sage: G = Groups().Commutative().free(index_set=[])
```

```
sage: G.rank()
0
```

CHAPTER

FOURTEEN

RIGHT-ANGLED ARTIN GROUPS

A *right-angled Artin group* (often abbrivated as RAAG) is a group which has a presentation whose only relations are commutators between generators. These are also known as graph groups, since they are (uniquely) encoded by (simple) graphs, or partially commmutative groups.

AUTHORS:

- Travis Scrimshaw (2013-09-01): Initial version

class `sage.groups.raag.`**`RightAngledArtinGroup`**`(`*G*`)`

Bases: `sage.groups.finitely_presented.FinitelyPresentedGroup`

The right-angled Artin group defined by a graph G.

Let $\Gamma = \{V(\Gamma), E(\Gamma)\}$ be a simple graph. A *right-angled Artin group* (commonly abbriated as RAAG) is the group

$$A_\Gamma = \langle g_v : v \in V(\Gamma) \mid [g_u, g_v] \text{ if } \{u, v\} \notin E(\Gamma) \rangle.$$

These are sometimes known as graph groups or partitally commutative groups. This RAAG's contains both free groups, given by the complete graphs, and free abelian groups, given by disjoint vertices.

> **Warning:** This is the opposite convention of some papers.

Right-angled Artin groups contain many remarkable properties and have a very rich structure despite their simple presentation. Here are some known facts:

- The word problem is solvable.
- They are known to be rigid; that is for any finite simple graphs Δ and Γ, we have $A_\Delta \cong A_\Gamma$ if and only if $\Delta \cong \Gamma$ [Droms1987].
- They embed as a finite index subgroup of a right-angled Coxeter group (which is the same definition as above except with the additional relations $g_v^2 = 1$ for all $v \in V(\Gamma)$).
- In [BB1997], it was shown they contain subgroups that statisfy the property FP_2 but are not finitely presented by considering the kernal of $\phi : A_\Gamma \to \mathbf{Z}$ by $g_v \mapsto 1$ (i.e. words of exponent sum 0).
- A_Γ has a finite $K(\pi, 1)$ space.
- A_Γ acts freely and cocompactly on a finite dimensional $CAT(0)$ space, and so it is biautomatic.
- Given an Artin group B with generators s_i, then any subgroup generated by a collection of $v_i = s_i^{k_i}$ where $k_i \geq 2$ is a RAAG where $[v_i, v_j] = 1$ if and only if $[s_i, s_j] = 1$ [CP2001].

The normal forms for RAAG's in Sage are those described in [VW1994] and gathers commuting groups together.

EXAMPLES:

```
sage: Gamma = Graph(4)
sage: G = RightAngledArtinGroup(Gamma)
sage: a,b,c,d = G.gens()
sage: a*c*d^4*a^-3*b
v0^-2*v1*v2*v3^4

sage: Gamma = graphs.CompleteGraph(4)
sage: G = RightAngledArtinGroup(Gamma)
sage: a,b,c,d = G.gens()
sage: a*c*d^4*a^-3*b
v0*v2*v3^4*v0^-3*v1

sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G
Right-angled Artin group of Cycle graph
sage: a,b,c,d,e = G.gens()
sage: e^-1*c*b*e*b^-1*c^-4
v2^-3
```

REFERENCES:

- Wikipedia article Artin_group#Right-angled_Artin_groups

class Element (*parent*, *lst*)

Bases: `sage.groups.finitely_presented.FinitelyPresentedGroupElement`

An element of a right-angled Artin group (RAAG).

Elements of RAAGs are modeled as lists of pairs `[i, p]` where `i` is the index of a vertex in the defining graph (with some fixed order of the vertices) and `p` is the power.

`RightAngledArtinGroup.`**`as_permutation_group`**`()`

Raise a `ValueError` error since right-angled Artin groups are infinite, so they have no isomorphic permutation group.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.as_permutation_group()
Traceback (most recent call last):
...
ValueError: the group is infinite
```

`RightAngledArtinGroup.`**`cardinality`**`()`

Return the number of group elements.

OUTPUT:

Infinity.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.cardinality()
+Infinity
```

`RightAngledArtinGroup.`**`gen`**`(i)`

Return the `i`-th generator of `self`.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.gen(2)
v2
```

RightAngledArtinGroup.**gens**()

Return the generators of `self`.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.gens()
(v0, v1, v2, v3, v4)
sage: Gamma = Graph([('x', 'y'), ('y', 'zeta')])
sage: G = RightAngledArtinGroup(Gamma)
sage: G.gens()
(vx, vy, vzeta)
```

RightAngledArtinGroup.**graph**()

Return the defining graph of `self`.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.graph()
Cycle graph: Graph on 5 vertices
```

RightAngledArtinGroup.**ngens**()

Return the number of generators of `self`.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.ngens()
5
```

RightAngledArtinGroup.**one**()

Return the identity element 1.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.one()
1
```

RightAngledArtinGroup.**one_element**()

Return the identity element 1.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.one()
1
```

RightAngledArtinGroup.**order**()

Return the number of group elements.

OUTPUT:

Infinity.

EXAMPLES:

```
sage: Gamma = graphs.CycleGraph(5)
sage: G = RightAngledArtinGroup(Gamma)
sage: G.cardinality()
+Infinity
```

CHAPTER

FIFTEEN

FUNCTOR THAT CONVERTS A COMMUTATIVE ADDITIVE GROUP INTO A MULTIPLICATIVE GROUP.

AUTHORS:

- Mark Shimozono (2013): initial version

class `sage.groups.group_exp.`**`GroupExp`**

Bases: `sage.categories.functor.Functor`

A functor that converts a commutative additive group into an isomorphic multiplicative group.

More precisely, given a commutative additive group G, define the exponential of G to be the isomorphic group with elements denoted e^g for every $g \in G$ and but with product in multiplicative notation

$$e^g e^h = e^{g+h} \qquad \text{for all } g, h \in G.$$

The class `GroupExp` implements the sage functor which sends a commutative additive group G to its exponential.

The creation of an instance of the functor `GroupExp` requires no input:

```
sage: E = GroupExp(); E
Functor from Category of commutative additive groups to Category of groups
```

The `GroupExp` functor (denoted E in the examples) can be applied to two kinds of input. The first is a commutative additive group. The output is its exponential. This is accomplished by `_apply_functor()`:

```
sage: EZ = E(ZZ); EZ
Multiplicative form of Integer Ring
```

Elements of the exponentiated group can be created and manipulated as follows:

```
sage: x = EZ(-3); x
-3
sage: x.parent()
Multiplicative form of Integer Ring
sage: EZ(-1)*EZ(6) == EZ(5)
True
sage: EZ(3)^(-1)
-3
sage: EZ.one()
0
```

The second kind of input the `GroupExp` functor accepts, is a homomorphism of commutative additive groups. The output is the multiplicative form of the homomorphism. This is achieved by `_apply_functor_to_morphism()`:

```
sage: L = RootSystem(['A',2]).ambient_space()
sage: EL = E(L)
sage: W = L.weyl_group(prefix="s")
sage: s2 = W.simple_reflection(2)
sage: def my_action(mu):
....:     return s2.action(mu)
sage: from sage.categories.morphism import SetMorphism
sage: from sage.categories.homset import Hom
sage: f = SetMorphism(Hom(L,L,CommutativeAdditiveGroups()), my_action)
sage: F = E(f); F
Generic endomorphism of Multiplicative form of Ambient space of the Root system of type ['A', 2]
sage: v = L.an_element(); v
(2, 2, 3)
sage: y = F(EL(v)); y
(2, 3, 2)
sage: y.parent()
Multiplicative form of Ambient space of the Root system of type ['A', 2]
```

class sage.groups.group_exp.**GroupExpElement**(*parent*, *x*)

Bases: sage.structure.element_wrapper.ElementWrapper, sage.structure.element.MultiplicativeGroupElement

An element in the exponential of a commutative additive group.

INPUT:

- self – the exponentiated group element being created
- parent – the exponential group (parent of self)
- x – the commutative additive group element being wrapped to form self.

EXAMPLES:

```
sage: G = QQ^2
sage: EG = GroupExp()(G)
sage: z = GroupExpElement(EG, vector(QQ, (1,-3))); z
(1, -3)
sage: z.parent()
Multiplicative form of Vector space of dimension 2 over Rational Field
sage: EG(vector(QQ,(1,-3)))==z
True
```

inverse()

Invert the element self.

EXAMPLES:

```
sage: EZ = GroupExp()(ZZ)
sage: EZ(-3).inverse()
3
```

class sage.groups.group_exp.**GroupExp_Class**(G)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

The multiplicative form of a commutative additive group.

INPUT:

- G: a commutative additive group

OUTPUT:

- The multiplicative form of G.

EXAMPLES:

```
sage: GroupExp()(QQ)
Multiplicative form of Rational Field
```

Element
alias of GroupExpElement

an_element()
Return an element of the multiplicative group.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).weight_lattice()
sage: EL = GroupExp()(L)
sage: x = EL.an_element(); x
2*Lambda[1] + 2*Lambda[2]
sage: x.parent()
Multiplicative form of Weight lattice of the Root system of type ['A', 2]
```

group_generators()
Return generators of `self`.

EXAMPLES:

```
sage: GroupExp()(ZZ).group_generators()
(1,)
```

one()
Return the identity element of the multiplicative group.

EXAMPLES:

```
sage: G = GroupExp()(ZZ^2)
sage: G.one()
(0, 0)
sage: x = G.an_element(); x
(1, 0)
sage: x == x * G.one()
True
```

product(x, y)
Return the product of x and y in the multiplicative group.

EXAMPLES:

```
sage: G = GroupExp()(ZZ)
sage: G.product(G(2),G(7))
9
sage: x = G(2)
sage: x.__mul__(G(7))
9
```

CHAPTER

SIXTEEN

SEMIDIRECT PRODUCT OF GROUPS

AUTHORS:

- Mark Shimozono (2013) initial version

class `sage.groups.group_semidirect_product.`**`GroupSemidirectProduct`**(*G*, *H*, *twist=None*, *act_to_right=True*, *prefix0=None*, *prefix1=None*, *print_tuple=False*, *category=Category of groups*)

Bases: `sage.sets.cartesian_product.CartesianProduct`

Return the semidirect product of the groups `G` and `H` using the homomorphism `twist`.

INPUT:

- `G` and `H` – multiplicative groups
- `twist` – (default: None) a function defining a homomorphism (see below)
- `act_to_right` – True or False (default: True)
- `prefix0` – (default: None) optional string
- `prefix1` – (default: None) optional string
- `print_tuple` – True or False (default: False)
- `category` – A category (default: Groups())

A semidirect product of groups G and H is a group structure on the Cartesian product $G \times H$ whose product agrees with that of G on $G \times 1_H$ and with that of H on $1_G \times H$, such that either $1_G \times H$ or $G \times 1_H$ is a normal subgroup. In the former case the group is denoted $G \ltimes H$ and in the latter, $G \rtimes H$.

If `act_to_right` is True, this indicates the group $G \ltimes H$ in which G acts on H by automorphisms. In this case there is a group homomorphism $\phi \in \mathrm{Hom}(G, \mathrm{Aut}(H))$ such that

$$ghg^{-1} = \phi(g)(h).$$

The homomorphism ϕ is specified by the input `twist`, which syntactically is the function $G \times H \to H$ defined by

$$twist(g, h) = \phi(g)(h).$$

The product on $G \ltimes H$ is defined by

$$\begin{aligned}(g_1, h_1)(g_2, h_2) &= g_1 h_1 g_2 h_2 \\ &= g_1 g_2 g_2^{-1} h_1 g_2 h_2 \\ &= (g_1 g_2, twist(g_2^{-1}, h_1) h_2)\end{aligned}$$

If act_to_right is False, the group $G \rtimes H$ is specified by a homomorphism $\psi \in \mathrm{Hom}(H, \mathrm{Aut}(G))$ such that

$$hgh^{-1} = \psi(h)(g)$$

Then twist is the function $H \times G \to G$ defined by

$$twist(h, g) = \psi(h)(g).$$

so that the product in $G \rtimes H$ is defined by

$$\begin{aligned}(g_1, h_1)(g_2, h_2) &= g_1 h_1 g_2 h_2 \\ &= g_1 h_1 g_2 h_1^{-1} h_1 h_2 \\ &= (g_1 twist(h_1, g_2), h_1 h_2)\end{aligned}$$

If prefix0 (resp. prefixl) is not None then it is used as a wrapper for printing elements of G (resp. H). If print_tuple is True then elements are printed in the style (g, h) and otherwise in the style $g * h$.

EXAMPLES:

```
sage: G = GL(2,QQ)
sage: V = QQ^2
sage: EV = GroupExp()(V) # make a multiplicative version of V
sage: def twist(g,v):
....:     return EV(g*v.value)
sage: H = GroupSemidirectProduct(G, EV, twist=twist, prefix1 = 't'); H
Semidirect product of General Linear Group of degree 2 over Rational Field acting on Multiplicat
sage: x = H.an_element(); x
t[(1, 0)]
sage: x^2
t[(2, 0)]
sage: cartan_type = CartanType(['A',2])
sage: W = WeylGroup(cartan_type, prefix="s")
sage: def twist(w,v):
....:     return w*v*(~w)
sage: WW = GroupSemidirectProduct(W,W, twist=twist, print_tuple=True)
sage: s = Family(cartan_type.index_set(), lambda i: W.simple_reflection(i))
sage: y = WW((s[1],s[2])); y
(s1, s2)
sage: y^2
(1, s2*s1)
sage: y.inverse()
(s1, s1*s2*s1)
```

Todo

- Functorial constructor for semidirect products for various categories
- Twofold Direct product as a special case of semidirect product

Element

alias of GroupSemidirectProductElement

act_to_right()

True if the left factor acts on the right factor and False if the right factor acts on the left factor.

EXAMPLES:

```
sage: def twist(x,y):
....:     return y
sage: GroupSemidirectProduct(WeylGroup(['A',2],prefix="s"), WeylGroup(['A',3],prefix="t"),tw
True
```

group_generators()

Return generators of `self`.

EXAMPLES:

```
sage: twist = lambda x,y: y
sage: import __main__
sage: __main__.twist = twist
sage: EZ = GroupExp()(ZZ)
sage: GroupSemidirectProduct(EZ,EZ,twist,print_tuple=True).group_generators()
((1, 0), (0, 1))
```

one()

The identity element of the semidirect product group.

EXAMPLES:

```
sage: G = GL(2,QQ)
sage: V = QQ^2
sage: EV = GroupExp()(V) # make a multiplicative version of V
sage: def twist(g,v):
....:     return EV(g*v.value)
sage: one = GroupSemidirectProduct(G, EV, twist=twist, prefix1 = 't').one(); one
1
sage: one.cartesian_projection(0)
[1 0]
[0 1]
sage: one.cartesian_projection(1)
(0, 0)
```

opposite_semidirect_product()

Create the same semidirect product but with the positions of the groups exchanged.

EXAMPLES:

```
sage: G = GL(2,QQ)
sage: L = QQ^2
sage: EL = GroupExp()(L)
sage: H = GroupSemidirectProduct(G, EL, twist = lambda g,v: EL(g*v.value), prefix1 = 't'); H
Semidirect product of General Linear Group of degree 2 over Rational Field acting on Multipl
sage: h = H((Matrix([[0,1],[1,0]]), EL.an_element())); h
[0 1]
[1 0] * t[(1, 0)]
sage: Hop = H.opposite_semidirect_product(); Hop
Semidirect product of Multiplicative form of Vector space of dimension 2 over Rational Field
sage: hop = h.to_opposite(); hop
t[(0, 1)] * [0 1]
[1 0]
sage: hop in Hop
True
```

product (x, y)

The product of elements x and y in the semidirect product group.

EXAMPLES:

```
sage: G = GL(2,QQ)
sage: V = QQ^2
sage: EV = GroupExp()(V) # make a multiplicative version of V
sage: def twist(g,v):
....:     return EV(g*v.value)
sage: S = GroupSemidirectProduct(G, EV, twist=twist, prefix1 = 't')
sage: g = G([[2,1],[3,1]]); g
[2 1]
[3 1]
sage: v = EV.an_element(); v
(1, 0)
sage: x = S((g,v)); x
[2 1]
[3 1] * t[(1, 0)]
sage: x*x # indirect doctest
[7 3]
[9 4] * t[(0, 3)]
```

class sage.groups.group_semidirect_product.**GroupSemidirectProductElement**

Bases: sage.sets.cartesian_product.CartesianProduct.Element

Element class for GroupSemidirectProduct.

inverse ()

The inverse of self.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).root_lattice()
sage: from sage.groups.group_exp import GroupExp
sage: EL = GroupExp()(L)
sage: W = L.weyl_group(prefix="s")
sage: def twist(w,v):
....:     return EL(w.action(v.value))
sage: G = GroupSemidirectProduct(W, EL, twist, prefix1='t')
sage: g = G.an_element(); g
s1*s2 * t[2*alpha[1] + 2*alpha[2]]
sage: g.inverse()
s2*s1 * t[2*alpha[1]]
```

to_opposite ()

Send an element to its image in the opposite semidirect product.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).root_lattice(); L
Root lattice of the Root system of type ['A', 2]
sage: from sage.groups.group_exp import GroupExp
sage: EL = GroupExp()(L)
sage: W = L.weyl_group(prefix="s"); W
Weyl Group of type ['A', 2] (as a matrix group acting on the root lattice)
sage: def twist(w,v):
....:     return EL(w.action(v.value))
sage: G = GroupSemidirectProduct(W, EL, twist, prefix1='t'); G
Semidirect product of Weyl Group of type ['A', 2] (as a matrix group acting on the root latt
sage: mu = L.an_element(); mu
2*alpha[1] + 2*alpha[2]
```

```
sage: w = W.an_element(); w
s1*s2
sage: g = G((w,EL(mu))); g
s1*s2 * t[2*alpha[1] + 2*alpha[2]]
sage: g.to_opposite()
t[-2*alpha[1]] * s1*s2
sage: g.to_opposite().parent()
Semidirect product of Multiplicative form of Root lattice of the Root system of type ['A', 2
```

CHAPTER

SEVENTEEN

MULTIPLICATIVE ABELIAN GROUPS

This module lets you compute with finitely generated Abelian groups of the form

$$G = \mathbf{Z}^r \oplus \mathbf{Z}_{k_1} \oplus \cdots \oplus \mathbf{Z}_{k_t}$$

It is customary to denote the infinite cyclic group $\mathbf{Z}$ as having order 0, so the data defining the Abelian group can be written as an integer vector

$$\vec{k} = (0, \dots, 0, k_1, \dots, k_t)$$

where there are r zeroes and t non-zero values. To construct this Abelian group in Sage, you can either specify all entries of $\vec{k}$ or only the non-zero entries together with the total number of generators:

```
sage: AbelianGroup([0,0,0,2,3])
Multiplicative Abelian group isomorphic to Z x Z x Z x C2 x C3
sage: AbelianGroup(5, [2,3])
Multiplicative Abelian group isomorphic to Z x Z x Z x C2 x C3
```

It is also legal to specify 1 as the order. The corresponding generator will be the neutral element, but it will still take up an index in the labelling of the generators:

```
sage: G = AbelianGroup([2,1,3], names='g')
sage: G.gens()
(g0, 1, g2)
```

Note that this presentation is not unique, for example $\mathbf{Z}_6 = \mathbf{Z}_2 \times \mathbf{Z}_3$. The orders of the generators $\vec{k} = (0, \dots, 0, k_1, \dots, k_t)$ has previously been called invariants in Sage, even though they are not necessarily the (unique) invariant factors of the group. You should now use `gens_orders()` instead:

```
sage: J = AbelianGroup([2,0,3,2,4]);  J
Multiplicative Abelian group isomorphic to C2 x Z x C3 x C2 x C4
sage: J.gens_orders()            # use this instead
(2, 0, 3, 2, 4)
sage: J.invariants()             # deprecated
(2, 0, 3, 2, 4)
sage: J.elementary_divisors()    # these are the "invariant factors"
(2, 2, 12, 0)
sage: for i in range(J.ngens()):
...       print i, J.gen(i), J.gen(i).order()     # or use this form
0 f0 2
1 f1 +Infinity
2 f2 3
3 f3 2
4 f4 4
```

Background on invariant factors and the Smith normal form (according to section 4.1 of [C1]): An abelian group is a group A for which there exists an exact sequence $\mathbf{Z}^k \rightarrow \mathbf{Z}^\ell \rightarrow A \rightarrow 1$, for some positive integers k, ℓ with $k \leq \ell$. For example, a finite abelian group has a decomposition

$$A = \langle a_1 \rangle \times \cdots \times \langle a_\ell \rangle,$$

where $ord(a_i) = p_i^{c_i}$, for some primes p_i and some positive integers c_i, $i = 1, ..., \ell$. GAP calls the list (ordered by size) of the $p_i^{c_i}$ the *abelian invariants*. In Sage they will be called *invariants*. In this situation, $k = \ell$ and $\phi : \mathbf{Z}^\ell \rightarrow A$ is the map $\phi(x_1, ..., x_\ell) = a_1^{x_1} ... a_\ell^{x_\ell}$, for $(x_1, ..., x_\ell) \in \mathbf{Z}^\ell$. The matrix of relations $M : \mathbf{Z}^k \rightarrow \mathbf{Z}^\ell$ is the matrix whose rows generate the kernel of ϕ as a $\mathbf{Z}$-module. In other words, $M = (M_{ij})$ is a $\ell \times \ell$ diagonal matrix with $M_{ii} = p_i^{c_i}$. Consider now the subgroup $B \subset A$ generated by $b_1 = a_1^{f_{1,1}} ... a_\ell^{f_{\ell,1}}$, ..., $b_m = a_1^{f_{1,m}} ... a_\ell^{f_{\ell,m}}$. The kernel of the map $\phi_B : \mathbf{Z}^m \rightarrow B$ defined by $\phi_B(y_1, ..., y_m) = b_1^{y_1} ... b_m^{y_m}$, for $(y_1, ..., y_m) \in \mathbf{Z}^m$, is the kernel of the matrix

$$F = \begin{pmatrix} f_{11} & f_{12} & \dots & f_{1m} \\ f_{21} & f_{22} & \dots & f_{2m} \\ \vdots & & \ddots & \vdots \\ f_{\ell,1} & f_{\ell,2} & \dots & f_{\ell,m} \end{pmatrix},$$

regarded as a map $\mathbf{Z}^m \rightarrow (\mathbf{Z}/p_1^{c_1}\mathbf{Z}) \times ... \times (\mathbf{Z}/p_\ell^{c_\ell}\mathbf{Z})$. In particular, $B \cong \mathbf{Z}^m/ker(F)$. If $B = A$ then the Smith normal form (SNF) of a generator matrix of $ker(F)$ and the SNF of M are the same. The diagonal entries s_i of the SNF $S = diag[s_1, s_2, s_3, ...s_r, 0, 0, ...0]$, are called *determinantal divisors* of F. where r is the rank. The {it invariant factors} of A are:

$$s_1, s_2/s_1, s_3/s_2, ...s_r/s_{r-1}.$$

Sage supports multiplicative abelian groups on any prescribed finite number $n \geq 0$ of generators. Use the `AbelianGroup()` function to create an abelian group, and the `gen()` and `gens()` methods to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `AbelianGroup()` function.

EXAMPLE 1:

We create an abelian group in zero or more variables; the syntax T(1) creates the identity element even in the rank zero case:

```
sage: T = AbelianGroup(0,[])
sage: T
Trivial Abelian group
sage: T.gens()
()
sage: T(1)
1
```

EXAMPLE 2:

An Abelian group uses a multiplicative representation of elements, but the underlying representation is lists of integer exponents:

```
sage: F = AbelianGroup(5,[3,4,5,5,7],names = list("abcde"))
sage: F
Multiplicative Abelian group isomorphic to C3 x C4 x C5 x C5 x C7
sage: (a,b,c,d,e) = F.gens()
sage: a*b^2*e*d
a*b^2*d*e
sage: x = b^2*e*d*a^7
sage: x
a*b^2*d*e
sage: x.list()
[1, 2, 0, 1, 1]
```

REFERENCES:

- [C1] H. Cohen Advanced topics in computational number theory, Springer, 2000.
- [C2] —-, A course in computational algebraic number theory, Springer, 1996.
- [R] J. Rotman, An introduction to the theory of groups, 4th ed, Springer, 1995.

> **Warning:** Many basic properties for infinite abelian groups are not implemented.

AUTHORS:

- William Stein, David Joyner (2008-12): added (user requested) is_cyclic, fixed elementary_divisors.
- David Joyner (2006-03): (based on free abelian monoids by David Kohel)
- David Joyner (2006-05) several significant bug fixes
- David Joyner (2006-08) trivial changes to docs, added random, fixed bug in how invariants are recorded
- David Joyner (2006-10) added dual_group method
- David Joyner (2008-02) fixed serious bug in word_problem
- David Joyner (2008-03) fixed bug in trivial group case
- David Loeffler (2009-05) added subgroups method
- Volker Braun (2012-11) port to new Parent base. Use tuples for immutables. Rename invariants to gens_orders.

`sage.groups.abelian_gps.abelian_group.`**`AbelianGroup`**`(`*n*, *gens_orders=None*, *names='f'*`)`

Create the multiplicative abelian group in n generators with given orders of generators (which need not be prime powers).

INPUT:

- **n – integer (optional). If not specified, will be derived** from `gens_orders`.
- **`gens_orders` – a list of non-negative integers in the form** $[a_0, a_1, \dots, a_{n-1}]$, typically written in increasing order. This list is padded with zeros if it has length less than n. The orders of the commuting generators, with 0 denoting an infinite cyclic factor.
- `names` – (optional) names of generators

Alternatively, you can also give input in the form `AbelianGroup(gens_orders, names="f")`, where the names keyword argument must be explicitly named.

OUTPUT:

Abelian group with generators and invariant type. The default name for generator `A.i` is `fi`, as in GAP.

EXAMPLES:

```
sage: F = AbelianGroup(5, [5,5,7,8,9], names='abcde')
sage: F(1)
1
sage: (a, b, c, d, e) = F.gens()
sage: mul([ a, b, a, c, b, d, c, d ], F(1))
a^2*b^2*c^2*d^2
sage: d * b**2 * c**3
b^2*c^3*d
sage: F = AbelianGroup(3,[2]*3); F
Multiplicative Abelian group isomorphic to C2 x C2 x C2
sage: H = AbelianGroup([2,3], names="xy"); H
Multiplicative Abelian group isomorphic to C2 x C3
```

```
sage: AbelianGroup(5)
Multiplicative Abelian group isomorphic to Z x Z x Z x Z x Z
sage: AbelianGroup(5).order()
+Infinity
```

Notice that 0's are prepended if necessary:

```
sage: G = AbelianGroup(5, [2,3,4]);  G
Multiplicative Abelian group isomorphic to Z x Z x C2 x C3 x C4
sage: G.gens_orders()
(0, 0, 2, 3, 4)
```

The invariant list must not be longer than the number of generators:

```
sage: AbelianGroup(2, [2,3,4])
Traceback (most recent call last):
...
ValueError: gens_orders (=(2, 3, 4)) must have length n (=2)
```

class `sage.groups.abelian_gps.abelian_group.`**`AbelianGroup_class`**(*generator_orders*, *names*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.groups.group.AbelianGroup`

The parent for Abelian groups with chosen generator orders.

> **Warning:** You should use `AbelianGroup()` to construct Abelian groups and not instantiate this class directly.

INPUT:

- `generator_orders` – list of integers. The orders of the (commuting) generators. Zero denotes an infinite cyclic generator.
- `names` – names of the group generators (optional).

EXAMPLES:

```
sage: Z2xZ3 = AbelianGroup([2,3])
sage: Z6 = AbelianGroup([6])
sage: Z2xZ3 is Z2xZ3, Z6 is Z6
(True, True)
sage: Z2xZ3 is Z6
False
sage: Z2xZ3 == Z6
True

sage: F = AbelianGroup(5,[5,5,7,8,9],names = list("abcde")); F
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
sage: F = AbelianGroup(5,[2, 4, 12, 24, 120],names = list("abcde")); F
Multiplicative Abelian group isomorphic to C2 x C4 x C12 x C24 x C120
sage: F.elementary_divisors()
(2, 4, 12, 24, 120)

sage: F.category()
Category of finite commutative groups
```

TESTS:

```
sage: AbelianGroup([]).gens_orders()
()
sage: AbelianGroup([1]).gens_orders()
(1,)
sage: AbelianGroup([1,1]).gens_orders()
(1, 1)
sage: AbelianGroup(0).gens_orders()
()
```

Element
alias of AbelianGroupElement

dual_group (*names='X'*, *base_ring=None*)
Returns the dual group.

INPUT:

- names – string or list of strings. The generator names for the dual group.
- base_ring – the base ring. If None (default), then a suitable cyclotomic field is picked automatically.

OUTPUT:

The ~sage.groups.abelian_gps.dual_abelian_group.DualAbelianGroup_class

EXAMPLES:

```
sage: G = AbelianGroup([2])
sage: G.dual_group()
Dual of Abelian Group isomorphic to Z/2Z over Cyclotomic Field of order 2 and degree 1
sage: G.dual_group().gens()
(X,)
sage: G.dual_group(names='Z').gens()
(Z,)

sage: G.dual_group(base_ring=QQ)
Dual of Abelian Group isomorphic to Z/2Z over Rational Field
```

TESTS:

```
sage: H = AbelianGroup(1)
sage: H.dual_group()
Traceback (most recent call last):
...
ValueError: the group must be finite
```

elementary_divisors ()
This returns the elementary divisors of the group, using Pari.

Note: Here is another algorithm for computing the elementary divisors $d_1, d_2, d_3, \ldots$, of a finite abelian group (where $d_1|d_2|d_3|\ldots$ are composed of prime powers dividing the invariants of the group in a way described below). Just factor the invariants a_i that define the abelian group. Then the biggest d_i is the product of the maximum prime powers dividing some a_j. In other words, the largest d_i is the product of p^v, where $v = max(ord_p(a_j) \mathrm{for all} j)$. Now divide out all those p^v's into the list of invariants a_i, and get a new list of "smaller invariants"". Repeat the above procedure on these ""smaller invariants"" to compute d_{i-1}, and so on. (Thanks to Robert Miller for communicating this algorithm.)

OUTPUT:

A tuple of integers.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3])
sage: G.elementary_divisors()
(6,)
sage: G = AbelianGroup(1, [6])
sage: G.elementary_divisors()
(6,)
sage: G = AbelianGroup(2,[2,6])
sage: G
Multiplicative Abelian group isomorphic to C2 x C6
sage: G.gens_orders()
(2, 6)
sage: G.elementary_divisors()
(2, 6)
sage: J = AbelianGroup([1,3,5,12])
sage: J.elementary_divisors()
(3, 60)
sage: G = AbelianGroup(2,[0,6])
sage: G.elementary_divisors()
(6, 0)
sage: AbelianGroup([3,4,5]).elementary_divisors()
(60,)
```

exponent()

Return the exponent of this abelian group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,7]); G
Multiplicative Abelian group isomorphic to C2 x C3 x C7
sage: G.exponent()
42
sage: G = AbelianGroup([2,4,6]); G
Multiplicative Abelian group isomorphic to C2 x C4 x C6
sage: G.exponent()
12
```

gen(*i=0*)

The i-th generator of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(5,[],names='a')
sage: F.0
a0
sage: F.2
a2
sage: F.gens_orders()
(0, 0, 0, 0, 0)

sage: G = AbelianGroup([2,1,3])
sage: G.gens()
(f0, 1, f2)
```

gens()

Return the generators of the group.

OUTPUT:

A tuple of group elements. The generators according to the chosen gens_orders().

EXAMPLES:

```
sage: F = AbelianGroup(5,[3,2],names='abcde')
sage: F.gens()
(a, b, c, d, e)
sage: [ g.order() for g in F.gens() ]
[+Infinity, +Infinity, +Infinity, 3, 2]
```

gens_orders()

Return the orders of the cyclic factors that this group has been defined with.

Use elementary_divisors() if you are looking for an invariant of the group.

OUTPUT:

A tuple of integers.

EXAMPLES:

```
sage: Z2xZ3 = AbelianGroup([2,3])
sage: Z2xZ3.gens_orders()
(2, 3)
sage: Z2xZ3.elementary_divisors()
(6,)

sage: Z6 = AbelianGroup([6])
sage: Z6.gens_orders()
(6,)
sage: Z6.elementary_divisors()
(6,)

sage: Z2xZ3.is_isomorphic(Z6)
True
sage: Z2xZ3 is Z6
False
```

TESTS:

```
sage: F = AbelianGroup(3, [2], names='abc')
sage: map(type, F.gens_orders())
[<type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer.Integer'>]
```

identity()

Return the identity element of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,2])
sage: e = G.identity()
sage: e
1
sage: g = G.gen(0)
sage: g*e
f0
sage: e*g
f0
```

invariants()

Return the orders of the cyclic factors that this group has been defined with.

For historical reasons this has been called invariants in Sage, even though they are not necessarily the invariant factors of the group. Use `gens_orders()` instead:

```
sage: J = AbelianGroup([2,0,3,2,4]);  J
Multiplicative Abelian group isomorphic to C2 x Z x C3 x C2 x C4
sage: J.invariants()    # deprecated
(2, 0, 3, 2, 4)
sage: J.gens_orders()   # use this instead
(2, 0, 3, 2, 4)
sage: for i in range(J.ngens()):
...       print i, J.gen(i), J.gen(i).order()     # or use this
0 f0 2
1 f1 +Infinity
2 f2 3
3 f3 2
4 f4 4
```

Use `elementary_divisors()` if you are looking for an invariant of the group.

OUTPUT:

A tuple of integers. Zero means infinite cyclic factor.

EXAMPLES:

```
sage: J = AbelianGroup([2,3])
sage: J.invariants()
(2, 3)
sage: J.elementary_divisors()
(6,)
```

TESTS:

```
sage: F = AbelianGroup(3, [2], names='abc')
sage: map(type, F.gens_orders())
[<type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer.Integer'>]
```

is_commutative()

Return True since this group is commutative.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9, 0])
sage: G.is_commutative()
True
sage: G.is_abelian()
True
```

is_cyclic()

Return True if the group is a cyclic group.

EXAMPLES:

```
sage: J = AbelianGroup([2,3])
sage: J.gens_orders()
(2, 3)
sage: J.elementary_divisors()
(6,)
sage: J.is_cyclic()
```

```
True
sage: G = AbelianGroup([6])
sage: G.gens_orders()
(6,)
sage: G.is_cyclic()
True
sage: H = AbelianGroup([2,2])
sage: H.gens_orders()
(2, 2)
sage: H.is_cyclic()
False
sage: H = AbelianGroup([2,4])
sage: H.elementary_divisors()
(2, 4)
sage: H.is_cyclic()
False
sage: H.permutation_group().is_cyclic()
False
sage: T = AbelianGroup([])
sage: T.is_cyclic()
True
sage: T = AbelianGroup(1,[0]); T
Multiplicative Abelian group isomorphic to Z
sage: T.is_cyclic()
True
sage: B = AbelianGroup([3,4,5])
sage: B.is_cyclic()
True
```

is_isomorphic (*left*, *right*)

Check whether `left` and `right` are isomorphic

INPUT:

- `right` – anything.

OUTPUT:

Boolean. Whether `left` and `right` are isomorphic as abelian groups.

EXAMPLES:

```
sage: G1 = AbelianGroup([2,3,4,5])
sage: G2 = AbelianGroup([2,3,4,5,1])
sage: G1.is_isomorphic(G2)
True
sage: G1 == G2    # syntactic sugar
True
```

is_subgroup (*left*, *right*)

Test whether `left` is a subgroup of `right`.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,4,5])
sage: G.is_subgroup(G)
True

sage: H = G.subgroup([G.1])
sage: H.is_subgroup(G)
True
```

```
sage: G.<a, b> = AbelianGroup(2)
sage: H.<c> = AbelianGroup(1)
sage: H < G
False
```

is_trivial()

Return whether the group is trivial

A group is trivial if it has precisely one element.

EXAMPLES:

```
sage: AbelianGroup([2, 3]).is_trivial()
False
sage: AbelianGroup([1, 1]).is_trivial()
True
```

list()

Return tuple of all elements of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3], names = "ab")
sage: G.list()
(1, b, b^2, a, a*b, a*b^2)

sage: G = AbelianGroup([]); G
Trivial Abelian group
sage: G.list()
(1,)
```

ngens()

The number of free generators of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(10000)
sage: F.ngens()
10000
```

order()

Return the order of this group.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3])
sage: G.order()
6
sage: G = AbelianGroup(3,[2,3,0])
sage: G.order()
+Infinity
```

permutation_group()

Return the permutation group isomorphic to this abelian group.

If the invariants are $q_1, \ldots, q_n$ then the generators of the permutation will be of order $q_1, \ldots, q_n$, respectively.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3]); G
Multiplicative Abelian group isomorphic to C2 x C3
sage: G.permutation_group()
Permutation Group with generators [(3,4,5), (1,2)]
```

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: G.random_element()
f1^2
```

subgroup(*gensH*, *names='f'*)

Create a subgroup of this group. The "big" group must be defined using "named" generators.

INPUT:

- **gensH – list of elements which are products of the** generators of the ambient abelian group G = self

EXAMPLES:

```
sage: G.<a,b,c> = AbelianGroup(3, [2,3,4]); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: H = G.subgroup([a*b,a]); H
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a*b, a}
sage: H < G
True
sage: F = G.subgroup([a,b^2])
sage: F
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a, b^2}
sage: F.gens()
(a, b^2)
sage: F = AbelianGroup(5,[30,64,729],names = list("abcde"))
sage: a,b,c,d,e = F.gens()
sage: F.subgroup([a,b])
Multiplicative Abelian subgroup isomorphic to Z x Z generated by {a, b}
sage: F.subgroup([c,e])
Multiplicative Abelian subgroup isomorphic to C2 x C3 x C5 x C729 generated by {c, e}
```

subgroup_reduced(*elts*, *verbose=False*)

Given a list of lists of integers (corresponding to elements of self), find a set of independent generators for the subgroup generated by these elements, and return the subgroup with these as generators, forgetting the original generators.

This is used by the `subgroups` routine.

An error will be raised if the elements given are not linearly independent over QQ.

EXAMPLE:

```
sage: G = AbelianGroup([4,4])
sage: G.subgroup( [ G([1,0]), G([1,2]) ])
Multiplicative Abelian subgroup isomorphic to C2 x C4
generated by {f0, f0*f1^2}
sage: AbelianGroup([4,4]).subgroup_reduced( [ [1,0], [1,2] ])
Multiplicative Abelian subgroup isomorphic to C2 x C4
generated by {f1^2, f0}
```

subgroups (*check=False*)

Compute all the subgroups of this abelian group (which must be finite).

TODO: This is *many orders of magnitude* slower than Magma.

INPUT:

- check: if True, performs the same computation in GAP and checks that the number of subgroups generated is the same. (I don't know how to convert GAP's output back into Sage, so we don't actually compare the subgroups).

ALGORITHM:

If the group is cyclic, the problem is easy. Otherwise, write it as a direct product A x B, where B is cyclic. Compute the subgroups of A (by recursion).

Now, for every subgroup C of A x B, let G be its *projection onto* A and H its *intersection with* B. Then there is a well-defined homomorphism f: G -> B/H that sends a in G to the class mod H of b, where (a,b) is any element of C lifting a; and every subgroup C arises from a unique triple (G, H, f).

EXAMPLES:

```
sage: AbelianGroup([2,3]).subgroups()
[Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {f0*f1^2},
 Multiplicative Abelian subgroup isomorphic to C2 generated by {f0},
 Multiplicative Abelian subgroup isomorphic to C3 generated by {f1},
 Trivial Abelian subgroup]
sage: len(AbelianGroup([2,4,8]).subgroups())
81
```

TESTS:

```
sage: AbelianGroup([]).subgroups()
[Trivial Abelian group]
```

Check that trac ticket #14196 is fixed:

```
sage: B = AbelianGroup([1,2])
sage: B.subgroups()
[Multiplicative Abelian subgroup isomorphic to C2 generated by {f1},
 Trivial Abelian subgroup]
```

class sage.groups.abelian_gps.abelian_group.**AbelianGroup_subgroup** (*ambient*, *gens*, *names='f'*)

Bases: sage.groups.abelian_gps.abelian_group.AbelianGroup_class

Subgroup subclass of AbelianGroup_class, so instance methods are inherited.

TODO:

- There should be a way to coerce an element of a subgroup into the ambient group.

ambient_group ()

Return the ambient group related to self.

OUTPUT:

A multiplicative Abelian group.

EXAMPLES:

```
sage: G.<a,b,c> = AbelianGroup([2,3,4])
sage: H = G.subgroup([a, b^2])
```

```
sage: H.ambient_group() is G
True
```

equals (*left*, *right*)

Check whether `left` and `right` are the same (sub)group.

INPUT:

- `right` – anything.

OUTPUT:

Boolean. If `right` is a subgroup, test whether `left` and `right` are the same subset of the ambient group. If `right` is not a subgroup, test whether they are isomorphic groups, see `is_isomorphic()`.

EXAMPLES:

```
sage: G = AbelianGroup(3, [2,3,4], names="abc"); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: a,b,c = G.gens()
sage: F = G.subgroup([a,b^2]); F
Multiplicative Abelian subgroup isomorphic to C2 x C3 generated by {a, b^2}
sage: F<G
True

sage: A = AbelianGroup(1, [6])
sage: A.subgroup(list(A.gens())) == A
True

sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: B = G.subgroup([b])
sage: A.equals(B)
False
sage: A == B        # sames as A.equals(B)
False
sage: A.is_isomorphic(B)
True
```

gen (*n*)

Return the nth generator of this subgroup.

EXAMPLE:

```
sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: A.gen(0)
a
```

gens ()

Return the generators for this subgroup.

OUTPUT:

A tuple of group elements generating the subgroup.

EXAMPLES:

```
sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: G.gens()
(a, b)
```

```
sage: A.gens()
(a,)
```

sage.groups.abelian_gps.abelian_group.**is_AbelianGroup**(*x*)

Return True if x is an Abelian group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group import is_AbelianGroup
sage: F = AbelianGroup(5,[5,5,7,8,9],names = list("abcde")); F
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
sage: is_AbelianGroup(F)
True
sage: is_AbelianGroup(AbelianGroup(7, [3]*7))
True
```

sage.groups.abelian_gps.abelian_group.**word_problem**(*words*, *g*, *verbose=False*)

G and H are abelian, g in G, H is a subgroup of G generated by a list (words) of elements of G. If g is in H, return the expression for g as a word in the elements of (words).

The 'word problem' for a finite abelian group G boils down to the following matrix-vector analog of the Chinese remainder theorem.

Problem: Fix integers $1 < n_1 \leq n_2 \leq ... \leq n_k$ (indeed, these n_i will all be prime powers), fix a generating set $g_i = (a_{i1}, ..., a_{ik})$ (with $a_{ij} \in \mathrm{Z}/n_j\mathrm{Z}$), for $1 \leq i \leq \ell$, for the group G, and let $d = (d_1, ..., d_k)$ be an element of the direct product $\mathrm{Z}/n_1\mathrm{Z} \times ... \times \mathrm{Z}/n_k\mathrm{Z}$. Find, if they exist, integers $c_1, ..., c_\ell$ such that $c_1 g_1 + ... + c_\ell g_\ell = d$. In other words, solve the equation $cA = d$ for $c \in \mathrm{Z}^\ell$, where A is the matrix whose rows are the g_i's. Of course, it suffices to restrict the c_i's to the range $0 \leq c_i \leq N - 1$, where N denotes the least common multiple of the integers $n_1, ..., n_k$.

This function does not solve this directly, as perhaps it should. Rather (for both speed and as a model for a similar function valid for more general groups), it pushes it over to GAP, which has optimized (non-deterministic) algorithms for the word problem. Essentially, this function is a wrapper for the GAP function 'Factorization'.

EXAMPLE:

```
sage: G.<a,b,c> = AbelianGroup(3,[2,3,4]); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: w = word_problem([a*b,a*c], b*c); w #random
[[a*b, 1], [a*c, 1]]
sage: prod([x^i for x,i in w]) == b*c
True
sage: w = word_problem([a*c,c],a); w #random
[[a*c, 1], [c, -1]]
sage: prod([x^i for x,i in w]) == a
True
sage: word_problem([a*c,c],a,verbose=True) #random
a = (a*c)^1*(c)^-1
[[a*c, 1], [c, -1]]

sage: A.<a,b,c,d,e> = AbelianGroup(5,[4, 5, 5, 7, 8])
sage: b1 = a^3*b*c*d^2*e^5
sage: b2 = a^2*b*c^2*d^3*e^3
sage: b3 = a^7*b^3*c^5*d^4*e^4
sage: b4 = a^3*b^2*c^2*d^3*e^5
sage: b5 = a^2*b^4*c^2*d^4*e^5
sage: w = word_problem([b1,b2,b3,b4,b5],e); w #random
[[a^3*b*c*d^2*e^5, 1], [a^2*b*c^2*d^3*e^3, 1], [a^3*b^3*d^4*e^4, 3], [a^2*b^4*c^2*d^4*e^5, 1]]
sage: prod([x^i for x,i in w]) == e
True
```

```
sage: word_problem([a,b,c,d,e],e)
[[e, 1]]
sage: word_problem([a,b,c,d,e],b)
[[b, 1]]
```

Warning:

1. Might have unpleasant effect when the word problem cannot be solved.
2. Uses permutation groups, so may be slow when group is large. The instance method word_problem of the class AbelianGroupElement is implemented differently (wrapping GAP's 'EpimorphismFromFreeGroup' and 'PreImagesRepresentative') and may be faster.

CHAPTER

EIGHTEEN

MULTIPLICATIVE ABELIAN GROUPS WITH VALUES

Often, one ends up with a set that forms an Abelian group. It would be nice if one could return an Abelian group class to encapsulate the data. However, `AbelianGroup()` is an abstract Abelian group defined by generators and relations. This module implements `AbelianGroupWithValues` that allows the group elements to be decorated with values.

An example where this module is used is the unit group of a number field, see `sage.rings.number_field.unit_group`. The units form a finitely generated Abelian group. We can think of the elements either as abstract Abelian group elements or as particular numbers in the number field. The `AbelianGroupWithValues()` keeps track of these associated values.

> **Warning:** Really, this requires a group homomorphism from the abstract Abelian group to the set of values. This is only checked if you pass the `check=True` option to `AbelianGroupWithValues()`.

EXAMPLES:

Here is $\mathbf{Z}_6$ with value -1 assigned to the generator:

```
sage: Z6 = AbelianGroupWithValues([-1], [6], names='g')
sage: g = Z6.gen(0)
sage: g.value()
-1
sage: g*g
g^2
sage: (g*g).value()
1
sage: for i in range(7):
...       print i, g^i, (g^i).value()
0 1 1
1 g -1
2 g^2 1
3 g^3 -1
4 g^4 1
5 g^5 -1
6 1 1
```

The elements come with a coercion embedding into the `values_group()`, so you can use the group elements instead of the values:

```
sage: CF3.<zeta> = CyclotomicField(3)
sage: Z3.<g> = AbelianGroupWithValues([zeta], [3])
sage: Z3.values_group()
Cyclotomic Field of order 3 and degree 2
sage: g.value()
zeta
```

```
sage: CF3(g)
zeta
sage: g + zeta
2*zeta
sage: zeta + g
2*zeta
```

sage.groups.abelian_gps.values.**AbelianGroupWithValues**(*values*, *n*, *gens_orders=None*, *names='f'*, *check=False*, *values_group=None*)

Construct an Abelian group with values associated to the generators.

INPUT:

- `values` – a list/tuple/iterable of values that you want to associate to the generators.
- **n – integer (optional). If not specified, will be derived** from `gens_orders`.
- **`gens_orders` – a list of non-negative integers in the form** $[a_0, a_1, \dots, a_{n-1}]$, typically written in increasing order. This list is padded with zeros if it has length less than n. The orders of the commuting generators, with 0 denoting an infinite cyclic factor.
- `names` – (optional) names of generators
- `values_group` – a parent or `None` (default). The common parent of the values. This might be a group, but can also just contain the values. For example, if the values are units in a ring then the `values_group` would be the whole ring. If `None` it will be derived from the values.

EXAMPLES:

```
sage: G = AbelianGroupWithValues([-1], [6])
sage: g = G.gen(0)
sage: for i in range(7):
...       print i, g^i, (g^i).value()
0 1 1
1 f -1
2 f^2 1
3 f^3 -1
4 f^4 1
5 f^5 -1
6 1 1
sage: G.values_group()
Integer Ring
```

The group elements come with a coercion embedding into the `values_group()`, so you can use them like their `value()`

```
sage: G.values_embedding()
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C6
  To:   Integer Ring
sage: g.value()
-1
sage: 0 + g
-1
sage: 1 + 2*g
-1
```

class sage.groups.abelian_gps.values.**AbelianGroupWithValuesElement**(*parent*, *exponents*, *value=None*)

Bases: `sage.groups.abelian_gps.abelian_group_element.AbelianGroupElement`

An element of an Abelian group with values assigned to generators.

INPUT:

- `exponents` – tuple of integers. The exponent vector defining the group element.
- `parent` – the parent.
- `value` – the value assigned to the group element or `None` (default). In the latter case, the value is computed as needed.

EXAMPLES:

```
sage: F = AbelianGroupWithValues([1,-1], [2,4])
sage: a,b = F.gens()
sage: TestSuite(a*b).run()
```

inverse()

Return the inverse element.

EXAMPLE:

```
sage: G.<a,b> = AbelianGroupWithValues([2,-1], [0,4])
sage: a.inverse()
a^-1
sage: a.inverse().value()
1/2
sage: a.__invert__().value()
1/2
sage: (~a).value()
1/2
sage: (a*b).value()
-2
sage: (a*b).inverse().value()
-1/2
```

value()

Return the value of the group element.

OUTPUT:

The value according to the values for generators, see `gens_values()`.

EXAMPLES:

```
sage: G = AbelianGroupWithValues([5], 1)
sage: G.0.value()
5
```

class `sage.groups.abelian_gps.values.`**`AbelianGroupWithValuesEmbedding`**(*domain*, *codomain*)

Bases: `sage.categories.morphism.Morphism`

The morphism embedding the Abelian group with values in its values group.

INPUT:

- `domain` – a `AbelianGroupWithValues_class`
- `codomain` – the values group (need not be in the cateory of groups, e.g. symbolic ring).

EXAMPLES:

```
sage: Z4.<g> = AbelianGroupWithValues([I], [4])
sage: embedding = Z4.values_embedding();  embedding
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C4
  To:   Symbolic Ring
sage: embedding(1)
1
sage: embedding(g)
I
sage: embedding(g^2)
-1
```

class `sage.groups.abelian_gps.values.`**`AbelianGroupWithValues_class`**(*generator_orders*, *names*, *values*, *values_group*)

Bases: `sage.groups.abelian_gps.abelian_group.AbelianGroup_class`

The class of an Abelian group with values associated to the generator.

INPUT:

- `generator_orders` – tuple of integers. The orders of the generators.
- `names` – string or list of strings. The names for the generators.
- `values` – Tuple the same length as the number of generators. The values assigned to the generators.
- `values_group` – the common parent of the values.

EXAMPLES:

```
sage: G.<a,b> = AbelianGroupWithValues([2,-1], [0,4])
sage: TestSuite(G).run()
```

`Element`

alias of `AbelianGroupWithValuesElement`

`gen`(*i=0*)

The i-th generator of the abelian group.

INPUT:

- `i` – integer (default: 0). The index of the generator.

OUTPUT:

A group element.

EXAMPLES:

```
sage: F = AbelianGroupWithValues([1,2,3,4,5], 5,[],names='a')
sage: F.0
a0
sage: F.0.value()
1
sage: F.2
a2
sage: F.2.value()
3

sage: G = AbelianGroupWithValues([-1,0,1], [2,1,3])
sage: G.gens()
(f0, 1, f2)
```

gens_values()

Return the values associated to the generators.

OUTPUT:

A tuple.

EXAMPLES:

```
sage: G = AbelianGroupWithValues([-1,0,1], [2,1,3])
sage: G.gens()
(f0, 1, f2)
sage: G.gens_values()
(-1, 0, 1)
```

values_embedding()

Return the embedding of `self` in `values_group()`.

OUTPUT:

A morphism.

EXAMPLES:

```
sage: Z4 = AbelianGroupWithValues([I], [4])
sage: Z4.values_embedding()
Generic morphism:
  From: Multiplicative Abelian group isomorphic to C4
  To:   Symbolic Ring
```

values_group()

The common parent of the values.

The values need to form a multiplicative group, but can be embedded in a larger structure. For example, if the values are units in a ring then the `values_group()` would be the whole ring.

OUTPUT:

The common parent of the values, containing the group generated by all values.

EXAMPLES:

```
sage: G = AbelianGroupWithValues([-1,0,1], [2,1,3])
sage: G.values_group()
Integer Ring

sage: Z4 = AbelianGroupWithValues([I], [4])
sage: Z4.values_group()
Symbolic Ring
```

CHAPTER

NINETEEN

DUAL GROUPS OF FINITE MULTIPLICATIVE ABELIAN GROUPS

The basic idea is very simple. Let G be an abelian group and G^* its dual (i.e., the group of homomorphisms from G to $\mathbf{C}^\times$). Let g_j, $j = 1, .., n$, denote generators of G - say g_j is of order $m_j > 1$. There are generators X_j, $j = 1, .., n$, of G^* for which $X_j(g_j) = \exp(2\pi i/m_j)$ and $X_i(g_j) = 1$ if $i \neq j$. These are used to construct G^*.

Sage supports multiplicative abelian groups on any prescribed finite number $n > 0$ of generators. Use `AbelianGroup()` function to create an abelian group, the `dual_group()` method to create its dual, and then the `gen()` and `gens()` methods to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `dual_group()` method.

EXAMPLES:

```
sage: F = AbelianGroup(5, [2,5,7,8,9], names='abcde')
sage: (a, b, c, d, e) = F.gens()

sage: Fd = F.dual_group(names='ABCDE')
sage: Fd.base_ring()
Cyclotomic Field of order 2520 and degree 576
sage: A,B,C,D,E = Fd.gens()
sage: A(a)
-1
sage: A(b), A(c), A(d), A(e)
(1, 1, 1, 1)

sage: Fd = F.dual_group(names='ABCDE', base_ring=CC)
sage: A,B,C,D,E = Fd.gens()
sage: A(a)    # abs tol 1e-8
-1.00000000000000 + 0.00000000000000*I
sage: A(b); A(c); A(d); A(e)
1.00000000000000
1.00000000000000
1.00000000000000
1.00000000000000
```

AUTHORS:

- David Joyner (2006-08) (based on abelian_groups)
- David Joyner (2006-10) modifications suggested by William Stein
- Volker Braun (2012-11) port to new Parent base. Use tuples for immutables. Default to cyclotomic base ring.

class `sage.groups.abelian_gps.dual_abelian_group.`**`DualAbelianGroup_class`**(*G*, *names*, *base_ring*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.groups.group.AbelianGroup`

Dual of abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(5,[3,5,7,8,9], names="abcde")
sage: F.dual_group()
Dual of Abelian Group isomorphic to Z/3Z x Z/5Z x Z/7Z x Z/8Z x Z/9Z
over Cyclotomic Field of order 2520 and degree 576
sage: F = AbelianGroup(4,[15,7,8,9], names="abcd")
sage: F.dual_group(base_ring=CC)
Dual of Abelian Group isomorphic to Z/15Z x Z/7Z x Z/8Z x Z/9Z
over Complex Field with 53 bits of precision
```

Element

alias of `DualAbelianGroupElement`

base_ring()

Return the scalars over which the group is dualized.

EXAMPLES:

```
sage: F = AbelianGroup(3,[5,64,729], names=list("abc"))
sage: Fd = F.dual_group(base_ring=CC)
sage: Fd.base_ring()
Complex Field with 53 bits of precision
```

gen(*i=0*)

The i-th generator of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(3,[1,2,3],names='a')
sage: Fd = F.dual_group(names="A")
sage: Fd.0
1
sage: Fd.1
A1
sage: Fd.gens_orders()
(1, 2, 3)
```

gens()

Return the generators for the group.

OUTPUT:

A tuple of group elements generating the group.

EXAMPLES:

```
sage: F = AbelianGroup([7,11]).dual_group()
sage: F.gens()
(X0, X1)
```

gens_orders()

The orders of the generators of the dual group.

OUTPUT:

A tuple of integers.

EXAMPLES:

```
sage: F = AbelianGroup([5]*1000)
sage: Fd = F.dual_group()
```

```
sage: invs = Fd.gens_orders(); len(invs)
1000
```

group()

Return the group that `self` is the dual of.

EXAMPLES:

```
sage: F = AbelianGroup(3,[5,64,729], names=list("abc"))
sage: Fd = F.dual_group(base_ring=CC)
sage: Fd.group() is F
True
```

invariants()

The invariants of the dual group.

You should use `gens_orders()` instead.

EXAMPLES:

```
sage: F = AbelianGroup([5]*1000)
sage: Fd = F.dual_group()
sage: invs = Fd.gens_orders(); len(invs)
1000
```

is_commutative()

Return True since this group is commutative.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = G.dual_group()
sage: Gd.is_commutative()
True
sage: Gd.is_abelian()
True
```

list()

Return tuple of all elements of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3], names="ab")
sage: Gd = G.dual_group(names="AB")
sage: Gd.list()
(1, B, B^2, A, A*B, A*B^2)
```

ngens()

The number of generators of the dual group.

EXAMPLES:

```
sage: F = AbelianGroup([7]*100)
sage: Fd = F.dual_group()
sage: Fd.ngens()
100
```

order()

Return the order of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = G.dual_group()
sage: Gd.order()
54
```

random_element()

Return a random element of this dual group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = G.dual_group(base_ring=CC)
sage: Gd.random_element()
X1^2

sage: N = 43^2-1
sage: G = AbelianGroup([N],names="a")
sage: Gd = G.dual_group(names="A", base_ring=CC)
sage: a, = G.gens()
sage: A, = Gd.gens()
sage: x = a^(N/4); y = a^(N/3); z = a^(N/14)
sage: X = A*Gd.random_element(); X
A^615
sage: len([a for a in [x,y,z] if abs(X(a)-1)>10^(-8)])
2
```

sage.groups.abelian_gps.dual_abelian_group.**is_DualAbelianGroup**(x)

Return True if x is the dual group of an abelian group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.dual_abelian_group import is_DualAbelianGroup
sage: F = AbelianGroup(5,[3,5,7,8,9], names=list("abcde"))
sage: Fd = F.dual_group()
sage: is_DualAbelianGroup(Fd)
True
sage: F = AbelianGroup(3,[1,2,3], names='a')
sage: Fd = F.dual_group()
sage: Fd.gens()
(1, X1, X2)
sage: F.gens()
(1, a1, a2)
```

CHAPTER

TWENTY

BASE CLASS FOR ABELIAN GROUP ELEMENTS

This is the base class for both `abelian_group_element` and `dual_abelian_group_element`.

As always, elements are immutable once constructed.

class `sage.groups.abelian_gps.element_base.`**`AbelianGroupElementBase`**(*parent*, *exponents*)

Bases: `sage.structure.element.MultiplicativeGroupElement`

Base class for abelian group elements

The group element is defined by a tuple whose `i`-th entry is an integer in the range from 0 (inclusively) to `G.gen(i).order()` (exclusively) if the i-th generator is of finite order, and an arbitrary integer if the i-th generator is of infinte order.

INPUT:

- `exponents` – `1` or a list/tuple/iterable of integers. The exponent vector (with respect to the parent generators) defining the group element.
- `parent` – Abelian group. The parent of the group element.

EXAMPLES:

```
sage: F = AbelianGroup(3,[7,8,9])
sage: Fd = F.dual_group(names="ABC")
sage: A,B,C = Fd.gens()
sage: A*B^-1 in Fd
True
```

`exponents`()

The exponents of the generators defining the group element.

OUTPUT:

A tuple of integers for an abelian group element. The integer can be arbitrary if the corresponding generator has infinite order. If the generator is of finite order, the integer is in the range from 0 (inclusive) to the order (exclusive).

EXAMPLES:

```
sage: F.<a,b,c,f> = AbelianGroup([7,8,9,0])
sage: (a^3*b^2*c).exponents()
(3, 2, 1, 0)
sage: F([3, 2, 1, 0])
a^3*b^2*c
sage: (c^42).exponents()
(0, 0, 6, 0)
sage: (f^42).exponents()
(0, 0, 0, 42)
```

inverse()

Returns the inverse element.

EXAMPLE:

```
sage: G.<a,b> = AbelianGroup([0,5])
sage: a.inverse()
a^-1
sage: a.__invert__()
a^-1
sage: a^-1
a^-1
sage: ~a
a^-1
sage: (a*b).exponents()
(1, 1)
sage: (a*b).inverse().exponents()
(-1, 4)
```

is_trivial()

Test whether `self` is the trivial group element `1`.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: G.<a,b> = AbelianGroup([0,5])
sage: (a^5).is_trivial()
False
sage: (b^5).is_trivial()
True
```

list()

Return a copy of the exponent vector.

Use `exponents()` instead.

OUTPUT:

The underlying coordinates used to represent this element. If this is a word in an abelian group on n generators, then this is a list of nonnegative integers of length n.

EXAMPLES:

```
sage: F = AbelianGroup(5,[2, 3, 5, 7, 8], names="abcde")
sage: a,b,c,d,e = F.gens()
sage: Ad = F.dual_group(names="ABCDE")
sage: A,B,C,D,E = Ad.gens()
sage: (A*B*C^2*D^20*E^65).exponents()
(1, 1, 2, 6, 1)
sage: X = A*B*C^2*D^2*E^-6
sage: X.exponents()
(1, 1, 2, 2, 2)
```

multiplicative_order()

Return the order of this element.

OUTPUT:

An integer or `infinity`.

EXAMPLES:

```
sage: F = AbelianGroup(3,[7,8,9])
sage: Fd = F.dual_group()
sage: A,B,C = Fd.gens()
sage: (B*C).order()
72

sage: F = AbelianGroup(3,[7,8,9]); F
Multiplicative Abelian group isomorphic to C7 x C8 x C9
sage: F.gens()[2].order()
9
sage: a,b,c = F.gens()
sage: (b*c).order()
72
sage: G = AbelianGroup(3,[7,8,9])
sage: type((G.0 * G.1).order())==Integer
True
```

order()

Return the order of this element.

OUTPUT:

An integer or `infinity`.

EXAMPLES:

```
sage: F = AbelianGroup(3,[7,8,9])
sage: Fd = F.dual_group()
sage: A,B,C = Fd.gens()
sage: (B*C).order()
72

sage: F = AbelianGroup(3,[7,8,9]); F
Multiplicative Abelian group isomorphic to C7 x C8 x C9
sage: F.gens()[2].order()
9
sage: a,b,c = F.gens()
sage: (b*c).order()
72
sage: G = AbelianGroup(3,[7,8,9])
sage: type((G.0 * G.1).order())==Integer
True
```

CHAPTER TWENTYONE

ABELIAN GROUP ELEMENTS

AUTHORS:

- David Joyner (2006-02); based on free_abelian_monoid_element.py, written by David Kohel.
- David Joyner (2006-05); bug fix in order
- David Joyner (2006-08); bug fix+new method in pow for negatives+fixed corresponding examples.
- David Joyner (2009-02): Fixed bug in order.
- Volker Braun (2012-11) port to new Parent base. Use tuples for immutables.

EXAMPLES:

Recall an example from abelian groups:

```
sage: F = AbelianGroup(5,[4,5,5,7,8],names = list("abcde"))
sage: (a,b,c,d,e) = F.gens()
sage: x = a*b^2*e*d^20*e^12
sage: x
a*b^2*d^6*e^5
sage: x = a^10*b^12*c^13*d^20*e^12
sage: x
a^2*b^2*c^3*d^6*e^4
sage: y = a^13*b^19*c^23*d^27*e^72
sage: y
a*b^4*c^3*d^6
sage: x*y
a^3*b*c*d^5*e^4
sage: x.list()
[2, 2, 3, 6, 4]
```

class `sage.groups.abelian_gps.abelian_group_element.`**`AbelianGroupElement`**(*parent*, *exponents*)

Bases: `sage.groups.abelian_gps.element_base.AbelianGroupElementBase`

Elements of an `AbelianGroup`

INPUT:

- `x` – list/tuple/iterable of integers (the element vector)
- `parent` – the parent `AbelianGroup`

EXAMPLES:

```
sage: F = AbelianGroup(5, [3,4,5,8,7], 'abcde')
sage: a, b, c, d, e = F.gens()
sage: a^2 * b^3 * a^2 * b^-4
```

```
a*b^3
sage: b^-11
b
sage: a^-11
a
sage: a*b in F
True
```

as_permutation()

Return the element of the permutation group G (isomorphic to the abelian group A) associated to a in A.

EXAMPLES:

```
sage: G = AbelianGroup(3,[2,3,4],names="abc"); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: a,b,c=G.gens()
sage: Gp = G.permutation_group(); Gp
Permutation Group with generators [(6,7,8,9), (3,4,5), (1,2)]
sage: a.as_permutation()
(1,2)
sage: ap = a.as_permutation(); ap
(1,2)
sage: ap in Gp
True
```

word_problem(*words*)

TODO - this needs a rewrite - see stuff in the matrix_grp directory.

G and H are abelian groups, g in G, H is a subgroup of G generated by a list (words) of elements of G. If self is in H, return the expression for self as a word in the elements of (words).

This function does not solve the word problem in Sage. Rather it pushes it over to GAP, which has optimized (non-deterministic) algorithms for the word problem.

> **Warning:** Don't use E (or other GAP-reserved letters) as a generator name.

EXAMPLE:

```
sage: G = AbelianGroup(2,[2,3], names="xy")
sage: x,y = G.gens()
sage: x.word_problem([x,y])
[[x, 1]]
sage: y.word_problem([x,y])
[[y, 1]]
sage: v = (y*x).word_problem([x,y]); v #random
[[x, 1], [y, 1]]
sage: prod([x^i for x,i in v]) == y*x
True
```

sage.groups.abelian_gps.abelian_group_element.**is_AbelianGroupElement**(*x*)

Return true if x is an abelian group element, i.e., an element of type AbelianGroupElement.

EXAMPLES: Though the integer 3 is in the integers, and the integers have an abelian group structure, 3 is not an AbelianGroupElement:

```
sage: from sage.groups.abelian_gps.abelian_group_element import is_AbelianGroupElement
sage: is_AbelianGroupElement(3)
False
sage: F = AbelianGroup(5, [3,4,5,8,7], 'abcde')
```

```
sage: is_AbelianGroupElement(F.0)
True
```

CHAPTER TWENTYTWO

ELEMENTS (CHARACTERS) OF THE DUAL GROUP OF A FINITE ABELIAN GROUP.

To obtain the dual group of a finite Abelian group, use the `dual_group()` method:

```
sage: F = AbelianGroup([2,3,5,7,8], names="abcde")
sage: F
Multiplicative Abelian group isomorphic to C2 x C3 x C5 x C7 x C8

sage: Fd = F.dual_group(names="ABCDE")
sage: Fd
Dual of Abelian Group isomorphic to Z/2Z x Z/3Z x Z/5Z x Z/7Z x Z/8Z
over Cyclotomic Field of order 840 and degree 192
```

The elements of the dual group can be evaluated on elements of the orignial group:

```
sage: a,b,c,d,e = F.gens()
sage: A,B,C,D,E = Fd.gens()
sage: A*B^2*D^7
A*B^2
sage: A(a)
-1
sage: B(b)
zeta840^140 - 1
sage: CC(_)      # abs tol 1e-8
-0.499999999999995 + 0.866025403784447*I
sage: A(a*b)
-1
sage: (A*B*C^2*D^20*E^65).exponents()
(1, 1, 2, 6, 1)
sage: B^(-1)
B^2
```

AUTHORS:

- David Joyner (2006-07); based on abelian_group_element.py.
- David Joyner (2006-10); modifications suggested by William Stein.
- Volker Braun (2012-11) port to new Parent base. Use tuples for immutables. Default to cyclotomic base ring.

class `sage.groups.abelian_gps.dual_abelian_group_element.`**`DualAbelianGroupElement`**(*parent*, *exponents*)

Bases: `sage.groups.abelian_gps.element_base.AbelianGroupElementBase`

Base class for abelian group elements

word_problem (*words*, *display=True*)

This is a rather hackish method and is included for completeness.

The word problem for an instance of DualAbelianGroup as it can for an AbelianGroup. The reason why is that word problem for an instance of AbelianGroup simply calls GAP (which has abelian groups implemented) and invokes "EpimorphismFromFreeGroup" and "PreImagesRepresentative". GAP does not have duals of abelian groups implemented. So, by using the same name for the generators, the method below converts the problem for the dual group to the corresponding problem on the group itself and uses GAP to solve that.

EXAMPLES:

```
sage: G = AbelianGroup(5,[3, 5, 5, 7, 8],names="abcde")
sage: Gd = G.dual_group(names="abcde")
sage: a,b,c,d,e = Gd.gens()
sage: u = a^3*b*c*d^2*e^5
sage: v = a^2*b*c^2*d^3*e^3
sage: w = a^7*b^3*c^5*d^4*e^4
sage: x = a^3*b^2*c^2*d^3*e^5
sage: y = a^2*b^4*c^2*d^4*e^5
sage: e.word_problem([u,v,w,x,y],display=False)
[[b^2*c^2*d^3*e^5, 245]]
```

The command e.word_problem([u,v,w,x,y],display=True) returns the same list but also prints $e = (b^2 * c^2 * d^3 * e^5)^2 45$.

sage.groups.abelian_gps.dual_abelian_group_element.**add_strings** (*x*, *z=0*)

This was in sage.misc.misc but commented out. Needed to add lists of strings in the word_problem method below.

Return the sum of the elements of x. If x is empty, return z.

INPUT:

- `x` – iterable
- `z` – the `0` that will be returned if `x` is empty.

OUTPUT:

The sum of the elements of `x`.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.dual_abelian_group_element import add_strings
sage: add_strings([], z='empty')
'empty'
sage: add_strings(['a', 'b', 'c'])
'abc'
```

sage.groups.abelian_gps.dual_abelian_group_element.**is_DualAbelianGroupElement** (*x*)

Test whether `x` is a dual Abelian group element.

INPUT:

- `x` – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.dual_abelian_group import is_DualAbelianGroupElement
sage: F = AbelianGroup(5,[5,5,7,8,9],names = list("abcde")).dual_group()
sage: is_DualAbelianGroupElement(F)
False
sage: is_DualAbelianGroupElement(F.an_element())
True
```

CHAPTER

TWENTYTHREE

HOMOMORPHISMS OF ABELIAN GROUPS

TODO:

- there must be a homspace first
- there should be hom and Hom methods in abelian group

AUTHORS:

- David Joyner (2006-03-03): initial version

class `sage.groups.abelian_gps.abelian_group_morphism.`**`AbelianGroupMap`**(*parent*)

Bases: `sage.categories.morphism.Morphism`

A set-theoretic map between AbelianGroups.

class `sage.groups.abelian_gps.abelian_group_morphism.`**`AbelianGroupMorphism`**(*G*, *H*, *genss*, *imgss*)

Bases: `sage.categories.morphism.Morphism`

Some python code for wrapping GAP's GroupHomomorphismByImages function for abelian groups. Returns "fail" if gens does not generate self or if the map does not extend to a group homomorphism, self - other.

EXAMPLES:

```
sage: G = AbelianGroup(3,[2,3,4],names="abc"); G
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: a,b,c = G.gens()
sage: H = AbelianGroup(2,[2,3],names="xy"); H
Multiplicative Abelian group isomorphic to C2 x C3
sage: x,y = H.gens()

sage: from sage.groups.abelian_gps.abelian_group_morphism import AbelianGroupMorphism
sage: phi = AbelianGroupMorphism(H,G,[x,y],[a,b])
```

AUTHORS:

- David Joyner (2006-02)

`image`(*J*)

Only works for finite groups.

J must be a subgroup of G. Computes the subgroup of H which is the image of J.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3],names="xy")
sage: x,y = G.gens()
sage: H = AbelianGroup(3,[2,3,4],names="abc")
```

```
sage: a,b,c = H.gens()
sage: phi = AbelianGroupMorphism(G,H,[x,y],[a,b])
```

kernel()

Only works for finite groups.

TODO: not done yet; returns a gap object but should return a Sage group.

EXAMPLES:

```
sage: H = AbelianGroup(3,[2,3,4],names="abc"); H
Multiplicative Abelian group isomorphic to C2 x C3 x C4
sage: a,b,c = H.gens()
sage: G = AbelianGroup(2,[2,3],names="xy"); G
Multiplicative Abelian group isomorphic to C2 x C3
sage: x,y = G.gens()
sage: phi = AbelianGroupMorphism(G,H,[x,y],[a,b])
sage: phi.kernel()
'Group([  ])'
```

class sage.groups.abelian_gps.abelian_group_morphism.**AbelianGroupMorphism_id**(*X*)

Bases: sage.groups.abelian_gps.abelian_group_morphism.AbelianGroupMap

Return the identity homomorphism from X to itself.

EXAMPLES:

sage.groups.abelian_gps.abelian_group_morphism.**is_AbelianGroupMorphism**(*f*)

CHAPTER

TWENTYFOUR

ADDITIVE ABELIAN GROUPS

Additive abelian groups are just modules over $\mathbf{Z}$. Hence the classes in this module derive from those in the module `sage.modules.fg_pid`. The only major differences are in the way elements are printed.

`sage.groups.additive_abelian.additive_abelian_group.`**`AdditiveAbelianGroup`**(*invs*, *remember_generators=True*)

Construct a finitely-generated additive abelian group.

INPUT:

- `invs` (list of integers): the invariants. These should all be greater than or equal to zero.
- `remember_generators` (boolean): whether or not to fix a set of generators (corresponding to the given invariants, which need not be in Smith form).

OUTPUT:

The abelian group $\bigoplus_i \mathbf{Z}/n_i\mathbf{Z}$, where n_i are the invariants.

EXAMPLE:

```
sage: AdditiveAbelianGroup([0, 2, 4])
Additive abelian group isomorphic to Z + Z/2 + Z/4
```

An example of the `remember_generators` switch:

```
sage: G = AdditiveAbelianGroup([0, 2, 3]); G
Additive abelian group isomorphic to Z + Z/2 + Z/3
sage: G.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))

sage: H = AdditiveAbelianGroup([0, 2, 3], remember_generators = False); H
Additive abelian group isomorphic to Z/6 + Z
sage: H.gens()
((0, 1, 2), (1, 0, 0))
```

There are several ways to create elements of an additive abelian group. Realize that there are two sets of generators: the "obvious" ones composed of zeros and ones, one for each invariant given to construct the group, the other being a set of minimal generators. Which set is the default varies with the use of the `remember_generators` switch.

First with "obvious" generators. Note that a raw list will use the minimal generators and a vector (a module element) will use the generators that pair up naturally with the invariants. We create the same element repeatedly.

```
sage: H=AdditiveAbelianGroup([3,2,0], remember_generators=True)
sage: H.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

```
sage: [H.0, H.1, H.2]
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
sage: p=H.0+H.1+6*H.2; p
(1, 1, 6)

sage: H.smith_form_gens()
((2, 1, 0), (0, 0, 1))
sage: q=H.linear_combination_of_smith_form_gens([5,6]); q
(1, 1, 6)
sage: p==q
True

sage: r=H(vector([1,1,6])); r
(1, 1, 6)
sage: p==r
True

sage: s=H(p)
sage: p==s
True
```

Again, but now where the generators are the minimal set. Coercing a list or a vector works as before, but the default generators are different.

```
sage: G=AdditiveAbelianGroup([3,2,0], remember_generators=False)
sage: G.gens()
((2, 1, 0), (0, 0, 1))
sage: [G.0, G.1]
[(2, 1, 0), (0, 0, 1)]
sage: p=5*G.0+6*G.1; p
(1, 1, 6)

sage: H.smith_form_gens()
((2, 1, 0), (0, 0, 1))
sage: q=G.linear_combination_of_smith_form_gens([5,6]); q
(1, 1, 6)
sage: p==q
True

sage: r=G(vector([1,1,6])); r
(1, 1, 6)
sage: p==r
True

sage: s=H(p)
sage: p==s
True
```

class `sage.groups.additive_abelian.additive_abelian_group.`**`AdditiveAbelianGroupElement`**(*parent*, *x*, *check=Tr*

Bases: `sage.modules.fg_pid.fgp_element.FGP_Element`

An element of an `AdditiveAbelianGroup_class`.

class `sage.groups.additive_abelian.additive_abelian_group.`**`AdditiveAbelianGroup_class`**(*cover*, *relations*)

Bases: `sage.modules.fg_pid.fgp_module.FGP_Module_class`, `sage.groups.old.AbelianGroup`

An additive abelian group, implemented using the **Z**-module machinery.

INPUT:

- `cover` – the covering group as **Z**-module.
- `relations` – the relations as submodule of `cover`.

Element

alias of `AdditiveAbelianGroupElement`

exponent()

Return the exponent of this group (the smallest positive integer N such that $Nx = 0$ for all x in the group). If there is no such integer, return 0.

EXAMPLES:

```
sage: AdditiveAbelianGroup([2,4]).exponent()
4
sage: AdditiveAbelianGroup([0, 2,4]).exponent()
0
sage: AdditiveAbelianGroup([]).exponent()
1
```

is_cyclic()

Returns `True` if the group is cyclic.

EXAMPLES:

With no common factors between the orders of the generators, the group will be cyclic.

```
sage: G=AdditiveAbelianGroup([6, 7, 55])
sage: G.is_cyclic()
True
```

Repeating primes in the orders will create a non-cyclic group.

```
sage: G=AdditiveAbelianGroup([6, 15, 21, 33])
sage: G.is_cyclic()
False
```

A trivial group is trivially cyclic.

```
sage: T=AdditiveAbelianGroup([1])
sage: T.is_cyclic()
True
```

is_multiplicative()

Return False since this is an additive group.

EXAMPLE:

```
sage: AdditiveAbelianGroup([0]).is_multiplicative()
False
```

order()

Return the order of this group (an integer or infinity)

EXAMPLES:

```
sage: AdditiveAbelianGroup([2,4]).order()
8
```

```
sage: AdditiveAbelianGroup([0, 2,4]).order()
+Infinity
sage: AdditiveAbelianGroup([]).order()
1
```

short_name()

Return a name for the isomorphism class of this group.

EXAMPLE:

```
sage: AdditiveAbelianGroup([0, 2,4]).short_name()
'Z + Z/2 + Z/4'
sage: AdditiveAbelianGroup([0, 2, 3]).short_name()
'Z + Z/2 + Z/3'
```

class sage.groups.additive_abelian.additive_abelian_group.**AdditiveAbelianGroup_fixed_gens**(*cove rels, gens*)

Bases: sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_class

A variant which fixes a set of generators, which need not be in Smith form (or indeed independent).

gens()

Return the specified generators for self (as a tuple). Compare `self.smithform_gens()`.

EXAMPLE:

```
sage: G = AdditiveAbelianGroup([2,3])
sage: G.gens()
((1, 0), (0, 1))
sage: G.smith_form_gens()
((1, 2),)
```

identity()

Return the identity (zero) element of this group.

EXAMPLE:

```
sage: G = AdditiveAbelianGroup([2, 3])
sage: G.identity()
(0, 0)
```

permutation_group()

Return the permutation group attached to this group.

EXAMPLE:

```
sage: G = AdditiveAbelianGroup([2, 3])
sage: G.permutation_group()
Permutation Group with generators [(3,4,5), (1,2)]
```

sage.groups.additive_abelian.additive_abelian_group.**cover_and_relations_from_invariants**(*invs*

A utility function to construct modules required to initialize the super class.

Given a list of integers, this routine constructs the obvious pair of free modules such that the quotient of the two free modules over **Z** is naturally isomorphic to the corresponding product of cyclic modules (and hence isomorphic to a direct sum of cyclic groups).

EXAMPLES:

```
sage: from sage.groups.additive_abelian.additive_abelian_group import cover_and_relations_from_i
sage: cr([0,2,3])
```

```
(Ambient free module of rank 3 over the principal ideal domain Integer Ring, Free module of degr
Echelon basis matrix:
[0 2 0]
[0 0 3])
```

CHAPTER

TWENTYFIVE

WRAPPER CLASS FOR ABELIAN GROUPS

This class is intended as a template for anything in Sage that needs the functionality of abelian groups. One can create an AdditiveAbelianGroupWrapper object from any given set of elements in some given parent, as long as an `_add_` method has been defined.

EXAMPLES:

We create a toy example based on the Mordell-Weil group of an elliptic curve over **Q**:

```
sage: E = EllipticCurve('30a2')
sage: pts = [E(4,-7,1), E(7/4, -11/8, 1), E(3, -2, 1)]
sage: M = AdditiveAbelianGroupWrapper(pts[0].parent(), pts, [3, 2, 2])
sage: M
Additive abelian group isomorphic to Z/3 + Z/2 + Z/2 embedded in Abelian
group of points on Elliptic Curve defined by y^2 + x*y + y = x^3 - 19*x + 26
over Rational Field
sage: M.gens()
((4 : -7 : 1), (7/4 : -11/8 : 1), (3 : -2 : 1))
sage: 3*M.0
(0 : 1 : 0)
sage: 3000000000000001 * M.0
(4 : -7 : 1)
sage: M == loads(dumps(M))   # known bug, see http://trac.sagemath.org/sage_trac/ticket/11599#comment
True
```

We check that ridiculous operations are being avoided:

```
sage: set_verbose(2, 'additive_abelian_wrapper.py')
sage: 300001 * M.0
verbose 1 (...: additive_abelian_wrapper.py, _discrete_exp) Calling discrete exp on (1, 0, 0)
(4 : -7 : 1)
sage: set_verbose(0, 'additive_abelian_wrapper.py')
```

TODO:

- Implement proper black-box discrete logarithm (using baby-step giant-step). The discrete_exp function can also potentially be speeded up substantially via caching.
- Think about subgroups and quotients, which probably won't work in the current implementation – some fiddly adjustments will be needed in order to be able to pass extra arguments to the subquotient's init method.

class `sage.groups.additive_abelian.additive_abelian_wrapper.`**`AdditiveAbelianGroupWrapper`**(*univers gens, invariants*)

Bases: `sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_fixed_ge`

The parent of AdditiveAbelianGroupWrapperElement

Element
alias of AdditiveAbelianGroupWrapperElement

generator_orders()
The orders of the generators with which this group was initialised. (Note that these are not necessarily a minimal set of generators.) Generators of infinite order are returned as 0. Compare `self.invariants()`, which returns the orders of a minimal set of generators.

EXAMPLE:

```
sage: V = Zmod(6)**2
sage: G = AdditiveAbelianGroupWrapper(V, [2*V.0, 3*V.1], [3, 2])
sage: G.generator_orders()
(3, 2)
sage: G.invariants()
(6,)
```

universe()
The ambient group in which this abelian group lives.

EXAMPLE:

```
sage: G = AdditiveAbelianGroupWrapper(QQbar, [sqrt(QQbar(2)), sqrt(QQbar(3))], [0, 0])
sage: G.universe()
Algebraic Field
```

class `sage.groups.additive_abelian.additive_abelian_wrapper.`**AdditiveAbelianGroupWrapperElemen**

Bases: sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroupElement

An element of an AdditiveAbelianGroupWrapper.

element()
Return the underlying object that this element wraps.

EXAMPLE:

```
sage: T = EllipticCurve('65a').torsion_subgroup().gen(0)
sage: T; type(T)
(0 : 0 : 1)
<class 'sage.groups.additive_abelian.additive_abelian_wrapper.EllipticCurveTorsionSubgroup_w
sage: T.element(); type(T.element())
(0 : 0 : 1)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field'>
```

class `sage.groups.additive_abelian.additive_abelian_wrapper.`**UnwrappingMorphism**(*domain*)
Bases: `sage.categories.morphism.Morphism`

The embedding into the ambient group. Used by the coercion framework.

CHAPTER

TWENTYSIX

CATALOG OF PERMUTATION GROUPS

Type `groups.permutation.<tab>` to access examples of groups implemented as permutation groups.

CHAPTER

TWENTYSEVEN

PERMUTATION GROUPS

A permutation group is a finite group G whose elements are permutations of a given finite set X (i.e., bijections $X \longrightarrow X$) and whose group operation is the composition of permutations. The number of elements of X is called the degree of G.

In Sage, a permutation is represented as either a string that defines a permutation using disjoint cycle notation, or a list of tuples, which represent disjoint cycles. That is:

```
(a,...,b)(c,...,d)...(e,...,f)  <--> [(a,...,b), (c,...,d),..., (e,...,f)]
                  () = identity <--> []
```

You can make the "named" permutation groups (see `permgp_named.py`) and use the following constructions:

- permutation group generated by elements,
- `direct_product_permgroups`, which takes a list of permutation groups and returns their direct product.

JOKE: Q: What's hot, chunky, and acts on a polygon? A: Dihedral soup. Renteln, P. and Dundes, A. "Foolproof: A Sampling of Mathematical Folk Humor." Notices Amer. Math. Soc. 52, 24-34, 2005.

AUTHORS:

- David Joyner (2005-10-14): first version
- David Joyner (2005-11-17)
- William Stein (2005-11-26): rewrite to better wrap Gap
- David Joyner (2005-12-21)
- William Stein and David Joyner (2006-01-04): added conjugacy_class_representatives
- David Joyner (2006-03): reorganization into subdirectory perm_gps; added __contains__, has_element; fixed _cmp_; added subgroup class+methods, PGL,PSL,PSp, PSU classes,
- David Joyner (2006-06): added PGU, functionality to SymmetricGroup, AlternatingGroup, direct_product_permgroups
- David Joyner (2006-08): added degree, ramification_module_decomposition_modular_curve and ramification_module_decomposition_hurwitz_curve methods to PSL(2,q), MathieuGroup, is_isomorphic
- Bobby Moretti (2006)-10): Added KleinFourGroup, fixed bug in DihedralGroup
- David Joyner (2006-10): added is_subgroup (fixing a bug found by Kiran Kedlaya), is_solvable, normalizer, is_normal_subgroup, Suzuki
- David Kohel (2007-02): fixed __contains__ to not enumerate group elements, following the convention for __call__
- David Harvey, Mike Hansen, Nick Alexander, William Stein (2007-02,03,04,05): Various patches

- Nathan Dunfield (2007-05): added orbits
- David Joyner (2007-06): added subgroup method (suggested by David Kohel), composition_series, lower_central_series, upper_central_series, cayley_table, quotient_group, sylow_subgroup, is_cyclic, homology, homology_part, cohomology, cohomology_part, poincare_series, molien_series, is_simple, is_monomial, is_supersolvable, is_nilpotent, is_perfect, is_polycyclic, is_elementary_abelian, is_pgroup, gens_small, isomorphism_type_info_simple_group. moved all the"named" groups to a new file.
- Nick Alexander (2007-07): move is_isomorphic to isomorphism_to, add from_gap_list
- William Stein (2007-07): put is_isomorphic back (and make it better)
- David Joyner (2007-08): fixed bugs in composition_series, upper/lower_central_series, derived_series,
- David Joyner (2008-06): modified is_normal (reported by W. J. Palenstijn), and added normalizes
- David Joyner (2008-08): Added example to docstring of cohomology.
- Simon King (2009-04): __cmp__ methods for PermutationGroup_generic and PermutationGroup_subgroup
- Nicolas Borie (2009): Added orbit, transversals, stabiliser and strong_generating_system methods
- Christopher Swenson (2012): Added a special case to compute the order efficiently. (This patch Copyright 2012 Google Inc. All Rights Reserved.)
- Javier Lopez Pena (2013): Added conjugacy classes.

REFERENCES:

- Cameron, P., Permutation Groups. New York: Cambridge University Press, 1999.
- Wielandt, H., Finite Permutation Groups. New York: Academic Press, 1964.
- Dixon, J. and Mortimer, B., Permutation Groups, Springer-Verlag, Berlin/New York, 1996.

Note: Though Suzuki groups are okay, Ree groups should *not* be wrapped as permutation groups - the construction is too slow - unless (for small values or the parameter) they are made using explicit generators.

`sage.groups.perm_gps.permgroup.`**`PermutationGroup`**(*gens=None*, *gap_group=None*, *domain=None*, *canonicalize=True*, *category=None*)

Return the permutation group associated to x (typically a list of generators).

INPUT:

- `gens` - list of generators (default: `None`)
- `gap_group` - a gap permutation group (default: `None`)
- `canonicalize` - bool (default: `True`); if `True`, sort generators and remove duplicates

OUTPUT:

- A permutation group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
```

We can also make permutation groups from PARI groups:

```
sage: H = pari('x^4 - 2*x^3 - 2*x + 1').polgalois()
sage: G = PariGroup(H, 4); G
PARI group [8, -1, 3, "D(4)"] of degree 4
```

```
sage: H = PermutationGroup(G); H          # optional - database_gap
Transitive group number 3 of degree 4
sage: H.gens()                            # optional - database_gap
[(1,2,3,4), (1,3)]
```

We can also create permutation groups whose generators are Gap permutation objects:

```
sage: p = gap('(1,2)(3,7)(4,6)(5,8)'); p
(1,2)(3,7)(4,6)(5,8)
sage: PermutationGroup([p])
Permutation Group with generators [(1,2)(3,7)(4,6)(5,8)]
```

Permutation groups can work on any domain. In the following examples, the permutations are specified in list notation, according to the order of the elements of the domain:

```
sage: list(PermutationGroup([['b','c','a']], domain=['a','b','c']))
[(), ('a','b','c'), ('a','c','b')]
sage: list(PermutationGroup([['b','c','a']], domain=['b','c','a']))
[()]
sage: list(PermutationGroup([['b','c','a']], domain=['a','c','b']))
[(), ('a','b')]
```

There is an underlying gap object that implements each permutation group:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G._gap_()
Group( [ (1,2,3,4) ] )
sage: gap(G)
Group( [ (1,2,3,4) ] )
sage: gap(G) is G._gap_()
True
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: current_randstate().set_seed_gap()
sage: G._gap_().DerivedSeries()
[ Group( [ (3,4), (1,2,3)(4,5) ] ), Group( [ (1,5)(3,4), (1,5)(2,4), (1,5,3) ] ) ]
```

TESTS:

```
sage: r = Permutation("(1,7,9,3)(2,4,8,6)")
sage: f = Permutation("(1,3)(4,6)(7,9)")
sage: PermutationGroup([r,f]) #See Trac #12597
Permutation Group with generators [(1,3)(4,6)(7,9), (1,7,9,3)(2,4,8,6)]

sage: PermutationGroup(SymmetricGroup(5))
Traceback (most recent call last):
...
TypeError: gens must be a tuple, list, or GapElement
```

class `sage.groups.perm_gps.permgroup.`**`PermutationGroup_generic`**(*gens=None, gap_group=None, canonicalize=True, domain=None, category=None*)

Bases: `sage.groups.old.FiniteGroup`

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
```

```
sage: G.center()
Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by [()]
sage: G.group_id()          # optional - database_gap
[120, 34]
sage: n = G.order(); n
120
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: TestSuite(G).run()
```

as_finitely_presented_group(*reduced=False*)

Return a finitely presented group isomorphic to `self`.

This method acts as wrapper for the GAP function `IsomorphismFpGroupByGenerators`, which yields an isomorphism from a given group to a finitely presented group.

INPUT:

- `reduced` – Default `False`, if `True` `FinitelyPresentedGroup.simplified` is called, attempting to simplify the presentation of the finitely presented group to be returned.

OUTPUT:

Finite presentation of self, obtained by taking the image of the isomorphism returned by the GAP function, `IsomorphismFpGroupByGenerators`.

ALGORITHM:

Uses GAP.

EXAMPLES:

```
sage: CyclicPermutationGroup(50).as_finitely_presented_group()
Finitely presented group < a | a^50 >
sage: DihedralGroup(4).as_finitely_presented_group()
Finitely presented group < a, b | b^2, a^4, (b*a)^2 >
sage: GeneralDihedralGroup([2,2]).as_finitely_presented_group()
Finitely presented group < a, b, c | a^2, b^2, c^2, (c*b)^2, (c*a)^2, (b*a)^2 >
```

GAP algorithm is not guaranteed to produce minimal or canonical presentation:

```
sage: G = PermutationGroup(['(1,2,3,4,5)', '(1,5)(2,4)'])
sage: G.is_isomorphic(DihedralGroup(5))
True
sage: K = G.as_finitely_presented_group(); K
Finitely presented group < a, b | b^2, (b*a)^2, b*a^-3*b*a^2 >
sage: K.as_permutation_group().is_isomorphic(DihedralGroup(5))
True
```

We can attempt to reduce the output presentation:

```
sage: PermutationGroup(['(1,2,3,4,5)','(1,3,5,2,4)']).as_finitely_presented_group()
Finitely presented group < a, b | b^-2*a^-1, b*a^-2 >
sage: PermutationGroup(['(1,2,3,4,5)','(1,3,5,2,4)']).as_finitely_presented_group(reduced=Tr
Finitely presented group < a | a^5 >
```

TESTS:

```
sage: PermutationGroup([]).as_finitely_presented_group()
Finitely presented group < a | a >
sage: S = SymmetricGroup(6)
sage: perm_ls = [S.random_element() for i in range(3)]
sage: G = PermutationGroup(perm_ls)
```

```
sage: G.as_finitely_presented_group().as_permutation_group().is_isomorphic(G)
True
```

D_9 is the only non-Abelian group of order 18 with an automorphism group of order 54 [THOMAS-WOODS]:

```
sage: D = DihedralGroup(9).as_finitely_presented_group().gap()
sage: D.Order(), D.IsAbelian(), D.AutomorphismGroup().Order()
(18, false, 54)
```

S_3 is the only non-Abelian group of order 6 [THOMAS-WOODS]:

```
sage: S = SymmetricGroup(3).as_finitely_presented_group().gap()
sage: S.Order(), S.IsAbelian()
(6, false)
```

We can manually construct a permutation representation using GAP coset enumeration methods:

```
sage: D = GeneralDihedralGroup([3,3,4]).as_finitely_presented_group().gap()
sage: ctab = D.CosetTable(D.Subgroup([]))
sage: gen_ls = gap.List(ctab, gap.PermList)
sage: PermutationGroup(gen_ls).is_isomorphic(GeneralDihedralGroup([3,3,4]))
True
sage: A = AlternatingGroup(5).as_finitely_presented_group().gap()
sage: ctab = A.CosetTable(A.Subgroup([]))
sage: gen_ls = gap.List(ctab, gap.PermList)
sage: PermutationGroup(gen_ls).is_isomorphic(AlternatingGroup(5))
True
```

AUTHORS:

- Davis Shurbert (2013-06-21): initial version

base (*seed=None*)

Returns a (minimum) base of this permutation group. A base B of a permutation group is a subset of the domain of the group such that the only group element stabilizing all of B is the identity.

The argument $seed$ is optional and must be a subset of the domain of $base$. When used, an attempt to create a base containing all or part of $seed$ will be made.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3),(6,7,8)])
sage: G.base()
[1, 6]
sage: G.base([2])
[2, 6]

sage: H = PermutationGroup([('a','b','c'),('a','y')])
sage: H.base()
['a', 'b', 'c']

sage: S = SymmetricGroup(13)
sage: S.base()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

sage: S = MathieuGroup(12)
sage: S.base()
[1, 2, 3, 4, 5]
sage: S.base([1,3,5,7,9,11]) # create a base for M12 with only odd integers
[1, 3, 5, 7, 9]
```

blocks_all(*representatives=True*)

Returns the list of block systems of imprimitivity.

For more information on primitivity, see the Wikipedia article on primitive group actions.

INPUT:

- `representative` (boolean) – whether to return all possible block systems of imprimitivity or only one of their representatives (the block can be obtained from its representative set S by computing the orbit of S under `self`).

  This parameter is set to `True` by default (as it is GAP's default behaviour).

OUTPUT:

This method returns a description of *all* block systems. Hence, the output is a "list of lists of lists" or a "list of lists" depending on the value of `representatives`. A bit more clearly, output is :

- A list of length (#number of different block systems) of
  - block systems, each of them being defined as
    - If `representatives = True` : a list of representatives of each set of the block system
    - If `representatives = False` : a partition of the elements defining an imprimitivity block.

See also:

- `is_primitive()`

EXAMPLE:

Picking an interesting group:

```
sage: g = graphs.DodecahedralGraph()
sage: g.is_vertex_transitive()
True
sage: ag = g.automorphism_group()
sage: ag.is_primitive()
False
```

Computing its blocks representatives:

```
sage: ag.blocks_all()
[[0, 15]]
```

Now the full block:

```
sage: sorted(ag.blocks_all(representatives = False)[0])
[[0, 15], [1, 16], [2, 12], [3, 13], [4, 9], [5, 10], [6, 11], [7, 18], [8, 17], [14, 19]]
```

TESTS:

```
sage: g = PermutationGroup([("a","b","c","d")])
sage: g.blocks_all()
[['a', 'c']]
sage: g.blocks_all(False)
[[['a', 'c'], ['b', 'd']]]
```

cardinality()

Return the number of elements of this group. See also: G.degree()

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.order()
12
sage: G = PermutationGroup([()])
sage: G.order()
1
sage: G = PermutationGroup([])
sage: G.order()
1
```

center()

Return the subgroup of elements that commute with every element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G.center()
Subgroup of (Permutation Group with generators [(1,2,3,4)]) generated by [(1,2,3,4)]
sage: G = PermutationGroup([[(1,2,3,4)], [(1,2)]])
sage: G.center()
Subgroup of (Permutation Group with generators [(1,2), (1,2,3,4)]) generated by [()]
```

centralizer(*g*)

Returns the centralizer of `g` in `self`.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: g = G([(1,3)])
sage: G.centralizer(g)
Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]) generated by [(2,4),
sage: g = G([(1,2,3,4)])
sage: G.centralizer(g)
Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]) generated by [(1,2,3
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: G.centralizer(H)
Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]) generated by [(1,2,3
```

character(*values*)

Returns a group character from `values`, where `values` is a list of the values of the character evaluated on the conjugacy classes.

EXAMPLES:

```
sage: G = AlternatingGroup(4)
sage: n = len(G.conjugacy_classes_representatives())
sage: G.character([1]*n)
Character of Alternating group of order 4!/2 as a permutation group
```

character_table()

Returns the matrix of values of the irreducible characters of a permutation group G at the conjugacy classes of G. The columns represent the conjugacy classes of G and the rows represent the different irreducible characters in the ordering given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: G.order()
12
sage: G.character_table()
```

```
[         1          1          1          1]
[         1 -zeta3 - 1      zeta3          1]
[         1      zeta3 -zeta3 - 1          1]
[         3          0          0         -1]
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: CT = gap(G).CharacterTable()
```

Type `print gap.eval("Display(%s)"%CT.name())` to display this nicely.

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.order()
8
sage: G.character_table()
[ 1  1  1  1  1]
[ 1 -1 -1  1  1]
[ 1 -1  1 -1  1]
[ 1  1 -1 -1  1]
[ 2  0  0  0 -2]
sage: CT = gap(G).CharacterTable()
```

Again, type `print gap.eval("Display(%s)"%CT.name())` to display this nicely.

```
sage: SymmetricGroup(2).character_table()
[ 1 -1]
[ 1  1]
sage: SymmetricGroup(3).character_table()
[ 1 -1  1]
[ 2  0 -1]
[ 1  1  1]
sage: SymmetricGroup(5).character_table()
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
sage: list(AlternatingGroup(6).character_table())
[(1, 1, 1, 1, 1, 1, 1), (5, 1, 2, -1, -1, 0, 0), (5, 1, -1, 2, -1, 0, 0), (8, 0, -1, -1, 0,
```

Suppose that you have a class function $f(g)$ on G and you know the values $v_1, \ldots, v_n$ on the conjugacy class elements in `conjugacy_classes_representatives(G)` = $[g_1, \ldots, g_n]$. Since the irreducible characters $\rho_1, \ldots, \rho_n$ of G form an E-basis of the space of all class functions (E a "sufficiently large" cyclotomic field), such a class function is a linear combination of these basis elements, $f = c_1\rho_1 + \cdots + c_n\rho_n$. To find the coefficients c_i, you simply solve the linear system `character_table_values(G)` $[v_1, ..., v_n] = [c_1, ..., c_n]$, where $[v_1, \ldots, v_n]$ = `character_table_values(G)` $^{(-1)}[c_1, ..., c_n]$.

AUTHORS:

- David Joyner and William Stein (2006-01-04)

cohomology(*n*, *p=0*)

Computes the group cohomology $H^n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `GroupHomology` function, written by Graham Ellis.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: G.cohomology(1,2)                            # optional - gap_packages
Multiplicative Abelian group isomorphic to C2
sage: G = SymmetricGroup(3)
sage: G.cohomology(5)                              # optional - gap_packages
Trivial Abelian group
sage: G.cohomology(5,2)                            # optional - gap_packages
Multiplicative Abelian group isomorphic to C2
sage: G.homology(5,3)                              # optional - gap_packages
Trivial Abelian group
sage: G.homology(5,4)                              # optional - gap_packages
Traceback (most recent call last):
...
ValueError: p must be 0 or prime
```

This computes $H^4(S_3, \mathbf{Z})$ and $H^4(S_3, \mathbf{Z}/2\mathbf{Z})$, respectively.

AUTHORS:

- David Joyner and Graham Ellis

REFERENCES:

- G. Ellis, 'Computing group resolutions', J. Symbolic Computation. Vol.38, (2004)1077-1118 (Available at http://hamilton.nuigalway.ie/).
- D. Joyner, 'A primer on computational group homology and cohomology', http://front.math.ucdavis.edu/0706.0549.

cohomology_part (*n*, *p=0*)

Computes the p-part of the group cohomology $H^n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's Homology function, written by Graham Ellis, applied to the p-Sylow subgroup of G.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.cohomology_part(7,2)                   # optional - gap_packages
Multiplicative Abelian group isomorphic to C2 x C2 x C2
sage: G = SymmetricGroup(3)
sage: G.cohomology_part(2,3)                   # optional - gap_packages
Multiplicative Abelian group isomorphic to C3
```

AUTHORS:

- David Joyner and Graham Ellis

commutator (*other=None*)

Returns the commutator subgroup of a group, or of a pair of groups.

INPUT:

- `other` - default: `None` - a permutation group.

OUTPUT:

Let G denote `self`. If `other` is `None` then this method returns the subgroup of G generated by the set of commutators,

$$\{[g_1, g_2] | g_1, g_2 \in G\} = \{g_1^{-1} g_2^{-1} g_1 g_2 | g_1, g_2 \in G\}$$

Let H denote `other`, in the case that it is not `None`. Then this method returns the group generated by the set of commutators,

$$\{[g,h]|g \in G\, h \in H\} = \{g^{-1}h^{-1}gh|g \in G\, h \in H\}$$

The two groups need only be permutation groups, there is no notion of requiring them to explicitly be subgroups of some other group.

Note: For the identical statement, the generators of the returned group can vary from one execution to the next.

EXAMPLES:

```
sage: G = DiCyclicGroup(4)
sage: G.commutator()
Permutation Group with generators [(1,3,5,7)(2,4,6,8)(9,11,13,15)(10,12,14,16)]

sage: G = SymmetricGroup(5)
sage: H = CyclicPermutationGroup(5)
sage: C = G.commutator(H)
sage: C.is_isomorphic(AlternatingGroup(5))
True
```

An abelian group will have a trivial commutator.

```
sage: G = CyclicPermutationGroup(10)
sage: G.commutator()
Permutation Group with generators [()]
```

The quotient of a group by its commutator is always abelian.

```
sage: G = DihedralGroup(20)
sage: C = G.commutator()
sage: Q = G.quotient(C)
sage: Q.is_abelian()
True
```

When forming commutators from two groups, the order of the groups does not matter.

```
sage: D = DihedralGroup(3)
sage: S = SymmetricGroup(2)
sage: C1 = D.commutator(S); C1
Permutation Group with generators [(1,2,3)]
sage: C2 = S.commutator(D); C2
Permutation Group with generators [(1,3,2)]
sage: C1 == C2
True
```

This method calls two different functions in GAP, so this tests that their results are consistent. The commutator groups may have different generators, but the groups are equal.

```
sage: G = DiCyclicGroup(3)
sage: C = G.commutator(); C
Permutation Group with generators [(5,7,6)]
sage: CC = G.commutator(G); CC
Permutation Group with generators [(5,6,7)]
sage: C == CC
True
```

The second group is checked.

```
sage: G = SymmetricGroup(2)
sage: G.commutator('junk')
Traceback (most recent call last):
...
TypeError: junk is not a permutation group
```

composition_series()

Return the composition series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.composition_series()  # random output
[Permutation Group with generators [(1,2,3)(4,5), (3,4)], Permutation Group with generators
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: CS = G.composition_series()
sage: CS[3]
Subgroup of (Permutation Group with generators [(1,2), (1,2,3)(4,5)]) generated by [()]
```

conjugacy_class(g)

Return the conjugacy class of `g` inside the group `self`.

INPUT:

- `g` – an element of the permutation group `self`

OUTPUT:

The conjugacy class of `g` in the group `self`. If `self` is the group denoted by G, this method computes the set $\{x^{-1}gx \mid x \in G\}$

EXAMPLES:

```
sage: G = DihedralGroup(3)
sage: g = G.gen(0)
sage: G.conjugacy_class(g)
Conjugacy class of (1,2,3) in Dihedral group of order 6 as a permutation group
```

conjugacy_classes()

Return a list with all the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = DihedralGroup(3)
sage: G.conjugacy_classes()
[Conjugacy class of () in Dihedral group of order 6 as a permutation group,
 Conjugacy class of (2,3) in Dihedral group of order 6 as a permutation group,
 Conjugacy class of (1,2,3) in Dihedral group of order 6 as a permutation group]
```

conjugacy_classes_representatives()

Returns a complete list of representatives of conjugacy classes in a permutation group G. The ordering is that given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: cl = G.conjugacy_classes_representatives(); cl
[(), (2,4), (1,2)(3,4), (1,2,3,4), (1,3)(2,4)]
```

```
sage: cl[3] in G
True

sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes_representatives()
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3)(4,5), (1,2,3,4), (1,2,3,4,5)]

sage: S = SymmetricGroup(['a','b','c'])
sage: S.conjugacy_classes_representatives()
[(), ('a','b'), ('a','b','c')]
```

AUTHORS:

- David Joyner and William Stein (2006-01-04)

conjugacy_classes_subgroups()

Returns a complete list of representatives of conjugacy classes of subgroups in a permutation group G. The ordering is that given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: cl = G.conjugacy_classes_subgroups()
sage: cl
[Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]) generated by [()],

sage: G = SymmetricGroup(3)
sage: G.conjugacy_classes_subgroups()
[Subgroup of (Symmetric group of order 3! as a permutation group) generated by [()], Subgrou
```

AUTHORS:

- David Joyner (2006-10)

conjugate(g)

Returns the group formed by conjugating `self` with `g`.

INPUT:

- `g` - a permutation group element, or an object that converts to a permutation group element, such as a list of integers or a string of cycles.

OUTPUT:

If `self` is the group denoted by H, then this method computes the group

$$g^{-1}Hg = \{g^{-1}hg | h \in H\}$$

which is the group H conjugated by g.

There are no restrictions on `self` and `g` belonging to a common permutation group, and correspondingly, there is no relationship (such as a common parent) between `self` and the output group.

EXAMPLES:

```
sage: G = DihedralGroup(6)
sage: a = PermutationGroupElement("(1,2,3,4)")
sage: G.conjugate(a)
Permutation Group with generators [(1,4)(2,6)(3,5), (1,5,6,2,3,4)]
```

The element performing the conjugation can be specified in several ways.

```
sage: G = DihedralGroup(6)
sage: strng = "(1,2,3,4)"
sage: G.conjugate(strng)
Permutation Group with generators [(1,4)(2,6)(3,5), (1,5,6,2,3,4)]
sage: G = DihedralGroup(6)
sage: lst = [2,3,4,1]
sage: G.conjugate(lst)
Permutation Group with generators [(1,4)(2,6)(3,5), (1,5,6,2,3,4)]
sage: G = DihedralGroup(6)
sage: cycles = [(1,2,3,4)]
sage: G.conjugate(cycles)
Permutation Group with generators [(1,4)(2,6)(3,5), (1,5,6,2,3,4)]
```

Conjugation is a group automorphism, so conjugate groups will be isomorphic.

```
sage: G = DiCyclicGroup(6)
sage: G.degree()
11
sage: cycle = [i+1 for i in range(1,11)] + [1]
sage: C = G.conjugate(cycle)
sage: G.is_isomorphic(C)
True
```

The conjugating element may be from a symmetric group with larger degree than the group being conjugated.

```
sage: G = AlternatingGroup(5)
sage: G.degree()
5
sage: g = "(1,3)(5,6,7)"
sage: H = G.conjugate(g); H
Permutation Group with generators [(1,4,6,3,2), (1,4,6)]
sage: H.degree()
6
```

The conjugating element is checked.

```
sage: G = SymmetricGroup(3)
sage: G.conjugate("junk")
Traceback (most recent call last):
...
TypeError: junk does not convert to a permutation group element
```

construction()

EXAMPLES:

```
sage: P1 = PermutationGroup([[(1,2)]])
sage: P1.construction()
(PermutationGroupFunctor[(1,2)], Permutation Group with generators [()])

sage: PermutationGroup([]).construction() is None
True
```

This allows us to perform computations like the following:

```
sage: P1 = PermutationGroup([[(1,2)]]); p1 = P1.gen()
sage: P2 = PermutationGroup([[(1,3)]]); p2 = P2.gen()
sage: p = p1*p2; p
(1,2,3)
sage: p.parent()
```

```
Permutation Group with generators [(1,2), (1,3)]
sage: p.parent().domain()
{1, 2, 3}
```

Note that this will merge permutation groups with different domains:

```
sage: g1 = PermutationGroupElement([(1,2),(3,4,5)])
sage: g2 = PermutationGroup([('a','b')], domain=['a', 'b']).gens()[0]
sage: g2
('a','b')
sage: p = g1*g2; p
(1,2)(3,4,5)('a','b')
```

cosets (*S*, *side='right'*)

Returns a list of the cosets of `S` in `self`.

INPUT:

- `S` - a subgroup of `self`. An error is raised if `S` is not a subgroup.
- `side` - default: 'right' - determines if right cosets or left cosets are returned. `side` refers to where the representative is placed in the products forming the cosets and thus allowable values are only 'right' and 'left'.

OUTPUT:

A list of lists. Each inner list is a coset of the subgroup in the group. The first element of each coset is the smallest element (based on the ordering of the elements of `self`) of all the group elements that have not yet appeared in a previous coset. The elements of each coset are in the same order as the subgroup elements used to build the coset's elements.

As a consequence, the subgroup itself is the first coset, and its first element is the identity element. For each coset, the first element listed is the element used as a representative to build the coset. These representatives form an increasing sequence across the list of cosets, and within a coset the representative is the smallest element of its coset (both orderings are based on of the ordering of elements of `self`).

In the case of a normal subgroup, left and right cosets should appear in the same order as part of the outer list. However, the list of the elements of a particular coset may be in a different order for the right coset versus the order in the left coset. So, if you check to see if a subgroup is normal, it is necessary to sort each individual coset first (but not the list of cosets, due to the ordering of the representatives). See below for examples of this.

Note: This is a naive implementation intended for instructional purposes, and hence is slow for larger groups. Sage and GAP provide more sophisticated functions for working quickly with cosets of larger groups.

EXAMPLES:

The default is to build right cosets. This example works with the symmetry group of an 8-gon and a normal subgroup. Notice that a straight check on the equality of the output is not sufficient to check normality, while sorting the individual cosets is sufficient to then simply test equality of the list of lists. Study the second coset in each list to understand the need for sorting the elements of the cosets.

```
sage: G = DihedralGroup(8)
sage: quarter_turn = G('(1,3,5,7)(2,4,6,8)'); quarter_turn
(1,3,5,7)(2,4,6,8)
sage: S = G.subgroup([quarter_turn])
sage: rc = G.cosets(S); rc
[[(), (1,3,5,7)(2,4,6,8), (1,5)(2,6)(3,7)(4,8), (1,7,5,3)(2,8,6,4)],
 [(2,8)(3,7)(4,6), (1,7)(2,6)(3,5), (1,5)(2,4)(6,8), (1,3)(4,8)(5,7)],
```

```
 [(1,2)(3,8)(4,7)(5,6), (1,8)(2,7)(3,6)(4,5), (1,6)(2,5)(3,4)(7,8), (1,4)(2,3)(5,8)(6,7)],
 [(1,2,3,4,5,6,7,8), (1,4,7,2,5,8,3,6), (1,6,3,8,5,2,7,4), (1,8,7,6,5,4,3,2)]]
sage: lc = G.cosets(S, side='left'); lc
[[(), (1,3,5,7)(2,4,6,8), (1,5)(2,6)(3,7)(4,8), (1,7,5,3)(2,8,6,4)],
 [(2,8)(3,7)(4,6), (1,3)(4,8)(5,7), (1,5)(2,4)(6,8), (1,7)(2,6)(3,5)],
 [(1,2)(3,8)(4,7)(5,6), (1,4)(2,3)(5,8)(6,7), (1,6)(2,5)(3,4)(7,8), (1,8)(2,7)(3,6)(4,5)],
 [(1,2,3,4,5,6,7,8), (1,4,7,2,5,8,3,6), (1,6,3,8,5,2,7,4), (1,8,7,6,5,4,3,2)]]

sage: S.is_normal(G)
True
sage: rc == lc
False
sage: rc_sorted = [sorted(c) for c in rc]
sage: lc_sorted = [sorted(c) for c in lc]
sage: rc_sorted == lc_sorted
True
```

An example with the symmetry group of a regular tetrahedron and a subgroup that is not normal. Thus, the right and left cosets are different (and so are the representatives). With each individual coset sorted, a naive test of normality is possible.

```
sage: A = AlternatingGroup(4)
sage: face_turn = A('(1,2,3)'); face_turn
(1,2,3)
sage: stabilizer = A.subgroup([face_turn])
sage: rc = A.cosets(stabilizer, side='right'); rc
[[(), (1,2,3), (1,3,2)],
 [(2,3,4), (1,3)(2,4), (1,4,2)],
 [(2,4,3), (1,4,3), (1,2)(3,4)],
 [(1,2,4), (1,4)(2,3), (1,3,4)]]
sage: lc = A.cosets(stabilizer, side='left'); lc
[[(), (1,2,3), (1,3,2)],
 [(2,3,4), (1,2)(3,4), (1,3,4)],
 [(2,4,3), (1,2,4), (1,3)(2,4)],
 [(1,4,2), (1,4,3), (1,4)(2,3)]]

sage: stabilizer.is_normal(A)
False
sage: rc_sorted = [sorted(c) for c in rc]
sage: lc_sorted = [sorted(c) for c in lc]
sage: rc_sorted == lc_sorted
False
```

TESTS:

The keyword `side` is checked for the two possible values.

```
sage: G = SymmetricGroup(3)
sage: S = G.subgroup([G("(1,2)")])
sage: G.cosets(S, side='junk')
Traceback (most recent call last):
...
ValueError: side should be 'right' or 'left', not junk
```

The subgroup argument is checked to see if it is a permutation group. Even a legitimate GAP object can be rejected.

```
sage: G=SymmetricGroup(3)
sage: G.cosets(gap(3))
```

```
Traceback (most recent call last):
...
TypeError: 3 is not a permutation group
```

The subgroup is verified as a subgroup of `self`.

```
sage: A = AlternatingGroup(3)
sage: G = SymmetricGroup(3)
sage: S = G.subgroup([G("(1,2)")])
sage: A.cosets(S)
Traceback (most recent call last):
...
ValueError: Subgroup of (Symmetric group of order 3! as a permutation group) generated by [(
```

AUTHOR:

- Rob Beezer (2011-01-31)

degree()

Returns the degree of this permutation group.

EXAMPLES:

```
sage: S = SymmetricGroup(['a','b','c'])
sage: S.degree()
3
sage: G = PermutationGroup([(1,3),(4,5)])
sage: G.degree()
5
```

Note that you can explicitly specify the domain to get a permutation group of smaller degree:

```
sage: G = PermutationGroup([(1,3),(4,5)], domain=[1,3,4,5])
sage: G.degree()
4
```

derived_series()

Return the derived series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.derived_series()  # random output
[Permutation Group with generators [(1,2,3)(4,5), (3,4)], Permutation Group with generators
```

direct_product(*other*, *maps=True*)

Wraps GAP's `DirectProduct`, `Embedding`, and `Projection`.

Sage calls GAP's `DirectProduct`, which chooses an efficient representation for the direct product. The direct product of permutation groups will be a permutation group again. For a direct product `D`, the GAP operation `Embedding(D,i)` returns the homomorphism embedding the i-th factor into `D`. The GAP operation `Projection(D,i)` gives the projection of `D` onto the i-th factor. This method returns a 5-tuple: a permutation group and 4 morphisms.

INPUT:

- `self, other` - permutation groups

OUTPUT:

- D - a direct product of the inputs, returned as a permutation group as well
- iota1 - an embedding of self into D
- iota2 - an embedding of other into D
- pr1 - the projection of D onto self (giving a splitting 1 - other - D - self - 1)
- pr2 - the projection of D onto other (giving a splitting 1 - self - D - other - 1)

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: D = G.direct_product(G,False)
sage: D
Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
sage: D,iota1,iota2,pr1,pr2 = G.direct_product(G)
sage: D; iota1; iota2; pr1; pr2
Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
  Defn: Embedding( Group( [ (1,2,3,4), (5,6,7,8) ] ), 1 )
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
  Defn: Embedding( Group( [ (1,2,3,4), (5,6,7,8) ] ), 2 )
Permutation group morphism:
  From: Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
  To:   Cyclic group of order 4 as a permutation group
  Defn: Projection( Group( [ (1,2,3,4), (5,6,7,8) ] ), 1 )
Permutation group morphism:
  From: Permutation Group with generators [(5,6,7,8), (1,2,3,4)]
  To:   Cyclic group of order 4 as a permutation group
  Defn: Projection( Group( [ (1,2,3,4), (5,6,7,8) ] ), 2 )
sage: g=D([(1,3),(2,4)]); g
(1,3)(2,4)
sage: d=D([(1,4,3,2),(5,7),(6,8)]); d
(1,4,3,2)(5,7)(6,8)
sage: iota1(g); iota2(g); pr1(d); pr2(d)
(1,3)(2,4)
(5,7)(6,8)
(1,4,3,2)
(1,3)(2,4)
```

domain()

Returns the underlying set that this permutation group acts on.

EXAMPLES:

```
sage: P = PermutationGroup([(1,2),(3,5)])
sage: P.domain()
{1, 2, 3, 4, 5}
sage: S = SymmetricGroup(['a', 'b', 'c'])
sage: S.domain()
{'a', 'b', 'c'}
```

exponent()

Computes the exponent of the group. The exponent e of a group G is the LCM of the orders of its elements, that is, e is the smallest integer such that $g^e = 1$ for all $g \in G$.

EXAMPLES:

```
sage: G = AlternatingGroup(4)
sage: G.exponent()
6
```

fitting_subgroup()

Returns the Fitting subgroup of `self`. The Fitting subgroup of a group G is the largest nilpotent normal subgroup of G.

EXAMPLES:

```
sage: G=PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.fitting_subgroup()
Subgroup of (Permutation Group with generators [(2,4), (1,2,3,4)]) generated by [(2,4), (1,2
sage: G=PermutationGroup([[(1,2,3,4)],[(1,2)]])
sage: G.fitting_subgroup()
Subgroup of (Permutation Group with generators [(1,2), (1,2,3,4)]) generated by [(1,2)(3,4),
```

fixed_points()

Return the list of points fixed by `self`, i.e., the subset of `.domain()` not moved by any element of `self`.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3)])
sage: G.fixed_points()
[]
sage: G = PermutationGroup([(1,2,3),(5,6)])
sage: G.fixed_points()
[4]
sage: G = PermutationGroup([[(1,4,7)],[(4,3),(6,7)]])
sage: G.fixed_points()
[2, 5]
```

frattini_subgroup()

Returns the Frattini subgroup of `self`. The Frattini subgroup of a group G is the intersection of all maximal subgroups of G.

EXAMPLES:

```
sage: G=PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.frattini_subgroup()
Subgroup of (Permutation Group with generators [(2,4), (1,2,3,4)]) generated by [(1,3)(2,4)]
sage: G=SymmetricGroup(4)
sage: G.frattini_subgroup()
Subgroup of (Symmetric group of order 4! as a permutation group) generated by [()]
```

gen(*i=None*)

Returns the i-th generator of `self`; that is, the i-th element of the list `self.gens()`.

The argument i may be omitted if there is only one generator (but this will raise an error otherwise).

EXAMPLES:

We explicitly construct the alternating group on four elements:

```
sage: A4 = PermutationGroup([[(1,2,3)],[(2,3,4)]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.gens()
[(2,3,4), (1,2,3)]
sage: A4.gen(0)
(2,3,4)
```

```
sage: A4.gen(1)
(1,2,3)
sage: A4.gens()[0]; A4.gens()[1]
(2,3,4)
(1,2,3)

sage: P1 = PermutationGroup([[(1,2)]]); P1.gen()
(1,2)
```

gens()

Return tuple of generators of this group. These need not be minimal, as they are the generators used in defining this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3)], [(1,2)]])
sage: G.gens()
[(1,2), (1,2,3)]
```

Note that the generators need not be minimal, though duplicates are removed:

```
sage: G = PermutationGroup([[(1,2)], [(1,3)], [(2,3)], [(1,2)]])
sage: G.gens()
[(2,3), (1,2), (1,3)]
```

We can use index notation to access the generators returned by `self.gens`:

```
sage: G = PermutationGroup([[(1,2,3,4), (5,6)], [(1,2)]])
sage: g = G.gens()
sage: g[0]
(1,2)
sage: g[1]
(1,2,3,4)(5,6)
```

TESTS:

We make sure that the trivial group gets handled correctly:

```
sage: SymmetricGroup(1).gens()
[()]
```

gens_small()

For this group, returns a generating set which has few elements. As neither irredundancy nor minimal length is proven, it is fast.

EXAMPLES:

```
sage: R = "(25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24)" ## R = right
sage: U = "( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19)" ## U = top
sage: L = "( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35)" ## L = left
sage: F = "(17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11)" ## F = front
sage: B = "(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27)" ## B = back or
sage: D = "(41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40)" ## D = down or
sage: G = PermutationGroup([R,L,U,F,B,D])
sage: len(G.gens_small())
2
```

The output may be unpredictable, due to the use of randomized algorithms in GAP. Note that both the following answers are equally valid.

```
sage: G = PermutationGroup([[('a','b')], [('b', 'c')], [('a', 'c')]])
sage: G.gens_small() # random
[('b','c'), ('a','c','b')] ## (on 64-bit Linux)
[('a','b'), ('a','c','b')] ## (on Solaris)
sage: len(G.gens_small()) == 2
True
```

group_id()

Return the ID code of this group, which is a list of two integers. Requires "optional" database_gap-4.4.x package.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.group_id()     # optional - database_gap
[12, 4]
```

group_primitive_id()

Return the index of this group in the GAP database of primitive groups.

Requires "optional" database_gap-4.4.x package.

OUTPUT:

A positive integer, following GAP's conventions. A `ValueError` is raised if the group is not primitive.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4,5)], [(1,5),(2,4)]])
sage: G.group_primitive_id()     # optional - database_gap
2
sage: G.degree()
5
```

From the information of the degree and the identification number, you can recover the isomorphism class of your group in the GAP database:

```
sage: H = PrimitiveGroup(5,2)  # optional - database_gap
sage: G == H                     # optional - database_gap
False
sage: G.is_isomorphic(H)         # optional - database_gap
True
```

has_element(*item*)

Returns boolean value of `item in self` - however *ignores* parentage.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)]); g
(1,2,3,4)
sage: G.has_element(g)
True
sage: h = H([(1,2),(3,4)]); h
(1,2)(3,4)
sage: G.has_element(h)
False
```

holomorph()

The holomorph of a group as a permutation group.

The holomorph of a group G is the semidirect product $G \rtimes_{id} Aut(G)$, where id is the identity function on $Aut(G)$, the automorphism group of G.

See Wikipedia article Holomorph (mathematics)

OUTPUT:

Returns the holomorph of a given group as permutation group via a wrapping of GAP's semidirect product function.

EXAMPLES:

Thomas and Wood's 'Group Tables' (Shiva Publishing, 1980) tells us that the holomorph of C_5 is the unique group of order 20 with a trivial center.

```
sage: C5 = CyclicPermutationGroup(5)
sage: A = C5.holomorph()
sage: A.order()
20
sage: A.is_abelian()
False
sage: A.center()
Subgroup of (Permutation Group with generators
[(5,6,7,8,9), (1,2,4,3)(6,7,9,8)]) generated by [()]
sage: A
Permutation Group with generators [(5,6,7,8,9), (1,2,4,3)(6,7,9,8)]
```

Noting that the automorphism group of D_4 is itself D_4, it can easily be shown that the holomorph is indeed an internal semidirect product of these two groups.

```
sage: D4 = DihedralGroup(4)
sage: H = D4.holomorph()
sage: H.gens()
[(3,8)(4,7), (2,3,5,8), (2,5)(3,8), (1,4,6,7)(2,3,5,8), (1,8)(2,7)(3,6)(4,5)]
sage: G = H.subgroup([H.gens()[0],H.gens()[1],H.gens()[2]])
sage: N = H.subgroup([H.gens()[3],H.gens()[4]])
sage: N.is_normal(H)
True
sage: G.is_isomorphic(D4)
True
sage: N.is_isomorphic(D4)
True
sage: G.intersection(N)
Permutation Group with generators [()]
sage: L = [H(x)*H(y) for x in G for y in N]; L.sort()
sage: L1 = H.list(); L1.sort()
sage: L == L1
True
```

Author:

- Kevin Halasz (2012-08-14)

homology(*n*, *p=0*)

Computes the group homology $H_n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `GroupHomology` function, written by Graham Ellis.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

AUTHORS:

•David Joyner and Graham Ellis

The example below computes $H_7(S_5, \mathbf{Z})$, $H_7(S_5, \mathbf{Z}/2\mathbf{Z})$, $H_7(S_5, \mathbf{Z}/3\mathbf{Z})$, and $H_7(S_5, \mathbf{Z}/5\mathbf{Z})$, respectively. To compute the 2-part of $H_7(S_5, \mathbf{Z})$, use the `homology_part` function.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.homology(7)                              # optional - gap_packages
Multiplicative Abelian group isomorphic to C2 x C2 x C4 x C3 x C5
sage: G.homology(7,2)                              # optional - gap_packages
Multiplicative Abelian group isomorphic to C2 x C2 x C2 x C2 x C2
sage: G.homology(7,3)                              # optional - gap_packages
Multiplicative Abelian group isomorphic to C3
sage: G.homology(7,5)                              # optional - gap_packages
Multiplicative Abelian group isomorphic to C5
```

REFERENCES:

•G. Ellis, "Computing group resolutions", J. Symbolic Computation. Vol.38, (2004)1077-1118 (Available at http://hamilton.nuigalway.ie/.

•D. Joyner, "A primer on computational group homology and cohomology", http://front.math.ucdavis.edu/0706.0549

homology_part (*n*, *p=0*)

Computes the p-part of the group homology $H_n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `Homology` function, written by Graham Ellis, applied to the p-Sylow subgroup of G.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.homology_part(7,2)                              # optional - gap_packages
Multiplicative Abelian group isomorphic to C2 x C2 x C2 x C2 x C4
```

AUTHORS:

•David Joyner and Graham Ellis

id()

(Same as `self.group_id()`.) Return the ID code of this group, which is a list of two integers. Requires "optional" database_gap-4.4.x package.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.group_id()     # optional - database_gap
[12, 4]
```

identity()

Return the identity element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)]])
sage: e = G.identity()
sage: e
()
sage: g = G.gen(0)
sage: g*e
```

```
(1,2,3)(4,5)
sage: e*g
(1,2,3)(4,5)

sage: S = SymmetricGroup(['a','b','c'])
sage: S.identity()
()
```

intersection(*other*)

Returns the permutation group that is the intersection of `self` and `other`.

INPUT:

- `other` - a permutation group.

OUTPUT:

A permutation group that is the set-theoretic intersection of `self` with `other`. The groups are viewed as subgroups of a symmetric group big enough to contain both group's symbol sets. So there is no strict notion of the two groups being subgroups of a common parent.

EXAMPLES:

```
sage: H = DihedralGroup(4)

sage: K = CyclicPermutationGroup(4)
sage: H.intersection(K)
Permutation Group with generators [(1,2,3,4)]

sage: L = DihedralGroup(5)
sage: H.intersection(L)
Permutation Group with generators [(1,4)(2,3)]

sage: M = PermutationGroup(["()"])
sage: H.intersection(M)
Permutation Group with generators [()]
```

Some basic properties.

```
sage: H = DihedralGroup(4)
sage: L = DihedralGroup(5)
sage: H.intersection(L) == L.intersection(H)
True
sage: H.intersection(H) == H
True
```

The group `other` is verified as such.

```
sage: H = DihedralGroup(4)
sage: H.intersection('junk')
Traceback (most recent call last):
...
TypeError: junk is not a permutation group
```

irreducible_characters()

Returns a list of the irreducible characters of `self`.

EXAMPLES:

```
sage: irr = SymmetricGroup(3).irreducible_characters()
sage: [x.values() for x in irr]
```

```
[[1, -1, 1], [2, 0, -1], [1, 1, 1]]
```

is_abelian()

Return True if this group is abelian.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_abelian()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_abelian()
True
```

is_commutative()

Return True if this group is commutative.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_commutative()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_commutative()
True
```

is_cyclic()

Return True if this group is cyclic.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_cyclic()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_cyclic()
True
```

is_elementary_abelian()

Return True if this group is elementary abelian. An elementary abelian group is a finite abelian group, where every nontrivial element has order p, where p is a prime.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_elementary_abelian()
False
sage: G = PermutationGroup(['(1,2,3)','(4,5,6)'])
sage: G.is_elementary_abelian()
True
```

is_isomorphic(*right*)

Return True if the groups are isomorphic.

INPUT:

- self - this group
- right - a permutation group

OUTPUT:

•boolean; `True` if `self` and `right` are isomorphic groups; `False` otherwise.

EXAMPLES:

```
sage: v = ['(1,2,3)(4,5)', '(1,2,3,4,5)']
sage: G = PermutationGroup(v)
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_isomorphic(H)
False
sage: G.is_isomorphic(G)
True
sage: G.is_isomorphic(PermutationGroup(list(reversed(v))))
True
```

is_monomial()

Returns `True` if the group is monomial. A finite group is monomial if every irreducible complex character is induced from a linear character of a subgroup.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_monomial()
True
```

is_nilpotent()

Return `True` if this group is nilpotent.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_nilpotent()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_nilpotent()
True
```

is_normal(*other*)

Return `True` if this group is a normal subgroup of `other`.

EXAMPLES:

```
sage: AlternatingGroup(4).is_normal(SymmetricGroup(4))
True
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H.is_normal(G)
False
```

is_perfect()

Return `True` if this group is perfect. A group is perfect if it equals its derived subgroup.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_perfect()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_perfect()
False
```

is_pgroup()

Returns `True` if this group is a p-group. A finite group is a p-group if its order is of the form p^n for a prime integer p and a nonnegative integer n.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3,4,5)'])
sage: G.is_pgroup()
True
```

is_polycyclic()

Return `True` if this group is polycyclic. A group is polycyclic if it has a subnormal series with cyclic factors. (For finite groups, this is the same as if the group is solvable - see `is_solvable`.)

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_polycyclic()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_polycyclic()
True
```

is_primitive(*domain=None*)

Returns `True` if `self` acts primitively on `domain`. A group G acts primitively on a set S if

1. G acts transitively on S and
2. the action induces no non-trivial block system on S.

INPUT:

- `domain` (optional)

See also:

- `blocks_all()`

EXAMPLES:

By default, test for primitivity of `self` on its domain:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(1,2)]])
sage: G.is_primitive()
True
sage: G = PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.is_primitive()
False
```

You can specify a domain on which to test primitivity:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(2,4)]])
sage: G.is_primitive([1..4])
False
sage: G.is_primitive([1,2,3])
True
sage: G = PermutationGroup([[(3,4,5,6)],[(3,4)]]) #S_4 on [3..6]
sage: G.is_primitive(G.non_fixed_points())
True
```

is_regular(*domain=None*)

Returns `True` if `self` acts regularly on `domain`. A group G acts regularly on a set S if

1. G acts transitively on S and

2. G acts semi-regularly on S.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G.is_regular()
True
sage: G = PermutationGroup([[(1,2,3,4)],[(5,6)]])
sage: G.is_regular()
False
```

You can pass in a domain on which to test regularity:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(5,6)]])
sage: G.is_regular([1..4])
True
sage: G.is_regular(G.non_fixed_points())
False
```

is_semi_regular(*domain=None*)

Returns `True` if `self` acts semi-regularly on `domain`. A group G acts semi-regularly on a set S if the point stabilizers of S in G are trivial.

`domain` is optional and may take several forms. See examples.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G.is_semi_regular()
True
sage: G = PermutationGroup([[(1,2,3,4)],[(5,6)]])
sage: G.is_semi_regular()
False
```

You can pass in a domain to test semi-regularity:

```
sage: G = PermutationGroup([[(1,2,3,4)],[(5,6)]])
sage: G.is_semi_regular([1..4])
True
sage: G.is_semi_regular(G.non_fixed_points())
False
```

is_simple()

Returns `True` if the group is simple. A group is simple if it has no proper normal subgroups.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_simple()
False
```

is_solvable()

Returns `True` if the group is solvable.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_solvable()
True
```

is_subgroup(*other*)

Returns `True` if `self` is a subgroup of `other`.

EXAMPLES:

```
sage: G = AlternatingGroup(5)
sage: H = SymmetricGroup(5)
sage: G.is_subgroup(H)
True
```

is_supersolvable()

Returns `True` if the group is supersolvable. A finite group is supersolvable if it has a normal series with cyclic factors.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_supersolvable()
True
```

is_transitive(*domain=None*)

Returns `True` if `self` acts transitively on `domain`. A group G acts transitively on set S if for all $x, y \in S$ there is some $g \in G$ such that $x^g = y$.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.is_transitive()
True
sage: G = PermutationGroup(['(1,2)(3,4)(5,6)'])
sage: G.is_transitive()
False

sage: G = PermutationGroup([[(1,2,3,4,5)],[(1,2)]]) #S_5 on [1..5]
sage: G.is_transitive([1,4,5])
True
sage: G.is_transitive([2..6])
False
sage: G.is_transitive(G.non_fixed_points())
True
sage: H = PermutationGroup([[(1,2,3)],[(4,5,6)]])
sage: H.is_transitive(H.non_fixed_points())
False
```

Note that this differs from the definition in GAP, where `IsTransitive` returns whether the group is transitive on the set of points moved by the group.

```
sage: G = PermutationGroup([(2,3)])
sage: G.is_transitive()
False
sage: gap(G).IsTransitive()
true
```

isomorphism_to(*right*)

Return an isomorphism from `self` to `right` if the groups are isomorphic, otherwise `None`.

INPUT:

- `self` - this group
- `right` - a permutation group

OUTPUT:

- `None` or a morphism of permutation groups.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.isomorphism_to(H) is None
True
sage: G = PermutationGroup([(1,2,3), (2,3)])
sage: H = PermutationGroup([(1,2,4), (1,4)])
sage: G.isomorphism_to(H)  # not tested, see below
Permutation group morphism:
  From: Permutation Group with generators [(2,3), (1,2,3)]
  To:   Permutation Group with generators [(1,2,4), (1,4)]
  Defn: [(2,3), (1,2,3)] -> [(2,4), (1,2,4)]
```

TESTS:

Partial check that the output makes some sense:

```
sage: G.isomorphism_to(H)
Permutation group morphism:
  From: Permutation Group with generators [(2,3), (1,2,3)]
  To:   Permutation Group with generators [(1,2,4), (1,4)]
  Defn: [(2,3), (1,2,3)] -> [...]
```

isomorphism_type_info_simple_group()

If the group is simple, then this returns the name of the group.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(5)
sage: G.isomorphism_type_info_simple_group()
rec(
  name := "Z(5)",
  parameter := 5,
  series := "Z" )
```

TESTS:

This shows that the issue at trac ticket 7360 is fixed:

```
sage: G = KleinFourGroup()
sage: G.is_simple()
False
sage: G.isomorphism_type_info_simple_group()
Traceback (most recent call last):
...
TypeError: Group must be simple.
```

largest_moved_point()

Return the largest point moved by a permutation in this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.largest_moved_point()
4
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.largest_moved_point()
10

sage: G = PermutationGroup([[('a','b','c'),('d','e')]])
sage: G.largest_moved_point()
```

```
'e'
```

Warning: The name of this function is not good; this function should be deprecated in term of degree:

```
sage: P = PermutationGroup([[1,2,3,4]])
sage: P.largest_moved_point()
4
sage: P.cardinality()
1
```

list()

Return list of all elements of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)], [(1,2)]])
sage: G.list()
[(), (1,2), (1,2,3,4), (1,3)(2,4), (1,3,4), (2,3,4), (1,4,3,2),
 (1,3,2,4), (1,3,4,2), (1,2,4,3), (1,4,2,3), (2,4,3), (1,4,3),
 (1,4)(2,3), (1,4,2), (1,3,2), (1,3), (3,4), (2,4), (1,4), (2,3),
 (1,2)(3,4), (1,2,3), (1,2,4)]

sage: G = PermutationGroup([[('a','b')]], domain=('a', 'b')); G
Permutation Group with generators [('a','b')]
sage: G.list()
[(), ('a','b')]
```

lower_central_series()

Return the lower central series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.lower_central_series()  # random output
[Permutation Group with generators [(1,2,3)(4,5), (3,4)], Permutation Group with generators
```

minimal_generating_set()

Return a minimal generating set

EXAMPLE:

```
sage: g = graphs.CompleteGraph(4)
sage: g.relabel(['a','b','c','d'])
sage: mgs = g.automorphism_group().minimal_generating_set(); len(mgs)
2
sage: mgs # random
[('b','d','c'), ('a','c','b','d')]
```

TESTS:

```
sage: PermutationGroup(["(1,2,3)(4,5,6)","(1,2,3,4,5,6)"]).minimal_generating_set()
[(2,5)(3,6), (1,5,3,4,2,6)]
```

molien_series()

Return the Molien series of a permutation group. The function

$$M(x) = (1/|G|) \sum_{g \in G} \det(1 - x * g)^{-1}$$

is sometimes called the "Molien series" of G. GAP's `MolienSeries` is associated to a character of a group G. How are these related? A group G, given as a permutation group on n points, has a "natural" representation of dimension n, given by permutation matrices. The Molien series of G is the one associated to that permutation representation of G using the above formula. Character values then count fixed points of the corresponding permutations.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.molien_series()
1/(-x^15 + x^14 + x^13 - x^10 - x^9 - x^8 + x^7 + x^6 + x^5 - x^2 - x + 1)
sage: G = SymmetricGroup(3)
sage: G.molien_series()
1/(-x^6 + x^5 + x^4 - x^2 - x + 1)
```

Some further tests (after trac ticket #15817):

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: S4ms = SymmetricGroup(4).molien_series()
sage: G.molien_series() / S4ms
x^5 + 2*x^4 + x^3 + x^2 + 1
```

This works for not-transitive groups:

```
sage: G = PermutationGroup([[(1,2)],[(3,4)]])
sage: G.molien_series() / S4ms
x^4 + x^3 + 2*x^2 + x + 1
```

This works for groups with fixed points:

```
sage: G = PermutationGroup([[(2,)]])
sage: G.molien_series()
1/(x^2 - 2*x + 1)
```

non_fixed_points()

Return the list of points not fixed by `self`, i.e., the subset of `self.domain()` moved by some element of `self`.

EXAMPLES:

```
sage: G = PermutationGroup([[(3,4,5)],[(7,10)]])
sage: G.non_fixed_points()
[3, 4, 5, 7, 10]
sage: G = PermutationGroup([[(2,3,6)],[(9,)]]) # note: 9 is fixed
sage: G.non_fixed_points()
[2, 3, 6]
```

normal_subgroups()

Return the normal subgroups of this group as a (sorted in increasing order) list of permutation groups.

The normal subgroups of $H = PSL(2,7) \times PSL(2,7)$ are 1, two copies of $PSL(2,7)$ and H itself, as the following example shows.

EXAMPLES:

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
```

```
sage: H = D[0]
sage: NH = H.normal_subgroups()
sage: len(NH)
4
sage: NH[1].is_isomorphic(G)
True
sage: NH[2].is_isomorphic(G)
True
```

normalizer(*g*)
Returns the normalizer of `g` in `self`.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: g = G([(1,3)])
sage: G.normalizer(g)
Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]) generated by [(2,4),
sage: g = G([(1,2,3,4)])
sage: G.normalizer(g)
Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]) generated by [(2,4),
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: G.normalizer(H)
Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]) generated by [(2,4),
```

normalizes(*other*)
Returns `True` if the group `other` is normalized by `self`. Wraps GAP's `IsNormal` function.

A group G normalizes a group U if and only if for every $g \in G$ and $u \in U$ the element u^g is a member of U. Note that U need not be a subgroup of G.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: H = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H.normalizes(G)
False
sage: G = SymmetricGroup(3)
sage: H = PermutationGroup( [ (4,5,6) ] )
sage: G.normalizes(H)
True
sage: H.normalizes(G)
True
```

In the last example, G and H are disjoint, so each normalizes the other.

orbit(*point*, *action='OnPoints'*)
Return the orbit of a point under a group action.

INPUT:

- `point` – can be a point or any of the list above, depending on the action to be considered.
- `action` – string. if `point` is an element from the domain, a tuple of elements of the domain, a tuple of tuples [...], this variable describes how the group is acting.

  The actions currently available through this method are `"OnPoints"`, `"OnTuples"`, `"OnSets"`, `"OnPairs"`, `"OnSetsSets"`, `"OnSetsDisjointSets"`, `"OnSetsTuples"`, `"OnTuplesSets"`, `"OnTuplesTuples"`. They are taken from GAP's list of group actions, see `gap.help('Group Actions')`.

It is set to "OnPoints" by default. See below for examples.

OUTPUT:

The orbit of point as a tuple. Each entry is an image under the action of the permutation group, if necessary converted to the corresponding container. That is, if action='OnSets' then each entry will be a set even if point was given by a list/tuple/iterable.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
sage: G.orbit(3)
(3, 4, 1)
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.orbit(3)
(3, 4, 10, 1, 2)
sage: G = PermutationGroup([ [('c','d')], [('a','c')] ])
sage: G.orbit('a')
('a', 'c', 'd')
```

Action of S_3 on sets:

```
sage: S3 = groups.permutation.Symmetric(3)
sage: S3.orbit((1,2), action = "OnSets")
({1, 2}, {2, 3}, {1, 3})
```

On tuples:

```
sage: S3.orbit((1,2), action = "OnTuples")
((1, 2), (2, 3), (2, 1), (3, 1), (1, 3), (3, 2))
```

Action of S_4 on sets of disjoint sets:

```
sage: S4 = groups.permutation.Symmetric(4)
sage: S4.orbit(((1,2),(3,4)), action = "OnSetsDisjointSets")
({{1, 2}, {3, 4}}, {{2, 3}, {1, 4}}, {{1, 3}, {2, 4}})
```

Action of S_4 (on a nonstandard domain) on tuples of sets:

```
sage: S4 = PermutationGroup([ [('c','d')], [('a','c')], [('a','b')] ])
sage: S4.orbit((('a','c'),('b','d')),"OnTuplesSets")
(({'a', 'c'}, {'b', 'd'}),
 ({'a', 'd'}, {'c', 'b'}),
 ({'c', 'b'}, {'a', 'd'}),
 ({'b', 'd'}, {'a', 'c'}),
 ({'c', 'd'}, {'a', 'b'}),
 ({'a', 'b'}, {'c', 'd'}))
```

Action of S_4 (on a very nonstandard domain) on tuples of sets:

```
sage: S4 = PermutationGroup([ [((11,(12,13)),'d')],
...           [((12,(12,11)),(11,(12,13)))], [((12,(12,11)),'b')] ])
sage: S4.orbit((( (11,(12,13)), (12,(12,11))),('b','d')),"OnTuplesSets")
(({(11, (12, 13)), (12, (12, 11))}, {'b', 'd'}),
 ({'d', (12, (12, 11))}, {(11, (12, 13)), 'b'}),
 ({(11, (12, 13)), 'b'}, {'d', (12, (12, 11))}),
 ({(11, (12, 13)), 'd'}, {'b', (12, (12, 11))}),
 ({'b', 'd'}, {(11, (12, 13)), (12, (12, 11))}),
 ({'b', (12, (12, 11))}, {(11, (12, 13)), 'd'}))
```

orbits()

Returns the orbits of the elements of the domain under the default group action.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
sage: G.orbits()
[[1, 3, 4], [2]]
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.orbits()
[[1, 2, 3, 4, 10], [5], [6], [7], [8], [9]]

sage: G = PermutationGroup([ [('c','d')], [('a','c')],[('b',)]])
sage: G.orbits()
[['a', 'c', 'd'], ['b']]
```

The answer is cached:

```
sage: G.orbits() is G.orbits()
True
```

AUTHORS:

- Nathan Dunfield

order()

Return the number of elements of this group. See also: G.degree()

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.order()
12
sage: G = PermutationGroup([()])
sage: G.order()
1
sage: G = PermutationGroup([])
sage: G.order()
1
```

poincare_series(*p=2*, *n=10*)

Returns the Poincare series of $G \mod p$ ($p \geq 2$ must be a prime), for n large. In other words, if you input a finite group G, a prime p, and a positive integer n, it returns a quotient of polynomials $f(x) = P(x)/Q(x)$ whose coefficient of x^k equals the rank of the vector space $H_k(G, \mathbf{Z}/p\mathbf{Z})$, for all k in the range $1 \leq k \leq n$.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.poincare_series(2,10)                              # optional - gap_packages
(x^2 + 1)/(x^4 - x^3 - x + 1)
sage: G = SymmetricGroup(3)
sage: G.poincare_series(2,10)                              # optional - gap_packages
1/(-x + 1)
```

AUTHORS:

- David Joyner and Graham Ellis

quotient(*N*)

Returns the quotient of this permutation group by the normal subgroup N, as a permutation group.

Wraps the GAP operator "/".

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3), (2,3)])
sage: N = PermutationGroup([(1,2,3)])
sage: G.quotient(N)
Permutation Group with generators [(1,2)]
sage: G.quotient(G)
Permutation Group with generators [()]
```

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: a = G.random_element()
sage: a in G
True
sage: a.parent() is G
True
sage: a^6
()
```

representative_action(*x*, *y*)

Return an element of self that maps x to y if it exists.

This method wraps the gap function `RepresentativeAction`, which can also return elements that map a given set of points on another set of points.

INPUT:

- `x, y` – two elements of the domain.

EXAMPLE:

```
sage: G = groups.permutation.Cyclic(14)
sage: g = G.representative_action(1,10)
sage: all(g(x) == 1+((x+9-1)%14) for x in G.domain())
True
```

TESTS:

```
sage: g = graphs.PetersenGraph()
sage: g.relabel(list("abcdefghik"))
sage: g.vertices()
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k']
sage: ag = g.automorphism_group()
sage: a = ag.representative_action('a','b')
sage: g == g.relabel(a,inplace=False)
True
sage: a('a') == 'b'
True
```

semidirect_product(*N*, *mapping*, *check=True*)

The semidirect product of `self` with `N`.

INPUT:

- `N` - A group which is acted on by `self` and naturally embeds as a normal subgroup of the returned semidirect product.
- `mapping` - A pair of lists that together define a homomorphism, ϕ : self $\rightarrow$ Aut(N), by giving, in the second list, the images of the generators of `self` in the order given in the first list.

- check - A boolean that, if set to False, will skip the initial tests which are made on mapping. This may be beneficial for large N, since in such cases the injectivity test can be expensive. Set to True by default.

OUTPUT:

The semidirect product of self and N defined by the action of self on N given in mapping (note that a homomorphism from A to the automorphism group of B is equivalent to an action of A on the B's underlying set). The semidirect product of two groups, H and N, is a construct similar to the direct product in so far as the elements are the Cartesian product of the elements of H and the elements of N. The operation, however, is built upon an action of H on N, and is defined as such:

$$(h_1, n_1)(h_2, n_2) = (h_1 h_2, n_1^{h_2} n_2)$$

This function is a wrapper for GAP's SemidirectProduct command. The permutation group returned is built upon a permutation representation of the semidirect product of self and N on a set of size $\mid N \mid$. The generators of N are given as their right regular representations, while the generators of self are defined by the underlying action of self on N. It should be noted that the defining action is not always faithful, and in this case the inputted representations of the generators of self are placed on additional letters and adjoined to the output's generators of self.

EXAMPLES:

Perhaps the most common example of a semidirect product comes from the family of dihedral groups. Each dihedral group is the semidirect product of C_2 with C_n, where, by convention, $3 \leq n$. In this case, the nontrivial element of C_2 acts on C_n so as to send each element to its inverse.

```
sage: C2 = CyclicPermutationGroup(2)
sage: C8 = CyclicPermutationGroup(8)
sage: alpha = PermutationGroupMorphism_im_gens(C8,C8,[(1,8,7,6,5,4,3,2)])
sage: S = C2.semidirect_product(C8,[[(1,2)],[alpha]])
sage: S == DihedralGroup(8)
False
sage: S.is_isomorphic(DihedralGroup(8))
True
sage: S.gens()
[(3,4,5,6,7,8,9,10), (1,2)(4,10)(5,9)(6,8)]
```

A more complicated example can be drawn from [THOMAS-WOODS]. It is there given that a semidirect product of D_4 and C_3 is isomorphic to one of C_2 and the dicyclic group of order 12. This nonabelian group of order 24 has very similar structure to the dicyclic and dihedral groups of order 24, the three being the only groups of order 24 with a two-element center and 9 conjugacy classes.

```
sage: D4 = DihedralGroup(4)
sage: C3 = CyclicPermutationGroup(3)
sage: alpha1 = PermutationGroupMorphism_im_gens(C3,C3,[(1,3,2)])
sage: alpha2 = PermutationGroupMorphism_im_gens(C3,C3,[(1,2,3)])
sage: S1 = D4.semidirect_product(C3,[[(1,2,3,4),(1,3)],[alpha1,alpha2]])
sage: C2 = CyclicPermutationGroup(2)
sage: Q = DiCyclicGroup(3)
sage: a = Q.gens()[0]; b=Q.gens()[1].inverse()
sage: alpha = PermutationGroupMorphism_im_gens(Q,Q,[a,b])
sage: S2 = C2.semidirect_product(Q,[[(1,2)],[alpha]])
sage: S1.is_isomorphic(S2)
True
sage: S1.is_isomorphic(DihedralGroup(12))
False
sage: S1.is_isomorphic(DiCyclicGroup(6))
False
sage: S1.center()
```

```
Subgroup of (Permutation Group with generators
[(5,6,7), (1,2,3,4)(6,7), (1,3)]) generated by [(1,3)(2,4)]
sage: len(S1.conjugacy_classes_representatives())
9
```

If your normal subgroup is large, and you are confident that your inputs will successfully create a semidirect product, then it is beneficial, for the sake of time efficiency, to set the `check` parameter to `False`.

```
sage: C2 = CyclicPermutationGroup(2)
sage: C2000 = CyclicPermutationGroup(500)
sage: alpha = PermutationGroupMorphism(C2000,C2000,[C2000.gen().inverse()])
sage: S = C2.semidirect_product(C2000,[[(1,2)],[alpha]],check=False)
```

TESTS:

```
sage: C3 = CyclicPermutationGroup(3)
sage: D4 = DihedralGroup(4)
sage: alpha = PermutationGroupMorphism(C3,C3,[C3("(1,3,2)")])
sage: alpha1 = PermutationGroupMorphism(C3,C3,[C3("(1,2,3)")])

sage: s = D4.semidirect_product('junk', [[(1,2,3,4),(1,2)], [alpha, alpha1]])
Traceback (most recent call last):
...
TypeError: junk is not a permutation group

sage: s = D4.semidirect_product(C3, [[(1,2,3,4),(1,2)], [alpha, alpha1]])
Traceback (most recent call last):
...
ValueError: the generator list must generate the calling group, [(1, 2, 3, 4), (1, 2)]
does not generate Dihedral group of order 8 as a permutation group

sage: s = D4.semidirect_product(C3, [[(1,2,3,4),(1,3)], [alpha]])
Traceback (most recent call last):
...
ValueError: the list of generators and the list of morphisms must be of equal length

sage: alpha2 = PermutationGroupMorphism(C3, D4, [D4("()")])
sage: s = D4.semidirect_product(C3, [[(1,2,3,4),(1,3)], [alpha, alpha2]])
Traceback (most recent call last):
...
ValueError: an element of the automorphism list is not an endomorphism (and is therefore not

sage: alpha3 = PermutationGroupMorphism(C3,C3,[C3("()")])
sage: s = D4.semidirect_product(C3, [[(1,2,3,4),(1,3)], [alpha, alpha3]])
Traceback (most recent call last):
...
ValueError: an element of the automorphism list is not an injection (and is therefore not an
```

REFERENCES:

AUTHOR:

- Kevin Halasz (2012-8-12)

smallest_moved_point()

Return the smallest point moved by a permutation in this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(3,4)], [(2,3,4)]])
sage: G.smallest_moved_point()
2
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.smallest_moved_point()
1
```

Note that this function uses the ordering from the domain:

```
sage: S = SymmetricGroup(['a','b','c'])
sage: S.smallest_moved_point()
'a'
```

socle()

Returns the socle of `self`. The socle of a group G is the subgroup generated by all minimal normal subgroups.

EXAMPLES:

```
sage: G=SymmetricGroup(4)
sage: G.socle()
Subgroup of (Symmetric group of order 4! as a permutation group) generated by [(1,2)(3,4), (
sage: G.socle().socle()
Subgroup of (Subgroup of (Symmetric group of order 4! as a permutation group) generated by [
```

solvable_radical()

Returns the solvable radical of `self`. The solvable radical (or just radical) of a group G is the largest solvable normal subgroup of G.

EXAMPLES:

```
sage: G=SymmetricGroup(4)
sage: G.solvable_radical()
Subgroup of (Symmetric group of order 4! as a permutation group) generated by [(1,2), (1,2,3
sage: G=SymmetricGroup(5)
sage: G.solvable_radical()
Subgroup of (Symmetric group of order 5! as a permutation group) generated by [()]
```

stabilizer(*point*, *action='OnPoints'*)

Return the subgroup of `self` which stabilize the given position. `self` and its stabilizers must have same degree.

INPUT:

- `point` – a point of the `domain()`, or a set of points depending on the value of `action`.
- `action` (string; default `"OnPoints"`) – should the group be considered to act on points (`action="OnPoints"`) or on sets of points (`action="OnSets"`) ? In the latter case, the first argument must be a subset of `domain()`.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
sage: G.stabilizer(1)
Subgroup of (Permutation Group with generators [(3,4), (1,3)]) generated by [(3,4)]
sage: G.stabilizer(3)
Subgroup of (Permutation Group with generators [(3,4), (1,3)]) generated by [(1,4)]
```

The stabilizer of a set of points:

```
sage: s10 = groups.permutation.Symmetric(10)
sage: s10.stabilizer([1..3],"OnSets").cardinality()
30240
sage: factorial(3)*factorial(7)
30240

sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.stabilizer(10)
Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4,10)]) generated by [(2,
sage: G.stabilizer(1)
Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4,10)]) generated by [(2,
sage: G = PermutationGroup([[(2,3,4)],[(6,7)]])
sage: G.stabilizer(1)
Subgroup of (Permutation Group with generators [(6,7), (2,3,4)]) generated by [(6,7), (2,3,4
sage: G.stabilizer(2)
Subgroup of (Permutation Group with generators [(6,7), (2,3,4)]) generated by [(6,7)]
sage: G.stabilizer(3)
Subgroup of (Permutation Group with generators [(6,7), (2,3,4)]) generated by [(6,7)]
sage: G.stabilizer(4)
Subgroup of (Permutation Group with generators [(6,7), (2,3,4)]) generated by [(6,7)]
sage: G.stabilizer(5)
Subgroup of (Permutation Group with generators [(6,7), (2,3,4)]) generated by [(6,7), (2,3,4
sage: G.stabilizer(6)
Subgroup of (Permutation Group with generators [(6,7), (2,3,4)]) generated by [(2,3,4)]
sage: G.stabilizer(7)
Subgroup of (Permutation Group with generators [(6,7), (2,3,4)]) generated by [(2,3,4)]
sage: G.stabilizer(8)
Traceback (most recent call last):
...
ValueError: 8 does not belong to the domain

sage: G = PermutationGroup([ [('c','d')], [('a','c')] ], domain='abcd')
sage: G.stabilizer('a')
Subgroup of (Permutation Group with generators [('c','d'), ('a','c')]) generated by [('c','d
sage: G.stabilizer('b')
Subgroup of (Permutation Group with generators [('c','d'), ('a','c')]) generated by [('c','d
sage: G.stabilizer('c')
Subgroup of (Permutation Group with generators [('c','d'), ('a','c')]) generated by [('a','d
sage: G.stabilizer('d')
Subgroup of (Permutation Group with generators [('c','d'), ('a','c')]) generated by [('a','c
```

TESTS:

```
sage: G.stabilizer(['a'],"OnMonkeys")
Traceback (most recent call last):
...
ValueError: 'action' must be equal to 'OnPoints' or to 'OnSets'.
```

strong_generating_system(*base_of_group=None*)

Return a Strong Generating System of `self` according the given base for the right action of `self` on itself.

`base_of_group` is a list of the positions on which `self` acts, in any order. The algorithm returns a list of transversals and each transversal is a list of permutations. By default, `base_of_group` is `[1, 2, 3, ..., d]` where d is the degree of the group.

For `base_of_group` = $[\mathrm{pos}_1, \mathrm{pos}_2, \dots, \mathrm{pos}_d]$ let G_i be the subgroup of G = `self` which stabilizes

$\mathrm{pos}_1, \mathrm{pos}_2, \dots, \mathrm{pos}_i$, so

$$G = G_0 \supset G_1 \supset G_2 \supset \cdots \supset G_n = \{e\}$$

Then the algorithm returns $[G_i.\mathrm{transversals}(\mathrm{pos}_{i+1})]_{1 \leq i \leq n}$

INPUT:

- `base_of_group` (optional) – default: `[1, 2, 3, ..., d]` – a list containing the integers $1, 2, \dots, d$ in any order (d is the degree of `self`)

OUTPUT:

- A list of lists of permutations from the group, which form a strong generating system.

EXAMPLES:

```
sage: G = PermutationGroup([[(7,8)],[(3,4)],[(4,5)]])
sage: G.strong_generating_system()
[[()], [()], [(), (3,4,5), (3,5)], [(), (4,5)], [()], [()], [(), (7,8)], [()]]
sage: G = PermutationGroup([[(1,2,3,4)],[(1,2)]])
sage: G.strong_generating_system()
[[(), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3)], [(), (2,3,4), (2,4,3)], [(), (3,4)], [()]]
sage: G = PermutationGroup([[(1,2,3)],[(4,5,7)],[(1,4,6)]])
sage: G.strong_generating_system()
[[(), (1,2,3), (1,4,6), (1,3,2), (1,5,7,4,6), (1,6,4), (1,7,5,4,6)], [(), (2,6,3), (2,5,7,6,
sage: G = PermutationGroup([[(1,2,3)],[(2,3,4)],[(3,4,5)]])
sage: G.strong_generating_system([5,4,3,2,1])
[[(), (1,5,3,4,2), (1,5,4,3,2), (1,5)(2,3), (1,5,2)], [(), (1,3)(2,4), (1,2)(3,4), (1,4)(2,3
sage: G = PermutationGroup([[(3,4)]])
sage: G.strong_generating_system()
[[()], [()], [(), (3,4)], [()]]
sage: G.strong_generating_system(base_of_group=[3,1,2,4])
[[(), (3,4)], [()], [()], [()]]
sage: G = TransitiveGroup(12,17)                # optional - database_gap
sage: G.strong_generating_system()              # optional - database_gap
[[(), (1,4,11,2)(3,6,5,8)(7,10,9,12), (1,8,3,2)(4,11,10,9)(5,12,7,6), (1,7)(2,8)(3,9)(4,10)(
```

TESTS:

```
sage: G = SymmetricGroup(10)
sage: H = PermutationGroup([G.random_element() for i in range(randrange(1,3,1))])
sage: prod(map(lambda x : len(x), H.strong_generating_system()),1) == H.cardinality()
True
```

structure_description(*G*, *latex=False*)

Return a string that tries to describe the structure of `G`.

This methods wraps GAP's `StructureDescription` method.

Requires the *optional* `database_gap` package.

For full details, including the form of the returned string and the algorithm to build it, see GAP's documentation.

INPUT:

- `latex` – a boolean (default: `False`). If `True` return a LaTeX formatted string.

OUTPUT:

- string

> **Warning:** From GAP's documentation: The string returned by `StructureDescription` is **not** an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()             # optional - database_gap
'C6'
sage: G.structure_description(latex=True)   # optional - database_gap
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True))    # optional - database_gap
C_{6} \times C_{6}
```

This method is mainly intended for small groups or groups with few normal subgroups. Even then there are some surprises:

```
sage: D3 = DihedralGroup(3)
sage: D3.structure_description()     # optional - database_gap
'S3'
```

We use the Sage notation for the degree of dihedral groups:

```
sage: D4 = DihedralGroup(4)
sage: D4.structure_description()     # optional - database_gap
'D4'
```

Works for finitely presented groups (trac ticket #17573):

```
sage: F.<x, y> = FreeGroup()
sage: G=F / [x^2*y^-1, x^3*y^2, x*y*x^-1*y^-1]
sage: G.structure_description()      # optional - database_gap
'C7'
```

And matrix groups (trac ticket #17573):

```
sage: groups.matrix.GL(4,2).structure_description() # optional - database_gap
'A8'
```

subgroup(*gens=None*, *gap_group=None*, *domain=None*, *category=None*, *canonicalize=True*, *check=True*)

Wraps the `PermutationGroup_subgroup` constructor. The argument `gens` is a list of elements of `self`.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3),(3,4,5)])
sage: g = G((1,2,3))
sage: G.subgroup([g])
Subgroup of (Permutation Group with generators [(3,4,5), (1,2,3)]) generated by [(1,2,3)]
```

subgroups()

Returns a list of all the subgroups of `self`.

OUTPUT:

Each possible subgroup of `self` is contained once in the returned list. The list is in order, according to the size of the subgroups, from the trivial subgroup with one element on through up to the whole group. Conjugacy classes of subgroups are contiguous in the list.

> **Warning:** For even relatively small groups this method can take a very long time to execute, or create vast amounts of output. Likely both. Its purpose is instructional, as it can be useful for studying small groups. The 156 subgroups of the full symmetric group on 5 symbols of order 120, S_5, can be computed in about a minute on commodity hardware in 2011. The 64 subgroups of the cyclic group of order $30030 = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$ takes about twice as long.
> For faster results, which still exhibit the structure of the possible subgroups, use `conjugacy_classes_subgroups()`.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: G.subgroups()
[Subgroup of (Symmetric group of order 3! as a permutation group) generated by [()],
 Subgroup of (Symmetric group of order 3! as a permutation group) generated by [(2,3)],
 Subgroup of (Symmetric group of order 3! as a permutation group) generated by [(1,2)],
 Subgroup of (Symmetric group of order 3! as a permutation group) generated by [(1,3)],
 Subgroup of (Symmetric group of order 3! as a permutation group) generated by [(1,2,3)],
 Subgroup of (Symmetric group of order 3! as a permutation group) generated by [(2,3), (1,2,

sage: G = CyclicPermutationGroup(14)
sage: G.subgroups()
[Subgroup of (Cyclic group of order 14 as a permutation group) generated by [()],
 Subgroup of (Cyclic group of order 14 as a permutation group) generated by [(1,8)(2,9)(3,10
 Subgroup of (Cyclic group of order 14 as a permutation group) generated by [(1,3,5,7,9,11,1
 Subgroup of (Cyclic group of order 14 as a permutation group) generated by [(1,2,3,4,5,6,7,
```

AUTHOR:

- Rob Beezer (2011-01-24)

sylow_subgroup(*p*)

Returns a Sylow p-subgroup of the finite group G, where p is a prime. This is a p-subgroup of G whose index in G is coprime to p. Wraps the GAP function `SylowSubgroup`.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)', '(2,3)'])
sage: G.sylow_subgroup(2)
Subgroup of (Permutation Group with generators [(2,3), (1,2,3)]) generated by [(2,3)]
sage: G.sylow_subgroup(5)
Subgroup of (Permutation Group with generators [(2,3), (1,2,3)]) generated by [()]
```

TESTS:

Implementation details should not prevent us from computing large subgroups (trac #5491):

```
sage: PSL(10,2).sylow_subgroup(7)
Subgroup of...
```

transversals(*point*)

If G is a permutation group acting on the set $X = \{1, 2, ..., n\}$ and H is the stabilizer subgroup of <integer>, a right (respectively left) transversal is a set containing exactly one element from each right (respectively left) coset of H. This method returns a right transversal of `self` by the stabilizer of `self` on <integer> position.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
sage: G.transversals(1)
[(), (1,3,4), (1,4,3)]
```

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.transversals(1)
[(), (1,2)(3,4), (1,3,2,10,4), (1,4,2,10,3), (1,10,4,3,2)]

sage: G = PermutationGroup([ [('c','d')], [('a','c')] ])
sage: G.transversals('a')
[(), ('a','c','d'), ('a','d','c')]
```

trivial_character()

Returns the trivial character of `self`.

EXAMPLES:

```
sage: SymmetricGroup(3).trivial_character()
Character of Symmetric group of order 3! as a permutation group
```

upper_central_series()

Return the upper central series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.upper_central_series()
[Subgroup of (Permutation Group with generators [(3,4), (1,2,3)(4,5)]) generated by [()]]
```

class `sage.groups.perm_gps.permgroup.`**PermutationGroup_subgroup**(*ambient*, *gens=None*, *gap_group=None*, *domain=None*, *category=None*, *canonicalize=True*, *check=True*)

Bases: `sage.groups.perm_gps.permgroup.PermutationGroup_generic`

Subgroup subclass of `PermutationGroup_generic`, so instance methods are inherited.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: H.subgroup(gens)
Subgroup of (Dihedral group of order 8 as a permutation group) generated by [(1,2,3,4)]
sage: K = H.subgroup(gens)
sage: K.list()
[(), (1,2,3,4), (1,3)(2,4), (1,4,3,2)]
sage: K.ambient_group()
Dihedral group of order 8 as a permutation group
sage: K.gens()
[(1,2,3,4)]
```

ambient_group()

Return the ambient group related to `self`.

EXAMPLES:

An example involving the dihedral group on four elements, D_8:

```
sage: G = DihedralGroup(4)
sage: H = CyclicPermutationGroup(4)
sage: gens = H.gens()
sage: S = PermutationGroup_subgroup(G, list(gens))
sage: S.ambient_group()
Dihedral group of order 8 as a permutation group
sage: S.ambient_group() == G
True
```

is_normal(*other=None*)

Return `True` if this group is a normal subgroup of `other`. If `other` is not specified, then it is assumed to be the ambient group.

EXAMPLES:

```
sage: S = SymmetricGroup(['a','b','c'])
sage: H = S.subgroup([('a', 'b', 'c')]); H
Subgroup of (Symmetric group of order 3! as a permutation group) generated by [('a','b','c')
sage: H.is_normal()
True
```

sage.groups.perm_gps.permgroup.**direct_product_permgroups**(*P*)

Takes the direct product of the permutation groups listed in `P`.

EXAMPLES:

```
sage: G1 = AlternatingGroup([1,2,4,5])
sage: G2 = AlternatingGroup([3,4,6,7])
sage: D = direct_product_permgroups([G1,G2,G1])
sage: D.order()
1728
sage: D = direct_product_permgroups([G1])
sage: D==G1
True
sage: direct_product_permgroups([])
Symmetric group of order 0! as a permutation group
```

sage.groups.perm_gps.permgroup.**from_gap_list**(*G, src*)

Convert a string giving a list of GAP permutations into a list of elements of `G`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup import from_gap_list
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: L = from_gap_list(G, "[(1,2,3)(4,5), (3,4)]"); L
[(1,2,3)(4,5), (3,4)]
sage: L[0].parent() is G
True
sage: L[1].parent() is G
True
```

sage.groups.perm_gps.permgroup.**hap_decorator**(*f*)

A decorator for permutation group methods that require HAP. It checks to see that HAP is installed as well as checks that the argument `p` is either 0 or prime.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup import hap_decorator
sage: def foo(self, n, p=0): print "Done"
sage: foo = hap_decorator(foo)
sage: foo(None, 3)    #optional - gap_packages
```

```
Done
sage: foo(None, 3, 0) # optional - gap_packages
Done
sage: foo(None, 3, 5) # optional - gap_packages
Done
sage: foo(None, 3, 4) #optional - gap_packages
Traceback (most recent call last):
...
ValueError: p must be 0 or prime
```

sage.groups.perm_gps.permgroup.**load_hap**()

Load the GAP hap package into the default GAP interpreter interface. If this fails, try one more time to load it.

EXAMPLES:

```
sage: sage.groups.perm_gps.permgroup.load_hap() # optional - gap_packages
```

CHAPTER TWENTYEIGHT

“NAMED” PERMUTATION GROUPS (SUCH AS THE SYMMETRIC GROUP, S_N)

You can construct the following permutation groups:

– **SymmetricGroup, S_n of order $n!$ (n can also be a list X of distinct** positive integers, in which case it returns S_X)

– **AlternatingGroup, A_n of order $n!/2$ (n can also be a list X** of distinct positive integers, in which case it returns A_X)

– DihedralGroup, D_n of order $2n$

– GeneralDihedralGroup, $Dih(G)$, where G is an abelian group

– CyclicPermutationGroup, C_n of order n

– DiCyclicGroup, nonabelian groups of order $4m$ with a unique element of order 2

– **TransitiveGroup, n^{th} transitive group of degree d** from the GAP tables of transitive groups (requires the “optional” package database_gap)

– TransitiveGroups(d), TransitiveGroups(), set of all of the above

– **PrimitiveGroup, n^{th} primitive group of degree d** from the GAP tables of primitive groups (requires the “optional” package database_gap)

– PrimitiveGroups(d), PrimitiveGroups(), set of all of the above

– MathieuGroup(degree), Mathieu group of degree 9, 10, 11, 12, 21, 22, 23, or 24.

– KleinFourGroup, subgroup of S_4 of order 4 which is not $C_2 \times C_2$

– QuaternionGroup, non-abelian group of order 8, $\{\pm 1, \pm I, \pm J, \pm K\}$

– SplitMetacyclicGroup, nonabelian groups of order p^m with cyclic subgroups of index p

– SemidihedralGroup, nonabelian 2-groups with cyclic subgroups of index 2

– **PGL(n,q), projective general linear group of $n \times n$ matrices over** the finite field GF(q)

– **PSL(n,q), projective special linear group of $n \times n$ matrices over** the finite field GF(q)

– **PSp(2n,q), projective symplectic linear group of $2n \times 2n$ matrices** over the finite field GF(q)

– **PSU(n,q), projective special unitary group of $n \times n$ matrices having** coefficients in the finite field $GF(q^2)$ that respect a fixed nondegenerate sesquilinear form, of determinant 1.

– **PGU(n,q), projective general unitary group of $n \times n$ matrices having** coefficients in the finite field $GF(q^2)$ that respect a fixed nondegenerate sesquilinear form, modulo the centre.

– SuzukiGroup(q), Suzuki group over GF(q), ${}^2B_2(2^{2k+1}) = Sz(2^{2k+1})$.

AUTHOR:

- David Joyner (2007-06): split from permgp.py (suggested by Nick Alexander)

REFERENCES: Cameron, P., Permutation Groups. New York: Cambridge University Press, 1999. Wielandt, H., Finite Permutation Groups. New York: Academic Press, 1964. Dixon, J. and Mortimer, B., Permutation Groups, Springer-Verlag, Berlin/New York, 1996.

NOTE: Though Suzuki groups are okay, Ree groups should *not* be wrapped as permutation groups - the construction is too slow - unless (for small values or the parameter) they are made using explicit generators.

class `sage.groups.perm_gps.permgroup_named.`**`AlternatingGroup`**`(`*domain=None*`)`

Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_symalt`

The alternating group of order $n!/2$, as a permutation group.

INPUT:

`n` – a positive integer, or list or tuple thereof

Note: This group is also available via `groups.permutation.Alternating()`.

EXAMPLES:

```
sage: G = AlternatingGroup(6)
sage: G.order()
360
sage: G
Alternating group of order 6!/2 as a permutation group
sage: G.category()
Category of finite permutation groups
sage: TestSuite(G).run() # long time

sage: G = AlternatingGroup([1,2,4,5])
sage: G
Alternating group of order 4!/2 as a permutation group
sage: G.domain()
{1, 2, 4, 5}
sage: G.category()
Category of finite permutation groups
sage: TestSuite(G).run()
```

TESTS:

```
sage: groups.permutation.Alternating(6)
Alternating group of order 6!/2 as a permutation group
```

class `sage.groups.perm_gps.permgroup_named.`**`CyclicPermutationGroup`**`(`*n*`)`

Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_unique`

A cyclic group of order n, as a permutation group.

INPUT: n – a positive integer

Note: This group is also available via `groups.permutation.Cyclic()`.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(8)
sage: G.order()
8
sage: G
```

```
Cyclic group of order 8 as a permutation group
sage: G.category()
Category of finite permutation groups
sage: TestSuite(G).run()
sage: C = CyclicPermutationGroup(10)
sage: C.is_abelian()
True
sage: C = CyclicPermutationGroup(10)
sage: C.as_AbelianGroup()
Multiplicative Abelian group isomorphic to C2 x C5
```

TESTS:

```
sage: groups.permutation.Cyclic(6)
Cyclic group of order 6 as a permutation group
```

as_AbelianGroup()

Returns the corresponding Abelian Group instance.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(8)
sage: C.as_AbelianGroup()
Multiplicative Abelian group isomorphic to C8
```

is_abelian()

Return True if this group is abelian.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(8)
sage: C.is_abelian()
True
```

is_commutative()

Return True if this group is commutative.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(8)
sage: C.is_commutative()
True
```

class sage.groups.perm_gps.permgroup_named.**DiCyclicGroup**(*n*)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The dicyclic group of order $4n$, for $n \geq 2$.

INPUT:

- n – a positive integer, two or greater

OUTPUT:

This is a nonabelian group similar in some respects to the dihedral group of the same order, but with far fewer elements of order 2 (it has just one). The permutation representation constructed here is based on the presentation

$$\langle a, x \mid a^{2n} = 1, x^2 = a^n, x^{-1}ax = a^{-1} \rangle$$

For $n = 2$ this is the group of quaternions ($\pm 1, \pm I, \pm J, \pm K$), which is the nonabelian group of order 8 that is not the dihedral group D_4, the symmetries of a square. For $n = 3$ this is the nonabelian group of order 12 that is not the dihedral group D_6 nor the alternating group A_4. This group of order 12 is also the semi-direct product of of C_2 by C_4, $C_3 \rtimes C_4$. [CONRAD2009]

When the order of the group is a power of 2 it is known as a "generalized quaternion group."

IMPLEMENTATION:

The presentation above means every element can be written as $a^i x^j$ with $0 \leq i < 2n$, $j = 0, 1$. We code a^i as the symbol $i+1$ and code $a^i x$ as the symbol $2n+i+1$. The two generators are then represented using a left regular representation.

Note: This group is also available via `groups.permutation.DiCyclic()`.

EXAMPLES:

A dicyclic group of order 384, with a large power of 2 as a divisor:

```
sage: n = 3*2^5
sage: G = DiCyclicGroup(n)
sage: G.order()
384
sage: a = G.gen(0)
sage: x = G.gen(1)
sage: a^(2*n)
()
sage: a^n==x^2
True
sage: x^-1*a*x==a^-1
True
```

A large generalized quaternion group (order is a power of 2):

```
sage: n = 2^10
sage: G=DiCyclicGroup(n)
sage: G.order()
4096
sage: a = G.gen(0)
sage: x = G.gen(1)
sage: a^(2*n)
()
sage: a^n==x^2
True
sage: x^-1*a*x==a^-1
True
```

Just like the dihedral group, the dicyclic group has an element whose order is half the order of the group. Unlike the dihedral group, the dicyclic group has only one element of order 2. Like the dihedral groups of even order, the center of the dicyclic group is a subgroup of order 2 (thus has the unique element of order 2 as its non-identity element).

```
sage: G=DiCyclicGroup(3*5*4)
sage: G.order()
240
sage: two = [g for g in G if g.order()==2]; two
[(1,5)(2,6)(3,7)(4,8)(9,13)(10,14)(11,15)(12,16)]
sage: G.center().order()
2
```

For small orders, we check this is really a group we do not have in Sage otherwise.

```
sage: G = DiCyclicGroup(2)
sage: H = DihedralGroup(4)
sage: G.is_isomorphic(H)
False
sage: G = DiCyclicGroup(3)
sage: H = DihedralGroup(6)
sage: K = AlternatingGroup(6)
sage: G.is_isomorphic(H) or G.is_isomorphic(K)
False
```

TESTS:

```
sage: groups.permutation.DiCyclic(6)
Diyclic group of order 24 as a permutation group
```

REFERENCES:

AUTHOR:

- Rob Beezer (2009-10-18)

is_abelian()

Return True if this group is abelian.

EXAMPLES:

```
sage: D = DiCyclicGroup(12)
sage: D.is_abelian()
False
```

is_commutative()

Return True if this group is commutative.

EXAMPLES:

```
sage: D = DiCyclicGroup(12)
sage: D.is_commutative()
False
```

class sage.groups.perm_gps.permgroup_named.**DihedralGroup**(*n*)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The Dihedral group of order $2n$ for any integer $n \geq 1$.

INPUT:

- n – a positive integer

OUTPUT:

The dihedral group of order $2n$, as a permutation group

Note: This group is also available via groups.permutation.Dihedral().

EXAMPLES:

```
sage: DihedralGroup(1)
Dihedral group of order 2 as a permutation group

sage: DihedralGroup(2)
```

```
Dihedral group of order 4 as a permutation group
sage: DihedralGroup(2).gens()
[(3,4), (1,2)]

sage: DihedralGroup(5).gens()
[(1,2,3,4,5), (1,5)(2,4)]
sage: list(DihedralGroup(5))
[(), (1,5)(2,4), (1,2,3,4,5), (1,4)(2,3), (1,3,5,2,4), (2,5)(3,4),
(1,3)(4,5), (1,5,4,3,2), (1,4,2,5,3), (1,2)(3,5)]

sage: G = DihedralGroup(6)
sage: G.order()
12
sage: G = DihedralGroup(5)
sage: G.order()
10
sage: G
Dihedral group of order 10 as a permutation group
sage: G.gens()
[(1,2,3,4,5), (1,5)(2,4)]

sage: DihedralGroup(0)
Traceback (most recent call last):
...
ValueError: n must be positive
```

TESTS:

```
sage: TestSuite(G).run()
sage: G.category()
Category of finite permutation groups
sage: TestSuite(G).run()

sage: groups.permutation.Dihedral(6)
Dihedral group of order 12 as a permutation group
```

class `sage.groups.perm_gps.permgroup_named.`**`GeneralDihedralGroup`**(*factors*)

Bases: `sage.groups.perm_gps.permgroup.PermutationGroup_generic`

The Generalized Dihedral Group generated by the abelian group with direct factors in the input list.

INPUT:

- `factors` - a list of the sizes of the cyclic factors of the abelian group being dihedralized (this will be sorted once entered)

OUTPUT:

For a given abelian group (noting that each finite abelian group can be represented as the direct product of cyclic groups), the General Dihedral Group it generates is simply the semi-direct product of the given group with C_2, where the nonidentity element of C_2 acts on the abelian group by turning each element into its inverse. In this implementation, each input abelian group will be standardized so as to act on a minimal amount of letters. This will be done by breaking the direct factors into products of p-groups, before this new set of factors is ordered from smallest to largest for complete standardization. Note that the generalized dihedral group corresponding to a cyclic group, C_n, is simply the dihedral group D_n.

EXAMPLES:

As is noted in [1], $Dih(C_3 \times C_3)$ has the presentation

$$\langle a, b, c \mid a^3, b^3, c^2, ab = ba, ac = ca^{-1}, bc = cb^{-1} \rangle$$

Note also the fact, verified by [1], that the dihedralization of $C_3 \times C_3$ is the only nonabelian group of order 18 with no element of order 6.

```
sage: G = GeneralDihedralGroup([3,3])
sage: G
Generalized dihedral group generated by C3 x C3
sage: G.order()
18
sage: G.gens()
[(4,5,6), (2,3)(5,6), (1,2,3)]
sage: a = G.gens()[2]; b = G.gens()[0]; c = G.gens()[1]
sage: a.order() == 3, b.order() == 3, c.order() == 2
(True, True, True)
sage: a*b == b*a, a*c == c*a.inverse(), b*c == c*b.inverse()
(True, True, True)
sage: G.subgroup([a,b,c]) == G
True
sage: G.is_abelian()
False
sage: all([x.order() != 6 for x in G])
True
```

If all of the direct factors are C_2, then the action turning each element into its inverse is trivial, and the semi-direct product becomes a direct product.

```
sage: G = GeneralDihedralGroup([2,2,2])
sage: G.order()
16
sage: G.gens()
[(7,8), (5,6), (3,4), (1,2)]
sage: G.is_abelian()
True
sage: H = KleinFourGroup()
sage: G.is_isomorphic(H.direct_product(H)[0])
True
```

If two nonidentical input lists generate isomorphic abelian groups, then they will generate identical groups (with each direct factor broken up into its prime factors), but they will still have distinct descriptions. Note that If $gcd(n, m) = 1$, then $C_n \times C_m \cong C_{nm}$, while the general dihedral groups generated by isomorphic abelian groups should be themselves isomorphic.

```
sage: G = GeneralDihedralGroup([6,34,46,14])
sage: H = GeneralDihedralGroup([7,17,3,46,2,2,2])
sage: G == H, G.gens() == H.gens()
(True, True)
sage: [x.order() for x in G.gens()]
[23, 17, 7, 2, 3, 2, 2, 2, 2]
sage: G
Generalized dihedral group generated by C6 x C34 x C46 x C14
sage: H
Generalized dihedral group generated by C7 x C17 x C3 x C46 x C2 x C2 x C2
```

A cyclic input yields a Classical Dihedral Group.

[1] A.D. Thomas and G.V. Wood, Group Tables (Exeter: Shiva Publishing, 1980)

```
sage: G = GeneralDihedralGroup([6])
sage: D = DihedralGroup(6)
sage: G.is_isomorphic(D)
True
```

A Generalized Dihedral Group will always have size twice the underlying group, be solvable (as it has an abelian subgroup with index 2), and, unless the underlying group is of the form ${C_2}^n$, be nonabelian (by the structure theorem of finite abelian groups and the fact that a semi-direct product is a direct product only when the underlying action is trivial).

```
sage: G = GeneralDihedralGroup([6,18,33,60])
sage: (6*18*33*60)*2
427680
sage: G.order()
427680
sage: G.is_solvable()
True
sage: G.is_abelian()
False
```

TESTS:

```
sage: G = GeneralDihedralGroup("foobar")
Traceback (most recent call last):
...
TypeError: factors of abelian group must be a list, not foobar

sage: GeneralDihedralGroup([])
Traceback (most recent call last):
...
ValueError: there must be at least one direct factor in the abelian group being dihedralized

sage: GeneralDihedralGroup([3, 1.5])
Traceback (most recent call last):
...
TypeError: the input list must consist of Integers

sage: GeneralDihedralGroup([4, -8])
Traceback (most recent call last):
...
ValueError: all direct factors must be greater than 1
```

REFERENCES:

AUTHOR:

- Kevin Halasz (2012-7-12)

class `sage.groups.perm_gps.permgroup_named.`**`JankoGroup`**(*n*)

Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_unique`

Janko Groups $J1$, $J2$, and $J3$. (Note that $J4$ is too big to be treated here.)

INPUT:

- `n` – an integer among $\{1, 2, 3\}$.

EXAMPLES:

```
sage: G = groups.permutation.Janko(1); G # optional - gap_packages internet
Janko group J1 of order 175560 as a permutation group
```

TESTS:

```
sage: G.category() # optional - gap_packages internet
Category of finite permutation groups
sage: TestSuite(G).run(skip=["_test_enumerated_set_contains", "_test_enumerated_set_iter_list"])
```

class sage.groups.perm_gps.permgroup_named.**KleinFourGroup**

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The Klein 4 Group, which has order 4 and exponent 2, viewed as a subgroup of S_4.

OUTPUT: – the Klein 4 group of order 4, as a permutation group of degree 4.

Note: This group is also available via groups.permutation.KleinFour().

EXAMPLES:

```
sage: G = KleinFourGroup(); G
The Klein 4 group of order 4, as a permutation group
sage: list(G)
[(), (3,4), (1,2), (1,2)(3,4)]
```

TESTS:

```
sage: G.category()
Category of finite permutation groups
sage: TestSuite(G).run()

sage: groups.permutation.KleinFour()
The Klein 4 group of order 4, as a permutation group
```

AUTHOR: – Bobby Moretti (2006-10)

class sage.groups.perm_gps.permgroup_named.**MathieuGroup**(*n*)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The Mathieu group of degree n.

INPUT: n – a positive integer in {9, 10, 11, 12, 21, 22, 23, 24}.

OUTPUT: – the Mathieu group of degree n, as a permutation group

Note: This group is also available via groups.permutation.Mathieu().

EXAMPLES:

```
sage: G = MathieuGroup(12)
sage: G
Mathieu group of degree 12 and order 95040 as a permutation group
```

TESTS:

```
sage: G.category()
Category of finite permutation groups
sage: TestSuite(G).run(skip=["_test_enumerated_set_contains", "_test_enumerated_set_iter_list"])

sage: groups.permutation.Mathieu(9)
Mathieu group of degree 9 and order 72 as a permutation group
```

Note: this is a fairly big group, so we skip some tests that currently require to list all the elements of the group, because there is no proper iterator yet.

class `sage.groups.perm_gps.permgroup_named.PGL(n, q, name='a')`
Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_plg`

The projective general linear groups over GF(q).

INPUT: n – positive integer; the degree q – prime power; the size of the ground field name – (default: 'a') variable name of indeterminate of finite field GF(q)

OUTPUT: PGL(n,q)

Note: This group is also available via `groups.permutation.PGL()`.

EXAMPLES:

```
sage: G = PGL(2,3); G
Permutation Group with generators [(3,4), (1,2,4)]
sage: print G
The projective general linear group of degree 2 over Finite Field of size 3
sage: G.base_ring()
Finite Field of size 3
sage: G.order()
24

sage: G = PGL(2, 9, 'b'); G
Permutation Group with generators [(3,10,9,8,4,7,6,5), (1,2,4)(5,6,8)(7,9,10)]
sage: G.base_ring()
Finite Field in b of size 3^2

sage: G.category()
Category of finite permutation groups
sage: TestSuite(G).run() # long time
```

TESTS:

```
sage: groups.permutation.PGL(2, 3)
Permutation Group with generators [(3,4), (1,2,4)]
```

class `sage.groups.perm_gps.permgroup_named.PGU(n, q, name='a')`
Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_pug`

The projective general unitary groups over GF(q).

INPUT: n – positive integer; the degree q – prime power; the size of the ground field name – (default: 'a') variable name of indeterminate of finite field GF(q)

OUTPUT: PGU(n,q)

Note: This group is also available via `groups.permutation.PGU()`.

EXAMPLES:

```
sage: PGU(2,3)
The projective general unitary group of degree 2 over Finite Field of size 3

sage: G = PGU(2, 8, name='alpha'); G
The projective general unitary group of degree 2 over Finite Field in alpha of size 2^3
sage: G.base_ring()
Finite Field in alpha of size 2^3
```

TESTS:

```
sage: groups.permutation.PGU(2, 3)
The projective general unitary group of degree 2 over Finite Field of size 3
```

class sage.groups.perm_gps.permgroup_named.**PSL**(*n*, *q*, *name='a'*)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_plg

The projective special linear groups over GF(q).

INPUT:

- n – positive integer; the degree
- q – either a prime power (the size of the ground field) or a finite field
- name – (default: 'a') variable name of indeterminate of finite field GF(q)

OUTPUT:

the group PSL(n,q)

Note: This group is also available via groups.permutation.PSL().

EXAMPLES:

```
sage: G = PSL(2,3); G
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
sage: G.order()
12
sage: G.base_ring()
Finite Field of size 3
sage: print G
The projective special linear group of degree 2 over Finite Field of size 3
```

We create two groups over nontrivial finite fields:

```
sage: G = PSL(2, 4, 'b'); G
Permutation Group with generators [(3,4,5), (1,2,3)]
sage: G.base_ring()
Finite Field in b of size 2^2
sage: G = PSL(2, 8); G
Permutation Group with generators [(3,8,6,4,9,7,5), (1,2,3)(4,7,5)(6,9,8)]
sage: G.base_ring()
Finite Field in a of size 2^3

sage: G.category()
Category of finite permutation groups
sage: TestSuite(G).run() # long time
```

TESTS:

```
sage: groups.permutation.PSL(2, 3)
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
```

Check that trac ticket #7424 is handled:

```
sage: PSL(2, GF(7,'x'))
Permutation Group with generators [(3,7,5)(4,8,6), (1,2,6)(3,4,8)]
sage: PSL(2, GF(3))
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
sage: PSL(2, QQ)
Traceback (most recent call last):
```

```
...
ValueError: q must be a prime power or a finite field
```

ramification_module_decomposition_hurwitz_curve()

Helps compute the decomposition of the ramification module for the Hurwitz curves X (over CC say) with automorphism group G = PSL(2,q), q a "Hurwitz prime" (ie, p is $\pm 1 \pmod 7$). Using this computation and Borne's formula helps determine the G-module structure of the RR spaces of equivariant divisors can be determined explicitly.

The output is a list of integer multiplicities: [m1,...,mn], where n is the number of conj classes of G=PSL(2,p) and mi is the multiplicity of pi_i in the ramification module of a Hurwitz curve with automorphism group G. Here IrrRepns(G) = [pi_1,...,pi_n] (in the order listed in the output of self.character_table()).

REFERENCE: David Joyner, Amy Ksir, Roger Vogeler, "Group representations on Riemann-Roch spaces of some Hurwitz curves," preprint, 2006.

EXAMPLES:

```
sage: G = PSL(2,13)
sage: G.ramification_module_decomposition_hurwitz_curve() # random, optional - database_gap
[0, 7, 7, 12, 12, 12, 13, 15, 14]
```

This means, for example, that the trivial representation does not occur in the ramification module of a Hurwitz curve with automorphism group PSL(2,13), since the trivial representation is listed first and that entry has multiplicity 0. The "randomness" is due to the fact that GAP randomly orders the conjugacy classes of the same order in the list of all conjugacy classes. Similarly, there is some randomness to the ordering of the characters.

If you try to use this function on a group PSL(2,q) where q is not a (smallish) "Hurwitz prime", an error message will be printed.

ramification_module_decomposition_modular_curve()

Helps compute the decomposition of the ramification module for the modular curve X(p) (over CC say) with automorphism group G = PSL(2,q), q a prime > 5. Using this computation and Borne's formula helps determine the G-module structure of the RR spaces of equivariant divisors can be determined explicitly.

The output is a list of integer multiplicities: [m1,...,mn], where n is the number of conj classes of G=PSL(2,p) and mi is the multiplicity of pi_i in the ramification module of a modular curve with automorphism group G. Here IrrRepns(G) = [pi_1,...,pi_n] (in the order listed in the output of self.character_table()).

REFERENCE: D. Joyner and A. Ksir, 'Modular representations on some Riemann-Roch spaces of modular curves $X(N)$', Computational Aspects of Algebraic Curves, (Editor: T. Shaska) Lecture Notes in Computing, WorldScientific, 2005.)

EXAMPLES:

```
sage: G = PSL(2,7)
sage: G.ramification_module_decomposition_modular_curve() # random, optional - database_gap
[0, 4, 3, 6, 7, 8]
```

This means, for example, that the trivial representation does not occur in the ramification module of X(7), since the trivial representation is listed first and that entry has multiplicity 0. The "randomness" is due to the fact that GAP randomly orders the conjugacy classes of the same order in the list of all conjugacy classes. Similarly, there is some randomness to the ordering of the characters.

sage.groups.perm_gps.permgroup_named.**PSP**

alias of PSp

class `sage.groups.perm_gps.permgroup_named.PSU(n, q, name='a')`
Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_pug`

The projective special unitary groups over GF(q).

INPUT: n – positive integer; the degree q – prime power; the size of the ground field name – (default: 'a') variable name of indeterminate of finite field GF(q)

OUTPUT: PSU(n,q)

Note: This group is also available via `groups.permutation.PSU()`.

EXAMPLES:

```
sage: PSU(2,3)
The projective special unitary group of degree 2 over Finite Field of size 3

sage: G = PSU(2, 8, name='alpha'); G
The projective special unitary group of degree 2 over Finite Field in alpha of size 2^3
sage: G.base_ring()
Finite Field in alpha of size 2^3
```

TESTS:

```
sage: groups.permutation.PSU(2, 3)
The projective special unitary group of degree 2 over Finite Field of size 3
```

class `sage.groups.perm_gps.permgroup_named.PSp(n, q, name='a')`
Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_plg`

The projective symplectic linear groups over GF(q).

INPUT: n – positive integer; the degree q – prime power; the size of the ground field name – (default: 'a') variable name of indeterminate of finite field GF(q)

OUTPUT: PSp(n,q)

Note: This group is also available via `groups.permutation.PSp()`.

EXAMPLES:

```
sage: G = PSp(2,3); G
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
sage: G.order()
12
sage: G = PSp(4,3); G
Permutation Group with generators [(3,4)(6,7)(9,10)(12,13)(17,20)(18,21)(19,22)(23,32)(24,33)(25
sage: G.order()
25920
sage: print G
The projective symplectic linear group of degree 4 over Finite Field of size 3
sage: G.base_ring()
Finite Field of size 3

sage: G = PSp(2, 8, name='alpha'); G
Permutation Group with generators [(3,8,6,4,9,7,5), (1,2,3)(4,7,5)(6,9,8)]
sage: G.base_ring()
Finite Field in alpha of size 2^3
```

TESTS:

```
sage: groups.permutation.PSp(2, 3)
Permutation Group with generators [(2,3,4), (1,2)(3,4)]
```

class `sage.groups.perm_gps.permgroup_named.`**`PermutationGroup_plg`**(*gens=None, gap_group=None, canonicalize=True, domain=None, category=None*)

Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_unique`

INPUT:

- `gens` - list of generators (default: `None`)
- `gap_group` - a gap permutation group (default: `None`)
- `canonicalize` - bool (default: `True`); if `True`, sort generators and remove duplicates

OUTPUT:

- A permutation group.

EXAMPLES:

We explicitly construct the alternating group on four elements:

```
sage: A4 = PermutationGroup([[(1,2,3)],[(2,3,4)]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.__init__([[(1,2,3)],[(2,3,4)]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.center()
Subgroup of (Permutation Group with generators [(2,3,4), (1,2,3)]) generated by [()]
sage: A4.category()
Category of finite permutation groups
sage: TestSuite(A4).run()
```

TESTS:

```
sage: TestSuite(PermutationGroup([[]])).run()
sage: TestSuite(PermutationGroup([])).run()
sage: TestSuite(PermutationGroup([(0,1)])).run()
```

`base_ring`()

EXAMPLES:

```
sage: G = PGL(2,3)
sage: G.base_ring()
Finite Field of size 3

sage: G = PSL(2,3)
sage: G.base_ring()
Finite Field of size 3
```

`matrix_degree`()

EXAMPLES:

```
sage: G = PSL(2,3)
sage: G.matrix_degree()
2
```

class `sage.groups.perm_gps.permgroup_named.`**`PermutationGroup_pug`**(*gens=None, gap_group=None, canonicalize=True, domain=None, category=None*)

Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_plg`

INPUT:

- `gens` - list of generators (default: `None`)
- `gap_group` - a gap permutation group (default: `None`)
- `canonicalize` - bool (default: `True`); if `True`, sort generators and remove duplicates

OUTPUT:

- A permutation group.

EXAMPLES:

We explicitly construct the alternating group on four elements:

```
sage: A4 = PermutationGroup([[(1,2,3)],[(2,3,4)]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.__init__([[(1,2,3)],[(2,3,4)]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.center()
Subgroup of (Permutation Group with generators [(2,3,4), (1,2,3)]) generated by [()]
sage: A4.category()
Category of finite permutation groups
sage: TestSuite(A4).run()
```

TESTS:

```
sage: TestSuite(PermutationGroup([[]])).run()
sage: TestSuite(PermutationGroup([])).run()
sage: TestSuite(PermutationGroup([(0,1)])).run()
```

`field_of_definition`()

EXAMPLES:

```
sage: PSU(2,3).field_of_definition()
Finite Field in a of size 3^2
```

class `sage.groups.perm_gps.permgroup_named.`**`PermutationGroup_symalt`**(*gens=None, gap_group=None, canonicalize=True, domain=None, category=None*)

Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_unique`

This is a class used to factor out some of the commonality in the SymmetricGroup and AlternatingGroup classes.

class sage.groups.perm_gps.permgroup_named.**PermutationGroup_unique**(*gens=None, gap_group=None, canonicalize=True, domain=None, category=None*)

Bases: sage.structure.unique_representation.CachedRepresentation, sage.groups.perm_gps.permgroup.PermutationGroup_generic

Todo

Fix the broken hash.

```
sage: G = SymmetricGroup(6)
sage: G3 = G.subgroup([G((1,2,3,4,5,6)),G((1,2))])
sage: hash(G) == hash(G3)   # todo: Should be True!
False
```

class sage.groups.perm_gps.permgroup_named.**PrimitiveGroup**(*d, n*)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The primitive group from the GAP tables of primitive groups.

INPUT:

- d – non-negative integer. the degree of the group.
- n – positive integer. the index of the group in the GAP database, starting at 1

OUTPUT:

The n-th primitive group of degree d.

EXAMPLES:

```
sage: PrimitiveGroup(0,1)
Trivial group
sage: PrimitiveGroup(1,1)
Trivial group
sage: G = PrimitiveGroup(5, 2); G            # optional - database_gap
D(2*5)
sage: G.gens()                               # optional - database_gap
[(2,4)(3,5), (1,2,3,5,4)]
sage: G.category()                           # optional - database_gap
Category of finite permutation groups
```

> **Warning:** this follows GAP's naming convention of indexing the primitive groups starting from 1:
>
> ```
> sage: PrimitiveGroup(5,0)                 # optional - database_gap
> Traceback (most recent call last):
> ...
> ValueError: Index n must be in {1,..,5}
> ```

Only primitive groups of "small" degree are available in GAP's database:

```
sage: PrimitiveGroup(2500,1)             # optional - database_gap
Traceback (most recent call last):
...
```

```
NotImplementedError: Only the primitive groups of degree less
than 2500 are available in GAP's database
```

group_primitive_id()

Return the index of this group in the GAP database of primitive groups.

Requires "optional" database_gap package.

OUTPUT:

A positive integer, following GAP's conventions.

EXAMPLES:

```
sage: G = PrimitiveGroup(5,2); G.group_primitive_id()   # optional - database_gap
2
```

sage.groups.perm_gps.permgroup_named.**PrimitiveGroups**(*d=None*)

Return the set of all primitive groups of a given degree `d`

INPUT:

- `d` – an integer (optional)

OUTPUT:

The set of all primitive groups of a given degree `d` up to isomorphisms using GAP. If `d` is not specified, it returns the set of all primitive groups up to isomorphisms stored in GAP.

> **Attention:** PrimitiveGroups requires the optional GAP database package. Please install it by running `sage -i database_gap`.

EXAMPLES:

```
sage: PrimitiveGroups(3)
Primitive Groups of degree 3
sage: PrimitiveGroups(7)
Primitive Groups of degree 7
sage: PrimitiveGroups(8)
Primitive Groups of degree 8
sage: PrimitiveGroups()
Primitive Groups
```

The database currently only contains primitive groups up to degree 2499:

```
sage: PrimitiveGroups(2500).cardinality() # optional - database_gap
Traceback (most recent call last):
...
NotImplementedError: Only the primitive groups of degree less
than 2500 are available in GAP's database
```

TODO:

This enumeration helper could be extended based on `PrimitiveGroupsIterator` in GAP. This method allows to enumerate groups with specified properties such as transitivity, solvability, ..., without creating all groups.

class sage.groups.perm_gps.permgroup_named.**PrimitiveGroupsAll**

Bases: `sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets`

The infinite set of all primitive groups up to isomorphisms.

EXAMPLES:

```
sage: L = PrimitiveGroups(); L
Primitive Groups
sage: L.category()
Category of infinite enumerated sets
sage: L.cardinality()
+Infinity

sage: p = L.__iter__()            # optional - database_gap
sage: (next(p), next(p), next(p), next(p), # optional - database_gap
...    next(p), next(p), next(p), next(p))
(Trivial group, Trivial group, S(2), A(3), S(3), A(4), S(4), C(5))
```

TESTS:

```
sage: TestSuite(PrimitiveGroups()).run() # optional - database_gap # long time
```

class `sage.groups.perm_gps.permgroup_named.`**`PrimitiveGroupsOfDegree`**(*n*)

Bases: `sage.structure.unique_representation.CachedRepresentation`, `sage.structure.parent.Parent`

The set of all primitive groups of a given degree up to isomorphisms.

EXAMPLES:

```
sage: S = PrimitiveGroups(5); S       # optional - database_gap
Primitive Groups of degree 5
sage: S.list()                          # optional - database_gap
[C(5), D(2*5), AGL(1, 5), A(5), S(5)]
sage: S.an_element() # optional - database_gap
C(5)
```

We write the cardinality of all primitive groups of degree 5:

```
sage: for G in PrimitiveGroups(5):     # optional - database_gap
...       print G.cardinality()
5
10
20
60
120
```

TESTS:

```
sage: TestSuite(PrimitiveGroups(3)).run() # optional - database_gap
```

`cardinality`()

Return the cardinality of `self`.

OUTPUT:

An integer. The number of primitive groups of a given degree up to isomorphism.

EXAMPLES:

```
sage: PrimitiveGroups(0).cardinality()                      # optional - database_gap
1
sage: PrimitiveGroups(2).cardinality()                      # optional - database_gap
1
sage: PrimitiveGroups(7).cardinality()                      # optional - database_gap
7
sage: PrimitiveGroups(12).cardinality()                     # optional - database_gap
```

```
6
sage: [PrimitiveGroups(i).cardinality() for i in range(11)] # optional - database_gap
[1, 1, 1, 2, 2, 5, 4, 7, 7, 11, 9]
```

The database_gap contains all primitive groups up to degree 2499:

```
sage: PrimitiveGroups(2500).cardinality()                    # optional - database_gap
Traceback (most recent call last):
...
NotImplementedError: Only the primitive groups of degree less than
2500 are available in GAP's database
```

TESTS:

```
sage: type(PrimitiveGroups(12).cardinality())               # optional - database_gap
<type 'sage.rings.integer.Integer'>
sage: type(PrimitiveGroups(0).cardinality())
<type 'sage.rings.integer.Integer'>
```

class `sage.groups.perm_gps.permgroup_named.`**`QuaternionGroup`**

Bases: `sage.groups.perm_gps.permgroup_named.DiCyclicGroup`

The quaternion group of order 8.

OUTPUT: The quaternion group of order 8, as a permutation group. See the `DiCyclicGroup` class for a generalization of this construction.

Note: This group is also available via `groups.permutation.Quaternion()`.

EXAMPLES:

The quaternion group is one of two non-abelian groups of order 8, the other being the dihedral group D_4. One way to describe this group is with three generators, I, J, K, so the whole group is then given as the set $\{\pm 1, \pm I, \pm J, \pm K\}$ with relations such as $I^2 = J^2 = K^2 = -1$, $IJ = K$ and $JI = -K$.

The examples below illustrate how to use this group in a similar manner, by testing some of these relations. The representation used here is the left-regular representation.

```
sage: Q = QuaternionGroup()
sage: I = Q.gen(0)
sage: J = Q.gen(1)
sage: K = I*J
sage: [I,J,K]
[(1,2,3,4)(5,6,7,8), (1,5,3,7)(2,8,4,6), (1,8,3,6)(2,7,4,5)]
sage: neg_one = I^2; neg_one
(1,3)(2,4)(5,7)(6,8)
sage: J^2 == neg_one and K^2 == neg_one
True
sage: J*I == neg_one*K
True
sage: Q.center().order() == 2
True
sage: neg_one in Q.center()
True
```

TESTS:

```
sage: groups.permutation.Quaternion()
Quaternion group of order 8 as a permutation group
```

AUTHOR: – Rob Beezer (2009-10-09)

class sage.groups.perm_gps.permgroup_named.**SemidihedralGroup**(*m*)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The semidihedral group of order 2^m.

INPUT:

- m - a positive integer; the power of 2 that is the group's order

OUTPUT:

The semidihedral group of order 2^m. These groups can be thought of as a semidirect product of $C_{2^{m-1}}$ with C_2, where the nontrivial element of C_2 is sent to the element of the automorphism group of $C_{2^{m-1}}$ that sends elements to their $-1+2^{m-2}$ th power. Thus, the group has the presentation:

$$\langle x, y \mid x^{2^{m-1}}, y^2, y^{-1}xy = x^{-1+2^{m-2}} \rangle$$

This family is notable because it is made up of non-abelian 2-groups that all contain cyclic subgroups of index 2. It is one of only four such families.

EXAMPLES:

In [GORENSTEIN1980] it is shown that the semidihedral groups have center of order 2. It is also shown that they have a Frattini subgroup equal to their commutator, which is a cyclic subgroup of order 2^{m-2}.

```
sage: G = SemidihedralGroup(12)
sage: G.order() == 2^12
True
sage: G.commutator() == G.frattini_subgroup()
True
sage: G.commutator().order() == 2^10
True
sage: G.commutator().is_cyclic()
True
sage: G.center().order()
2
```

```
sage: G = SemidihedralGroup(4)
sage: len([H for H in G.subgroups() if H.is_cyclic() and H.order() == 8])
1
sage: G.gens()
[(2,4)(3,7)(6,8), (1,2,3,4,5,6,7,8)]
sage: x = G.gens()[1]; y = G.gens()[0]
sage: x.order() == 2^3; y.order() == 2
True
True
sage: y*x*y == x^(-1+2^2)
True
```

TESTS:

```
sage: G = SemidihedralGroup(4.4)
Traceback (most recent call last):
...
TypeError: m must be an integer, not 4.40000000000000
```

```
sage: G = SemidihedralGroup(-5)
Traceback (most recent call last):
...
ValueError: the exponent must be greater than 3, not -5
```

REFERENCES:

AUTHOR:

- Kevin Halasz (2012-8-7)

class sage.groups.perm_gps.permgroup_named.**SplitMetacyclicGroup**(p, m)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The split metacyclic group of order p^m.

INPUT:

- p - a prime number that is the prime underlying this p-group
- m - a positive integer such that the order of this group is the p^m. Be aware that, for even p, m must be greater than 3, while for odd p, m must be greater than 2.

OUTPUT:

The split metacyclic group of order p^m. This family of groups has presentation

$$\langle x, y \mid x^{p^{m-1}}, y^p, y^{-1}xy = x^{1+p^{m-2}} \rangle$$

This family is notable because, for odd p, these are the only p-groups with a cyclic subgroup of index p, a result proven in [GORENSTEIN]. It is also shown in [GORENSTEIN] that this is one of four families containing nonabelian 2-groups with a cyclic subgroup of index 2 (with the others being the dicyclic groups, the dihedral groups, and the semidihedral groups).

EXAMPLES:

Using the last relation in the group's presentation, one can see that the elements of the form $y^i x$, $0 \leq i \leq p-1$ all have order p^{m-1}, as it can be shown that their p th powers are all $x^{p^{m-2}+p}$, an element with order p^{m-2}. Manipulation of the same relation shows that none of these elements are powers of any other. Thus, there are p cyclic maximal subgroups in each split metacyclic group. It is also proven in [GORENSTEIN] that this family has commutator subgroup of order p, and the Frattini subgroup is equal to the center, with this group being cyclic of order p^{m-2}. These characteristics are necessary to identify these groups in the case that $p = 2$, although the possession of a cyclic maximal subgroup in a non-abelian p-group is enough for odd p given the group's order.

```
sage: G = SplitMetacyclicGroup(2,8)
sage: G.order() == 2**8
True
sage: G.is_abelian()
False
sage: len([H for H in G.subgroups() if H.order() == 2^7 and H.is_cyclic()])
2
sage: G.commutator().order()
2
sage: G.frattini_subgroup() == G.center()
True
sage: G.center().order() == 2^6
True
sage: G.center().is_cyclic()
True

sage: G = SplitMetacyclicGroup(3,3)
sage: len([H for H in G.subgroups() if H.order() == 3^2 and H.is_cyclic()])
3
sage: G.commutator().order()
3
sage: G.frattini_subgroup() == G.center()
True
```

```
sage: G.center().order()
3
```

TESTS:

```
sage: G = SplitMetacyclicGroup(3,2.5)
Traceback (most recent call last):
...
TypeError: both p and m must be integers

sage: G = SplitMetacyclicGroup(4,3)
Traceback (most recent call last):
...
ValueError: p must be prime, 4 is not prime

sage: G = SplitMetacyclicGroup(2,2)
Traceback (most recent call last):
...
ValueError: if prime is 2, the exponent must be greater than 3, not 2

sage: G = SplitMetacyclicGroup(11,2)
Traceback (most recent call last):
...
ValueError: if prime is odd, the exponent must be greater than 2, not 2
```

REFERENCES:

AUTHOR:

- Kevin Halasz (2012-8-7)

class sage.groups.perm_gps.permgroup_named.**SuzukiGroup**(*q*, *name='a'*)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

The Suzuki group over GF(q), $^2B_2(2^{2k+1}) = Sz(2^{2k+1})$.

A wrapper for the GAP function SuzukiGroup.

INPUT:

q – 2^n, an odd power of 2; the size of the ground field. (Strictly speaking, n should be greater than 1, or else this group os not simple.)

name – (default: 'a') variable name of indeterminate of finite field GF(q)

OUTPUT:

- A Suzuki group.

Note: This group is also available via groups.permutation.Suzuki().

EXAMPLES:

```
sage: SuzukiGroup(8)
Permutation Group with generators [(1,2)(3,10)(4,42)(5,18)(6,50)(7,26)(8,58)(9,34)(12,28)(13,45)
(1,28,10,44)(3,50,11,42)(4,43,53,64)(5,9,39,52)(6,36,63,13)(7,51,60,57)(8,33,37,16)(12,24,55,29)
sage: print SuzukiGroup(8)
The Suzuki group over Finite Field in a of size 2^3

sage: G = SuzukiGroup(32, name='alpha')
sage: G.order()
32537600
```

```
sage: G.order().factor()
2^10 * 5^2 * 31 * 41
sage: G.base_ring()
Finite Field in alpha of size 2^5
```

TESTS:

```
sage: groups.permutation.Suzuki(8)
Permutation Group with generators [(1,2)(3,10)(4,42)(5,18)(6,50)(7,26)(8,58)(9,34)(12,28)(13,45)
(1,28,10,44)(3,50,11,42)(4,43,53,64)(5,9,39,52)(6,36,63,13)(7,51,60,57)(8,33,37,16)(12,24,55,29)
```

REFERENCES:

- http://en.wikipedia.org/wiki/Group_of_Lie_type#Suzuki-Ree_groups

base_ring()

EXAMPLES:

```
sage: G = SuzukiGroup(32, name='alpha')
sage: G.base_ring()
Finite Field in alpha of size 2^5
```

class sage.groups.perm_gps.permgroup_named.**SuzukiSporadicGroup**

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_unique

Suzuki Sporadic Group

EXAMPLES:

```
sage: G = groups.permutation.SuzukiSporadic(); G # optional - gap_packages internet
Sporadic Suzuki group acting on 1782 points
```

TESTS:

```
sage: G.category() # optional - gap_packages internet
Category of finite permutation groups
sage: TestSuite(G).run(skip=["_test_enumerated_set_contains", "_test_enumerated_set_iter_list"])
```

class sage.groups.perm_gps.permgroup_named.**SymmetricGroup**(*domain=None*)

Bases: sage.groups.perm_gps.permgroup_named.PermutationGroup_symalt

The full symmetric group of order $n!$, as a permutation group.

If n is a list or tuple of positive integers then it returns the symmetric group of the associated set.

INPUT:

- n – a positive integer, or list or tuple thereof

Note: This group is also available via groups.permutation.Symmetric().

EXAMPLES:

```
sage: G = SymmetricGroup(8)
sage: G.order()
40320
sage: G
Symmetric group of order 8! as a permutation group
sage: G.degree()
8
sage: S8 = SymmetricGroup(8)
sage: G = SymmetricGroup([1,2,4,5])
```

```
sage: G
Symmetric group of order 4! as a permutation group
sage: G.domain()
{1, 2, 4, 5}
sage: G = SymmetricGroup(4)
sage: G
Symmetric group of order 4! as a permutation group
sage: G.domain()
{1, 2, 3, 4}
sage: G.category()
Join of Category of finite permutation groups
 and Category of finite weyl groups
```

TESTS:

```
sage: groups.permutation.Symmetric(4)
Symmetric group of order 4! as a permutation group
```

algebra (*base_ring*, *category=None*)

Return the symmetric group algebra associated to `self`.

INPUT:

- `base_ring` – a ring
- `category` – a category (default: the category of `self`)

If `self` is the symmetric group on $1, \ldots, n$, then this is special cased to take advantage of the features in `SymmetricGroupAlgebra`. Otherwise the usual group algebra is returned.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: S4.algebra(QQ)
Symmetric group algebra of order 4 over Rational Field

sage: S3 = SymmetricGroup([1,2,3])
sage: A = S3.algebra(QQ); A
Symmetric group algebra of order 3 over Rational Field
sage: a = S3.an_element(); a
(1,2,3)
sage: A(a)
(1,2,3)
```

We illustrate the choice of the category:

```
sage: A.category()
Join of Category of coxeter group algebras over Rational Field
    and Category of finite group algebras over Rational Field
sage: A = S3.algebra(QQ, category=Semigroups())
sage: A.category()
Category of finite dimensional semigroup algebras over Rational Field
```

In the following case, a usual group algebra is returned:

sage: S = SymmetricGroup([2,3,5]) sage: S.algebra(QQ) Group algebra of Symmetric group of order 3! as a permutation group over Rational Field sage: a = S.an_element(); a (2,3,5) sage: S.algebra(QQ)(a) B[(2,3,5)]

cartan_type ()

Return the Cartan type of `self`

The symmetric group S_n is a Coxeter group of type A_{n-1}.

EXAMPLES:

```
sage: A = SymmetricGroup([2,3,7]); A.cartan_type()
['A', 2]

sage: A = SymmetricGroup([]); A.cartan_type()
['A', 0]
```

conjugacy_class(*g*)

Return the conjugacy class of `g` inside the symmetric group `self`.

INPUT:

•`g` – a partition or an element of the symmetric group `self`

OUTPUT:

A conjugacy class of a symmetric group.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: g = G((1,2,3,4))
sage: G.conjugacy_class(g)
Conjugacy class of cycle type [4, 1] in
 Symmetric group of order 5! as a permutation group
```

conjugacy_classes()

Return a list of the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes()
[Conjugacy class of cycle type [1, 1, 1, 1, 1] in
     Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [2, 1, 1, 1] in
     Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [2, 2, 1] in
     Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [3, 1, 1] in
     Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [3, 2] in
     Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [4, 1] in
     Symmetric group of order 5! as a permutation group,
 Conjugacy class of cycle type [5] in
     Symmetric group of order 5! as a permutation group]
```

conjugacy_classes_iterator()

Iterate over the conjugacy classes of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: list(G.conjugacy_classes_iterator()) == G.conjugacy_classes()
True
```

conjugacy_classes_representatives()

Return a complete list of representatives of conjugacy classes in a permutation group G.

Let S_n be the symmetric group on n letters. The conjugacy classes are indexed by partitions λ of n. The ordering of the conjugacy classes is reverse lexicographic order of the partitions.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes_representatives()
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3)(4,5),
 (1,2,3,4), (1,2,3,4,5)]

sage: S = SymmetricGroup(['a','b','c'])
sage: S.conjugacy_classes_representatives()
[(), ('a','b'), ('a','b','c')]
```

TESTS:

Check some border cases:

```
sage: S = SymmetricGroup(0)
sage: S.conjugacy_classes_representatives()
[()]
sage: S = SymmetricGroup(1)
sage: S.conjugacy_classes_representatives()
[()]
```

index_set()

Return the index set for the descents of the symmetric group `self`.

EXAMPLES:

```
sage: S8 = SymmetricGroup(8)
sage: S8.index_set()
(1, 2, 3, 4, 5, 6, 7)

sage: S = SymmetricGroup([3,1,4,5])
sage: S.index_set()
(3, 1, 4)
```

major_index(*parameter=None*)

Return the *major index generating polynomial* of `self`, which is a gadget counting the elements of `self` by major index.

INPUT:

- `parameter` – an element of a ring; the result is more explicit with a formal variable (default: element `q` of Univariate Polynomial Ring in `q` over Integer Ring)

$$P(q) = \sum_{g \in S_n} q^{\text{major index}(g)}$$

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: S4.major_index()
q^6 + 3*q^5 + 5*q^4 + 6*q^3 + 5*q^2 + 3*q + 1
sage: K.<t> = QQ[]
sage: S4.major_index(t)
t^6 + 3*t^5 + 5*t^4 + 6*t^3 + 5*t^2 + 3*t + 1
```

simple_reflection(*i*)

For i in the index set of `self`, this returns the elementary transposition $s_i = (i, i + 1)$.

EXAMPLES:

```
sage: A = SymmetricGroup(5)
sage: A.simple_reflection(3)
(3,4)

sage: A = SymmetricGroup([2,3,7])
sage: A.simple_reflections()
Finite family {2: (2,3), 3: (3,7)}
```

young_subgroup(*comp*)
Return the Young subgroup associated with the composition `comp`.

EXAMPLES:

```
sage: S = SymmetricGroup(8)
sage: c = Composition([2,2,2,2])
sage: S.young_subgroup(c)
Subgroup of (Symmetric group of order 8! as a permutation group)
 generated by [(7,8), (5,6), (3,4), (1,2)]

sage: S = SymmetricGroup(['a','b','c'])
sage: S.young_subgroup([2,1])
Subgroup of (Symmetric group of order 3! as a permutation group)
 generated by [('a','b')]

sage: Y = S.young_subgroup([2,2,2,2,2])
Traceback (most recent call last):
...
ValueError: The composition is not of expected size
```

class `sage.groups.perm_gps.permgroup_named.`**`TransitiveGroup`**(*d*, *n*)
Bases: `sage.groups.perm_gps.permgroup_named.PermutationGroup_unique`

The transitive group from the GAP tables of transitive groups.

INPUT: d – non-negative integer; the degree n – positive integer; the index of the group in the GAP database, starting at 1

OUTPUT: the n-th transitive group of degree d

Note: This group is also available via `groups.permutation.Transitive()`.

EXAMPLES:

```
sage: TransitiveGroup(0,1)
Transitive group number 1 of degree 0
sage: TransitiveGroup(1,1)
Transitive group number 1 of degree 1
sage: G = TransitiveGroup(5, 2); G         # optional - database_gap
Transitive group number 2 of degree 5
sage: G.gens()                            # optional - database_gap
[(1,2,3,4,5), (1,4)(2,3)]

sage: G.category()                        # optional - database_gap
Category of finite permutation groups
```

Warning: this follows GAP's naming convention of indexing the transitive groups starting from 1:

```
sage: TransitiveGroup(5,0)                    # optional - database_gap
Traceback (most recent call last):
...
ValueError: Index n must be in {1,..,5}
```

Warning: only transitive groups of "small" degree are available in GAP's database:

```
sage: TransitiveGroup(31,1)                   # optional - database_gap
Traceback (most recent call last):
...
NotImplementedError: Only the transitive groups of order less than 30 are available in GAP's da
```

TESTS:

```
sage: groups.permutation.Transitive(1, 1)
Transitive group number 1 of degree 1

sage: TestSuite(TransitiveGroup(0,1)).run()
sage: TestSuite(TransitiveGroup(1,1)).run()
sage: TestSuite(TransitiveGroup(5,2)).run()# optional - database_gap

sage: TransitiveGroup(1,5)                    # optional - database_gap
Traceback (most recent call last):
...
ValueError: Index n must be in {1,..,1}
```

`sage.groups.perm_gps.permgroup_named.`**`TransitiveGroups`**(*d=None*)

INPUT:

- `d` – an integer (optional)

Returns the set of all transitive groups of a given degree `d` up to isomorphisms. If `d` is not specified, it returns the set of all transitive groups up to isomorphisms.

Warning: TransitiveGroups requires the optional GAP database package. Please install it with `sage -i database_gap`.

EXAMPLES:

```
sage: TransitiveGroups(3)
Transitive Groups of degree 3
sage: TransitiveGroups(7)
Transitive Groups of degree 7
sage: TransitiveGroups(8)
Transitive Groups of degree 8

sage: TransitiveGroups()
Transitive Groups
```

Warning: in practice, the database currently only contains transitive groups up to degree 30:

```
sage: TransitiveGroups(31).cardinality() # optional - database_gap
Traceback (most recent call last):
...
NotImplementedError: Only the transitive groups of order less than 30 are available in GAP's da
```

class sage.groups.perm_gps.permgroup_named.**TransitiveGroupsAll**

Bases: sage.sets.disjoint_union_enumerated_sets.DisjointUnionEnumeratedSets

The infinite set of all transitive groups up to isomorphisms.

EXAMPLES:

```
sage: L = TransitiveGroups(); L
Transitive Groups
sage: L.category()
Category of infinite enumerated sets
sage: L.cardinality()
+Infinity

sage: p = L.__iter__()            # optional - database_gap
sage: (next(p), next(p), next(p), next(p), next(p), next(p), next(p), next(p)) # optional - data
(Transitive group number 1 of degree 0, Transitive group number 1 of degree 1, Transitive group
```

TESTS:

```
sage: TestSuite(TransitiveGroups()).run() # optional - database_gap # long time
```

class sage.groups.perm_gps.permgroup_named.**TransitiveGroupsOfDegree**(*n*)

Bases: sage.structure.unique_representation.CachedRepresentation, sage.structure.parent.Parent

The set of all transitive groups of a given (small) degree up to isomorphisms.

EXAMPLES:

```
sage: S = TransitiveGroups(4); S       # optional - database_gap
Transitive Groups of degree 4
sage: list(S)                          # optional - database_gap
[Transitive group number 1 of degree 4, Transitive group number 2 of degree 4, Transitive group

sage: TransitiveGroups(5).an_element() # optional - database_gap
Transitive group number 1 of degree 5
```

We write the cardinality of all transitive groups of degree 5:

```
sage: for G in TransitiveGroups(5):     # optional - database_gap
...       print G.cardinality()
5
10
20
60
120
```

TESTS:

```
sage: TestSuite(TransitiveGroups(3)).run() # optional - database_gap
```

cardinality()

Returns the cardinality of `self`, that is the number of transitive groups of a given degree.

EXAMPLES:

```
sage: TransitiveGroups(0).cardinality()                      # optional - database_gap
1
sage: TransitiveGroups(2).cardinality()                      # optional - database_gap
1
sage: TransitiveGroups(7).cardinality()                      # optional - database_gap
7
sage: TransitiveGroups(12).cardinality()                     # optional - database_gap
301
sage: [TransitiveGroups(i).cardinality() for i in range(11)] # optional - database_gap
[1, 1, 1, 2, 5, 5, 16, 7, 50, 34, 45]
```

Warning: The database_gap contains all transitive groups up to degree 30:

```
sage: TransitiveGroups(31).cardinality()                     # optional - database_gap
Traceback (most recent call last):
...
NotImplementedError: Only the transitive groups of order less than 30 are available in GAP'
```

TESTS:

```
sage: type(TransitiveGroups(12).cardinality())               # optional - database_gap
<type 'sage.rings.integer.Integer'>
sage: type(TransitiveGroups(0).cardinality())
<type 'sage.rings.integer.Integer'>
```

CHAPTER

TWENTYNINE

PERMUTATION GROUP ELEMENTS

AUTHORS:

- David Joyner (2006-02)
- David Joyner (2006-03): word problem method and reorganization
- Robert Bradshaw (2007-11): convert to Cython

EXAMPLES: The Rubik's cube group:

```
sage: f= [(17,19,24,22),(18,21,23,20),(6,25,43,16),(7,28,42,13),(8,30,41,11)]
sage: b=[(33,35,40,38),(34,37,39,36),( 3, 9,46,32),( 2,12,47,29),( 1,14,48,27)]
sage: l=[( 9,11,16,14),(10,13,15,12),( 1,17,41,40),( 4,20,44,37),( 6,22,46,35)]
sage: r=[(25,27,32,30),(26,29,31,28),( 3,38,43,19),( 5,36,45,21),( 8,33,48,24)]
sage: u=[( 1, 3, 8, 6),( 2, 5, 7, 4),( 9,33,25,17),(10,34,26,18),(11,35,27,19)]
sage: d=[(41,43,48,46),(42,45,47,44),(14,22,30,38),(15,23,31,39),(16,24,32,40)]
sage: cube = PermutationGroup([f,b,l,r,u,d])
sage: F=cube.gens()[0]
sage: B=cube.gens()[1]
sage: L=cube.gens()[2]
sage: R=cube.gens()[3]
sage: U=cube.gens()[4]
sage: D=cube.gens()[5]
sage: cube.order()
43252003274489856000
sage: F.order()
4
```

The interested user may wish to explore the following commands: move = cube.random_element() and time word_problem([F,B,L,R,U,D], move, False). This typically takes about 5 minutes (on a 2 Ghz machine) and outputs a word ('solving' the cube in the position move) with about 60 terms or so.

OTHER EXAMPLES: We create element of a permutation group of large degree.

```
sage: G = SymmetricGroup(30)
sage: s = G(srange(30,0,-1)); s
(1,30)(2,29)(3,28)(4,27)(5,26)(6,25)(7,24)(8,23)(9,22)(10,21)(11,20)(12,19)(13,18)(14,17)(15,16)
```

class `sage.groups.perm_gps.permgroup_element.`**`PermutationGroupElement`**

Bases: `sage.structure.element.MultiplicativeGroupElement`

An element of a permutation group.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G
Permutation Group with generators [(1,2,3)(4,5)]
```

```
sage: g = G.random_element()
sage: g in G
True
sage: g = G.gen(0); g
(1,2,3)(4,5)
sage: print g
(1,2,3)(4,5)
sage: g*g
(1,3,2)
sage: g**(-1)
(1,3,2)(4,5)
sage: g**2
(1,3,2)
sage: G = PermutationGroup([(1,2,3)])
sage: g = G.gen(0); g
(1,2,3)
sage: g.order()
3
```

This example illustrates how permutations act on multivariate polynomials.

```
sage: R = PolynomialRing(RationalField(), 5, ["x","y","z","u","v"])
sage: x, y, z, u, v = R.gens()
sage: f = x**2 - y**2 + 3*z**2
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma = G.gen(0)
sage: f * sigma
3*x^2 + y^2 - z^2
```

cycle_string(*singletons=False*)

Return string representation of this permutation.

EXAMPLES:

```
sage: g = PermutationGroupElement([(1,2,3),(4,5)])
sage: g.cycle_string()
'(1,2,3)(4,5)'

sage: g = PermutationGroupElement([3,2,1])
sage: g.cycle_string(singletons=True)
'(1,3)(2)'
```

cycle_tuples(*singletons=False*)

Return self as a list of disjoint cycles, represented as tuples rather than permutation group elements.

INPUT:

- `singletons` - boolean (default: False) whether or not consider the cycle that correspond to fixed point

EXAMPLES:

```
sage: p = PermutationGroupElement('(2,6)(4,5,1)')
sage: p.cycle_tuples()
[(1, 4, 5), (2, 6)]
sage: p.cycle_tuples(singletons=True)
[(1, 4, 5), (2, 6), (3,)]
```

EXAMPLES:

```
sage: S = SymmetricGroup(4)
sage: S.gen(0).cycle_tuples()
[(1, 2, 3, 4)]

sage: S = SymmetricGroup(['a','b','c','d'])
sage: S.gen(0).cycle_tuples()
[('a', 'b', 'c', 'd')]
sage: S([('a', 'b'), ('c', 'd')]).cycle_tuples()
[('a', 'b'), ('c', 'd')]
```

cycles()

Return self as a list of disjoint cycles.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5,6,7)'])
sage: g = G.0
sage: g.cycles()
[(1,2,3), (4,5,6,7)]
sage: a, b = g.cycles()
sage: a(1), b(1)
(2, 1)
```

dict()

Returns a dictionary associating each element of the domain with its image.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4)); g
(1,2,3,4)
sage: v = g.dict(); v
{1: 2, 2: 3, 3: 4, 4: 1}
sage: type(v[1])
<type 'int'>
sage: x = G([2,1]); x
(1,2)
sage: x.dict()
{1: 2, 2: 1, 3: 3, 4: 4}
```

domain()

Returns the domain of self.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: x = G([2,1,4,3]); x
(1,2)(3,4)
sage: v = x.domain(); v
[2, 1, 4, 3]
sage: type(v[0])
<type 'int'>
sage: x = G([2,1]); x
(1,2)
sage: x.domain()
[2, 1, 3, 4]
```

TESTS:

```
sage: S = SymmetricGroup(0)
sage: x = S.one(); x
()
sage: x.domain()
[]
```

has_descent (*i*, *side='right'*, *positive=False*)

INPUT:

- `i`: an element of the index set
- `side`: "left" or "right" (default: "right")
- `positive`: a boolean (default: False)

Returns whether `self` has a left (resp. right) descent at position `i`. If `positive` is True, then test for a non descent instead.

Beware that, since permutations are acting on the right, the meaning of descents is the reverse of the usual convention. Hence, `self` has a left descent at position `i` if `self(i) > self(i+1)`.

EXAMPLES:

```
sage: S = SymmetricGroup([1,2,3])
sage: S.one().has_descent(1)
False
sage: S.one().has_descent(2)
False
sage: s = S.simple_reflections()
sage: x = s[1]*s[2]
sage: x.has_descent(1, side = "right")
False
sage: x.has_descent(2, side = "right")
True
sage: x.has_descent(1, side = "left")
True
sage: x.has_descent(2, side = "left")
False
sage: S._test_has_descent()
```

The symmetric group acting on a set not of the form $(1, \dots, n)$ is also supported:

```
sage: S = SymmetricGroup([2,4,1])
sage: s = S.simple_reflections()
sage: x = s[2]*s[4]
sage: x.has_descent(4)
True
sage: S._test_has_descent()
```

inverse ()

Returns the inverse permutation.

OUTPUT:

For an element of a permutation group, this method returns the inverse element, which is both the inverse function and the inverse as an element of a group.

EXAMPLES:

```
sage: s = PermutationGroupElement("(1,2,3)(4,5)")
sage: s.inverse()
(1,3,2)(4,5)
```

```
sage: A = AlternatingGroup(4)
sage: t = A("(1,2,3)")
sage: t.inverse()
(1,3,2)
```

There are several ways (syntactically) to get an inverse of a permutation group element.

```
sage: s = PermutationGroupElement("(1,2,3,4)(6,7,8)")
sage: s.inverse() == s^-1
True
sage: s.inverse() == ~s
True
```

matrix()

Returns deg x deg permutation matrix associated to the permutation self

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: g.matrix()
[0 1 0 0 0]
[0 0 1 0 0]
[1 0 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
```

orbit(*n*, *sorted=True*)

Returns the orbit of the integer n under this group element, as a sorted list.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: g.orbit(4)
[4, 5]
sage: g.orbit(3)
[1, 2, 3]
sage: g.orbit(10)
[10]
```

```
sage: s = SymmetricGroup(['a', 'b']).gen(0); s
('a','b')
sage: s.orbit('a')
['a', 'b']
```

order()

Return the order of this group element, which is the smallest positive integer n for which $g^n = 1$.

EXAMPLES:

```
sage: s = PermutationGroupElement('(1,2)(3,5,6)')
sage: s.order()
6
```

TESTS:

```
sage: prod(primes(150))
1492182350939279320058875736615841068547583863326864530410
sage: L = [tuple(range(sum(primes(p))+1, sum(primes(p))+1+p)) for p in primes(150)]
```

```
sage: t=PermutationGroupElement(L).order(); t
1492182350939279320058875736615841068547583863326864530410
sage: type(t)
<type 'sage.rings.integer.Integer'>
```

sign()

Returns the sign of self, which is $(-1)^s$, where s is the number of swaps.

EXAMPLES:

```
sage: s = PermutationGroupElement('(1,2)(3,5,6)')
sage: s.sign()
-1
```

ALGORITHM: Only even cycles contribute to the sign, thus

$$sign(sigma) = (-1)^{\sum_c len(c)-1}$$

where the sum is over cycles in self.

tuple()

Return tuple of images of the domain under self.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: s = G([2,1,5,3,4])
sage: s.tuple()
(2, 1, 5, 3, 4)

sage: S = SymmetricGroup(['a', 'b'])
sage: S.gen().tuple()
('b', 'a')
```

word_problem(*words*, *display=True*)

G and H are permutation groups, g in G, H is a subgroup of G generated by a list (words) of elements of G. If g is in H, return the expression for g as a word in the elements of (words).

This function does not solve the word problem in Sage. Rather it pushes it over to GAP, which has optimized algorithms for the word problem. Essentially, this function is a wrapper for the GAP functions "EpimorphismFromFreeGroup" and "PreImagesRepresentative".

EXAMPLE:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]], canonicalize=False)
sage: g1, g2 = G.gens()
sage: h = g1^2*g2*g1
sage: h.word_problem([g1,g2], False)
('x1^2*x2^-1*x1', '(1,2,3)(4,5)^2*(3,4)^-1*(1,2,3)(4,5)')
sage: h.word_problem([g1,g2])
   x1^2*x2^-1*x1
   [['(1,2,3)(4,5)', 2], ['(3,4)', -1], ['(1,2,3)(4,5)', 1]]
('x1^2*x2^-1*x1', '(1,2,3)(4,5)^2*(3,4)^-1*(1,2,3)(4,5)')
```

sage.groups.perm_gps.permgroup_element.**is_PermutationGroupElement**(*x*)

Returns True if x is a PermutationGroupElement.

EXAMPLES:

```
sage: p = PermutationGroupElement([(1,2),(3,4,5)])
sage: from sage.groups.perm_gps.permgroup_element import is_PermutationGroupElement
```

```
sage: is_PermutationGroupElement(p)
True
```

sage.groups.perm_gps.permgroup_element.**make_permgroup_element**(*G, x*)

Returns a PermutationGroupElement given the permutation group G and the permutation x in list notation.

This is function is used when unpickling old (pre-domain) versions of permutation groups and their elements. This now does a bit of processing and calls `make_permgroup_element_v2()` which is used in unpickling the current PermutationGroupElements.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_element import make_permgroup_element
sage: S = SymmetricGroup(3)
sage: make_permgroup_element(S, [1,3,2])
(2,3)
```

sage.groups.perm_gps.permgroup_element.**make_permgroup_element_v2**(*G, x, domain*)

Returns a PermutationGroupElement given the permutation group G, the permutation x in list notation, and the domain `domain` of the permutation group.

This is function is used when unpickling permutation groups and their elements.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_element import make_permgroup_element_v2
sage: S = SymmetricGroup(3)
sage: make_permgroup_element_v2(S, [1,3,2], S.domain())
(2,3)
```

sage.groups.perm_gps.permgroup_element.**standardize_generator**(*g, convert_dict=None*)

Standardizes the input for permutation group elements to a list of tuples. This was factored out of the PermutationGroupElement.__init__ since PermutationGroup_generic.__init__ needs to do the same computation in order to compute the domain of a group when it's not explicitly specified.

INPUT:

- `g` - a list, tuple, string, GapElement, PermutationGroupElement, Permutation
- `convert_dict` - (optional) a dictionary used to convert the points to a number compatible with GAP.

OUTPUT:

The permutation in as a list of cycles.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_element import standardize_generator
sage: standardize_generator('(1,2)')
[(1, 2)]

sage: p = PermutationGroupElement([(1,2)])
sage: standardize_generator(p)
[(1, 2)]
sage: standardize_generator(p._gap_())
[(1, 2)]
sage: standardize_generator((1,2))
[(1, 2)]
sage: standardize_generator([(1,2)])
[(1, 2)]
```

```
sage: standardize_generator(Permutation([2,1,3]))
[(1, 2), (3,)]

sage: d = {'a': 1, 'b': 2}
sage: p = SymmetricGroup(['a', 'b']).gen(0); p
('a','b')
sage: standardize_generator(p, convert_dict=d)
[(1, 2)]
sage: standardize_generator(p._gap_(), convert_dict=d)
[(1, 2)]
sage: standardize_generator(('a','b'), convert_dict=d)
[(1, 2)]
sage: standardize_generator([('a','b')], convert_dict=d)
[(1, 2)]
```

sage.groups.perm_gps.permgroup_element.**string_to_tuples**(*g*)

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_element import string_to_tuples
sage: string_to_tuples('(1,2,3)')
[(1, 2, 3)]
sage: string_to_tuples('(1,2,3)(4,5)')
[(1, 2, 3), (4, 5)]
sage: string_to_tuples(' (1,2, 3) (4,5)')
[(1, 2, 3), (4, 5)]
sage: string_to_tuples('(1,2)(3)')
[(1, 2), (3,)]
```

CHAPTER

THIRTY

PERMUTATION GROUP HOMOMORPHISMS

AUTHORS:

- David Joyner (2006-03-21): first version
- David Joyner (2008-06): fixed kernel and image to return a group, instead of a string.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
sage: phi.image(G)
Subgroup of (Dihedral group of order 8 as a permutation group) generated by [(1,2,3,4)]
sage: phi.kernel()
Subgroup of (Cyclic group of order 4 as a permutation group) generated by [()]
sage: phi.image(g)
(1,2,3,4)
sage: phi(g)
(1,2,3,4)
sage: phi.codomain()
Dihedral group of order 8 as a permutation group
sage: phi.codomain()
Dihedral group of order 8 as a permutation group
sage: phi.domain()
Cyclic group of order 4 as a permutation group
```

class `sage.groups.perm_gps.permgroup_morphism.`**`PermutationGroupMorphism`**

Bases: `sage.categories.morphism.Morphism`

A set-theoretic map between PermutationGroups.

`image`(*J*)

J must be a subgroup of G. Computes the subgroup of H which is the image of J.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
sage: phi.image(G)
Subgroup of (Dihedral group of order 8 as a permutation group) generated by [(1,2,3,4)]
sage: phi.image(g)
(1,2,3,4)
```

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
sage: H = D[0]
sage: pr1 = D[3]
sage: pr1.image(G)
Subgroup of (The projective special linear group of degree 2 over Finite Field of size 7) ge
sage: G.is_isomorphic(pr1.image(G))
True
```

kernel()

Returns the kernel of this homomorphism as a permutation group.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens(G, H, [1])
sage: phi.kernel()
Subgroup of (Cyclic group of order 4 as a permutation group) generated by [(1,2,3,4)]
```

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
sage: H = D[0]
sage: pr1 = D[3]
sage: G.is_isomorphic(pr1.kernel())
True
```

class sage.groups.perm_gps.permgroup_morphism.**PermutationGroupMorphism_from_gap**(*G*, *H*, *gap_hom*)

Bases: sage.groups.perm_gps.permgroup_morphism.PermutationGroupMorphism

This is a Python trick to allow Sage programmers to create a group homomorphism using GAP using very general constructions. An example of its usage is in the direct_product instance method of the PermutationGroup_generic class in permgroup.py.

Basic syntax:

PermutationGroupMorphism_from_gap(domain_group, range_group,'phi:=gap_hom_command;','phi') And don't forget the line: from sage.groups.perm_gps.permgroup_morphism import PermutationGroupMorphism_from_gap in your program.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_morphism import PermutationGroupMorphism_from_gap
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: PermutationGroupMorphism_from_gap(H, G, gap.Identity)
Permutation group morphism:
  From: Subgroup of (Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]) generated by [(1
  To:   Permutation Group with generators [(1,2)(3,4), (1,2,3,4)]
  Defn: Identity
```

class sage.groups.perm_gps.permgroup_morphism.**PermutationGroupMorphism_id**

Bases: sage.groups.perm_gps.permgroup_morphism.PermutationGroupMorphism

class sage.groups.perm_gps.permgroup_morphism.**PermutationGroupMorphism_im_gens**(*G*, *H*, *gens=None*)

Bases: `sage.groups.perm_gps.permgroup_morphism.PermutationGroupMorphism`

Some python code for wrapping GAP's GroupHomomorphismByImages function but only for permutation groups. Can be expensive if G is large. Returns "fail" if gens does not generate self or if the map does not extend to a group homomorphism, self - other.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens())); phi
Permutation group morphism:
  From: Cyclic group of order 4 as a permutation group
  To:   Dihedral group of order 8 as a permutation group
  Defn: [(1,2,3,4)] -> [(1,2,3,4)]
sage: g = G([(1,3),(2,4)]); g
(1,3)(2,4)
sage: phi(g)
(1,3)(2,4)
sage: images = ((4,3,2,1),)
sage: phi = PermutationGroupMorphism_im_gens(G, G, images)
sage: g = G([(1,2,3,4)]); g
(1,2,3,4)
sage: phi(g)
(1,4,3,2)
```

AUTHORS:

- David Joyner (2006-02)

`sage.groups.perm_gps.permgroup_morphism.`**`is_PermutationGroupMorphism`**`(`*f*`)`

Returns True if the argument `f` is a PermutationGroupMorphism.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_morphism import is_PermutationGroupMorphism
sage: G = CyclicPermutationGroup(4)
sage: H = DihedralGroup(4)
sage: phi = PermutationGroupMorphism_im_gens(G, H, map(H, G.gens()))
sage: is_PermutationGroupMorphism(phi)
True
```

CHAPTER THIRTYONE

RUBIK'S CUBE GROUP FUNCTIONS

Note: "Rubiks cube" is trademarked. We shall omit the trademark symbol below for simplicity.

NOTATION:

B denotes a clockwise quarter turn of the back face, D denotes a clockwise quarter turn of the down face, and similarly for F (front), L (left), R (right), and U (up). Products of moves are read right to left, so for example, $R \cdot U$ means move U first and then R.

See `CubeGroup.parse()` for all possible input notations.

The "Singmaster notation":

- moves: U, D, R, L, F, B as in the diagram below,
- corners: xyz means the facet is on face x (in R, F, L, U, D, B) and the clockwise rotation of the corner sends $x - y - z$
- edges: xy means the facet is on face x and a flip of the edge sends $x - y$.

```
sage: rubik = CubeGroup()
sage: rubik.display2d("")
             +--------------+
             |  1    2    3 |
             |  4   top   5 |
             |  6    7    8 |
+------------+--------------+-------------+------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+------------+--------------+-------------+------------+
             | 41   42   43 |
             | 44 bottom 45 |
             | 46   47   48 |
             +--------------+
```

AUTHORS:

- David Joyner (2006-10-21): first version
- David Joyner (2007-05): changed faces, added legal and solve
- David Joyner(2007-06): added plotting functions
- David Joyner (2007, 2008): colors corrected, "solve" rewritten (again),typos fixed.
- Robert Miller (2007, 2008): editing, cleaned up display2d
- Robert Bradshaw (2007, 2008): RubiksCube object, 3d plotting.

- David Joyner (2007-09): rewrote docstring for CubeGroup's "solve".
- Robert Bradshaw (2007-09): Versatile parse function for all input types.
- Robert Bradshaw (2007-11): Cleanup.

REFERENCES:

- Cameron, P., Permutation Groups. New York: Cambridge University Press, 1999.
- Wielandt, H., Finite Permutation Groups. New York: Academic Press, 1964.
- Dixon, J. and Mortimer, B., Permutation Groups, Springer-Verlag, Berlin/New York, 1996.
- Joyner,D., Adventures in Group Theory, Johns Hopkins Univ Press, 2002.

class `sage.groups.perm_gps.cubegroup.`**`CubeGroup`**

Bases: `sage.groups.perm_gps.permgroup.PermutationGroup_generic`

A python class to help compute Rubik's cube group actions.

Note: This group is also available via `groups.permutation.RubiksCube()`.

EXAMPLES:

If G denotes the cube group then it may be regarded as a subgroup of `SymmetricGroup(48)`, where the 48 facets are labeled as follows.

```
sage: rubik = CubeGroup()
sage: rubik.display2d("")
             +--------------+
             |  1    2    3 |
             |  4   top   5 |
             |  6    7    8 |
+------------+--------------+-------------+------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+------------+--------------+-------------+------------+
             | 41   42   43 |
             | 44 bottom 45 |
             | 46   47   48 |
             +--------------+

sage: rubik
The Rubik's cube group with generators R,L,F,B,U,D in SymmetricGroup(48).

TESTS::

    sage: groups.permutation.RubiksCube()
    The Rubik's cube group with generators R,L,F,B,U,D in SymmetricGroup(48).
```

`B``()`

Return the generator B in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.B()
(1,14,48,27)(2,12,47,29)(3,9,46,32)(33,35,40,38)(34,37,39,36)
```

`D``()`

Return the generator D in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.D()
(14,22,30,38)(15,23,31,39)(16,24,32,40)(41,43,48,46)(42,45,47,44)
```

F()

Return the generator F in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.F()
(6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,22)(18,21,23,20)
```

L()

Return the generator L in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.L()
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
```

R()

Return the generator R in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.R()
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
```

U()

Return the generator U in Singmaster notation.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.U()
(1,3,8,6)(2,5,7,4)(9,33,25,17)(10,34,26,18)(11,35,27,19)
```

display2d(*mv*)

Print the 2d representation of `self`.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.display2d("R")
             +--------------+
             |  1    2   38 |
             |  4   top  36 |
             |  6    7   33 |
+------------+--------------+-------------+------------+
|  9  10  11 | 17   18    3 | 27   29  32 | 48  34  35 |
| 12 left 13 | 20  front  5 | 26 right 31 | 45 rear 37 |
| 14  15  16 | 22   23    8 | 25   28  30 | 43  39  40 |
+------------+--------------+-------------+------------+
             | 41   42   19 |
             | 44 bottom 21 |
             | 46   47   24 |
             +--------------+
```

faces (*mv*)

Return the dictionary of faces created by the effect of the move mv, which is a string of the form $X^a*Y^b*...$, where $X, Y, \ldots$ are in $\{R, L, F, B, U, D\}$ and $a, b, \ldots$ are integers. We call this ordering of the faces the "BDFLRU, L2R, T2B ordering".

EXAMPLES:

```
sage: rubik = CubeGroup()
```

Here is the dictionary of the solved state:

```
sage: sorted(rubik.faces("").items())
[('back', [[33, 34, 35], [36, 0, 37], [38, 39, 40]]),
 ('down', [[41, 42, 43], [44, 0, 45], [46, 47, 48]]),
 ('front', [[17, 18, 19], [20, 0, 21], [22, 23, 24]]),
 ('left', [[9, 10, 11], [12, 0, 13], [14, 15, 16]]),
 ('right', [[25, 26, 27], [28, 0, 29], [30, 31, 32]]),
 ('up', [[1, 2, 3], [4, 0, 5], [6, 7, 8]])]
```

Now the dictionary of the state obtained after making the move R followed by L:

```
sage: sorted(rubik.faces("R*U").items())
[('back', [[48, 26, 27], [45, 0, 37], [43, 39, 40]]),
 ('down', [[41, 42, 11], [44, 0, 21], [46, 47, 24]]),
 ('front', [[9, 10, 8], [20, 0, 7], [22, 23, 6]]),
 ('left', [[33, 34, 35], [12, 0, 13], [14, 15, 16]]),
 ('right', [[19, 29, 32], [18, 0, 31], [17, 28, 30]]),
 ('up', [[3, 5, 38], [2, 0, 36], [1, 4, 25]])]
```

facets (*g=None*)

Return the set of facets on which the group acts. This function is a "constant".

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.facets() == range(1,49)
True
```

gen_names ()

Return the names of the generators.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.gen_names()
['B', 'D', 'F', 'L', 'R', 'U']
```

legal (*state, mode='quiet'*)

Return 1 (true) if the dictionary `state` (in the same format as returned by the faces method) represents a legal position (or state) of the Rubik's cube or 0 (false) otherwise.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: r0 = rubik.faces("")
sage: r1 = {'back': [[33, 34, 35], [36, 0, 37], [38, 39, 40]], 'down': [[41, 42, 43], [44, 0
sage: rubik.legal(r0)
1
sage: rubik.legal(r0,"verbose")
(1, ())
sage: rubik.legal(r1)
0
```

move (*mv*)

Return the group element and the reordered list of facets, as moved by the list `mv` (read left-to-right)

INPUT:

- `mv` – A string of the form `Xa*Yb*...`, where `X`, `Y`, ... are in `R`, `L`, `F`, `B`, `U`, `D` and `a`, `b`, ... are integers.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.move("")[0]
()
sage: rubik.move("R")[0]
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
sage: rubik.R()
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
```

parse (*mv*, *check=True*)

This function allows one to create the permutation group element from a variety of formats.

INPUT:

- `mv` – Can one of the following:
  - `list` - list of facets (as returned by self.facets())
  - `dict` - list of faces (as returned by `self.faces()`)
  - `str` - either cycle notation (passed to GAP) or a product of generators or Singmaster notation
  - `perm_group element` - returned as an element of `self`
- `check` – check if the input is valid

EXAMPLES:

```
sage: C = CubeGroup()
sage: C.parse(range(1,49))
()
sage: g = C.parse("L"); g
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
sage: C.parse(str(g)) == g
True
sage: facets = C.facets(g); facets
[17, 2, 3, 20, 5, 22, 7, 8, 11, 13, 16, 10, 15, 9, 12, 14, 41, 18, 19, 44, 21, 46, 23, 24, 2
sage: C.parse(facets)
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
sage: C.parse(facets) == g
True
sage: faces = C.faces("L"); faces
{'back': [[33, 34, 6], [36, 0, 4], [38, 39, 1]],
 'down': [[40, 42, 43], [37, 0, 45], [35, 47, 48]],
 'front': [[41, 18, 19], [44, 0, 21], [46, 23, 24]],
 'left': [[11, 13, 16], [10, 0, 15], [9, 12, 14]],
 'right': [[25, 26, 27], [28, 0, 29], [30, 31, 32]],
 'up': [[17, 2, 3], [20, 0, 5], [22, 7, 8]]}
sage: C.parse(faces) == C.parse("L")
True
sage: C.parse("L' R2") == C.parse("L^(-1)*R^2")
True
sage: C.parse("L' R2")
(1,40,41,17)(3,43)(4,37,44,20)(5,45)(6,35,46,22)(8,48)(9,14,16,11)(10,12,15,13)(19,38)(21,36
sage: C.parse("L^4")
()
```

```
sage: C.parse("L^(-1)*R")
(1,40,41,17)(3,38,43,19)(4,37,44,20)(5,36,45,21)(6,35,46,22)(8,33,48,24)(9,14,16,11)(10,12,1
```

plot3d_cube(*mv*, *title=True*)

Displays F, U, R faces of the cube after the given move `mv`. Mostly included for the purpose of drawing pictures and checking moves.

INPUT:

- `mv` – A string in the Singmaster notation
- `title` – (Default: `True`) Display the title information

The first one below is "superflip+4 spot" (in 26q* moves) and the second one is the superflip (in 20f* moves). Type show(P) to view them.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: P = rubik.plot3d_cube("U^2*F*U^2*L*R^(-1)*F^2*U*F^3*B^3*R*L*U^2*R*D^3*U*L^3*R*D*R^3*L^
sage: P = rubik.plot3d_cube("R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*F^3*B^3*D^3*F^2*D^3*R^2*U
```

plot_cube(*mv*, *title=True*, *colors=[(1, 0.63, 1), (1, 1, 0), (1, 0, 0), (0, 1, 0), (1, 0.6, 0.3), (0, 0, 1)]*)

Input the move mv, as a string in the Singmaster notation, and output the 2D plot of the cube in that state.

Type `P.show()` to display any of the plots below.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: P = rubik.plot_cube("R^2*U^2*R^2*U^2*R^2*U^2", title = False)
sage: # (R^2U^2)^3  permutes 2 pairs of edges (uf,ub)(fr,br)
sage: P = rubik.plot_cube("R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*F^3*B^3*D^3*F^2*D^3*R^2*U^3
sage: # the superflip (in 20f* moves)
sage: P = rubik.plot_cube("U^2*F*U^2*L*R^(-1)*F^2*U*F^3*B^3*R*L*U^2*R*D^3*U*L^3*R*D*R^3*L^3*
sage: # "superflip+4 spot" (in 26q* moves)
```

repr2d(*mv*)

Displays a 2D map of the Rubik's cube after the move mv has been made. Nothing is returned.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: print rubik.repr2d("")
             +--------------+
             |  1    2    3 |
             |  4   top   5 |
             |  6    7    8 |
+------------+--------------+-------------+------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+------------+--------------+-------------+------------+
             | 41   42   43 |
             | 44 bottom 45 |
             | 46   47   48 |
             +--------------+

sage: print rubik.repr2d("R")
             +--------------+
             |  1    2   38 |
             |  4   top  36 |
```

```
              |  6    7   33 |
+------------+--------------+-------------+------------+
|  9  10  11 | 17   18    3 | 27   29  32 | 48  34  35 |
| 12 left 13 | 20  front  5 | 26 right 31 | 45 rear 37 |
| 14  15  16 | 22   23    8 | 25   28  30 | 43  39  40 |
+------------+--------------+-------------+------------+
              | 41   42   19 |
              | 44 bottom 21 |
              | 46   47   24 |
              +--------------+
```

You can see the right face has been rotated but not the left face.

solve (*state*, *algorithm='default'*)

Solves the cube in the `state`, given as a dictionary as in `legal`. See the `solve` method of the RubiksCube class for more details.

This may use GAP's `EpimorphismFromFreeGroup` and `PreImagesRepresentative` as explained below, if 'gap' is passed in as the algorithm.

This algorithm

1. constructs the free group on 6 generators then computes a reasonable set of relations which they satisfy
2. computes a homomorphism from the cube group to this free group quotient
3. takes the cube position, regarded as a group element, and maps it over to the free group quotient
4. using those relations and tricks from combinatorial group theory (stabilizer chains), solves the "word problem" for that element.
5. uses python string parsing to rewrite that in cube notation.

The Rubik's cube group has about 4.3×10^{19} elements, so this process is time-consuming. See http://www.gap-system.org/Doc/Examples/rubik.html for an interesting discussion of some GAP code analyzing the Rubik's cube.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: state = rubik.faces("R")
sage: rubik.solve(state)
'R'
sage: state = rubik.faces("R*U")
sage: rubik.solve(state, algorithm='gap')       # long time
'R*U'
```

You can also check this another (but similar) way using the `word_problem` method (eg, G = rubik.group(); g = G("(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)"); g.word_problem([b,d,f,l,r,u]), though the output will be less intuitive).

class `sage.groups.perm_gps.cubegroup.`**RubiksCube** (*state=None*, *history=[]*, *colors=[(1, 0.63, 1), (1, 1, 0), (1, 0, 0), (0, 1, 0), (1, 0.6, 0.3), (0, 0, 1)]*)

Bases: `sage.structure.sage_object.SageObject`

The Rubik's cube (in a given state).

EXAMPLES:

```
sage: C = RubiksCube().move("R U R'")
sage: C.show3d()
```

```
sage: C = RubiksCube("R*L"); C
             +--------------+
             | 17    2   38 |
             | 20   top  36 |
             | 22    7   33 |
+------------+--------------+-------------+------------+
| 11   13  16 | 41   18     3 | 27   29  32 | 48  34    6 |
| 10 left 15 | 44  front   5 | 26 right 31 | 45 rear   4 |
|  9   12  14 | 46   23     8 | 25   28  30 | 43  39    1 |
+------------+--------------+-------------+------------+
             | 40   42   19 |
             | 37 bottom 21 |
             | 35   47   24 |
             +--------------+
sage: C.show()
sage: C.solve(algorithm='gap')  # long time
'L R'
sage: C == RubiksCube("L*R")
True
```

cubie (*size, gap, x, y, z, colors, stickers=True*)

Return the cubie at (x, y, z).

INPUT:

- `size` – The size of the cubie
- `gap` – The gap between cubies
- `x, y, z` – The position of the cubie
- `colors` – The list of colors
- `stickers` – (Default `True`) Boolean to display stickers

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.cubie(0.15, 0.025, 0,0,0, C.colors*3)
Graphics3d Object
```

facets ()

Return the facets of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.facets()
[3, 5, 38, 2, 36, 1, 4, 25, 33, 34, 35, 12, 13, 14, 15, 16, 9, 10,
 8, 20, 7, 22, 23, 6, 19, 29, 32, 18, 31, 17, 28, 30, 48, 26, 27,
 45, 37, 43, 39, 40, 41, 42, 11, 44, 21, 46, 47, 24]
```

move (*g*)

Move the Rubik's cube by `g`.

EXAMPLES:

```
sage: RubiksCube().move("R*U") == RubiksCube("R*U")
True
```

plot ()

Return a plot of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.plot()
Graphics object consisting of 55 graphics primitives
```

plot3d(*stickers=True*)

Return a 3D plot of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.plot3d()
Graphics3d Object
```

scramble(*moves=30*)

Scramble the Rubik's cube.

EXAMPLES:

```
sage: C = RubiksCube()
sage: C.scramble() # random
                         +--------------+
                         | 38   29   35 |
                         | 20   top  42 |
                         | 11   44   30 |
            +------------+--------------+-------------+------------+
            | 48  13  17 |  6   15   24 | 43   23   9 |  1  36  32 |
            |  4 left 18 |  7  front 37 | 12 right 26 |  5 rear 10 |
            | 33  31  40 | 14   28    8 | 25   47  16 | 22   2   3 |
            +------------+--------------+-------------+------------+
                         | 46   21   19 |
                         | 45 bottom 39 |
                         | 27   34   41 |
                         +--------------+
```

show()

Show a plot of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.show()
```

show3d()

Show a 3D plot of `self`.

EXAMPLES:

```
sage: C = RubiksCube("R*U")
sage: C.show3d()
```

solve(*algorithm='hybrid'*, *timeout=15*)

Solve the Rubik's cube.

INPUT:

- `algorithm` – must be one of the following:
  - `hybrid` - try `kociemba` for timeout seconds, then `dietz`
  - `kociemba` - Use Dik T. Winter's program (reasonable speed, few moves)

–`dietz` - Use Eric Dietz's cubex program (fast but lots of moves)

–`optimal` - Use Michael Reid's optimal program (may take a long time)

–`gap` - Use GAP word solution (can be slow)

EXAMPLES:

```
sage: C = RubiksCube("R U F L B D")
sage: C.solve()
'R U F L B D'
```

Dietz's program is much faster, but may give highly non-optimal solutions:

```
sage: s = C.solve('dietz'); s
"U' L' L' U L U' L U D L L D' L' D L' D' L D L' U' L D' L' U L' B' U' L' U B L D L D' U' L'
sage: C2 = RubiksCube(s)
sage: C == C2
True
```

undo()

Undo the last move of the Rubik's cube.

EXAMPLES:

```
sage: C = RubiksCube()
sage: D = C.move("R*U")
sage: D.undo() == C
True
```

`sage.groups.perm_gps.cubegroup.`**color_of_square**(*facet, colors=['lpurple', 'yellow', 'red', 'green', 'orange', 'blue']*)

Return the color the facet has in the solved state.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import *
sage: color_of_square(41)
'blue'
```

`sage.groups.perm_gps.cubegroup.`**create_poly**(*face, color*)

Create the polygon given by `face` with color `color`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import create_poly, red
sage: create_poly('ur', red)
Graphics object consisting of 1 graphics primitive
```

`sage.groups.perm_gps.cubegroup.`**cubie_centers**(*label*)

Return the cubie center list element given by `label`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import cubie_centers
sage: cubie_centers(3)
[0, 2, 2]
```

`sage.groups.perm_gps.cubegroup.`**cubie_colors**(*label, state0*)

Return the color of the cubie given by `label` at `state0`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import cubie_colors
sage: G = CubeGroup()
sage: g = G.parse("R*U")
sage: cubie_colors(3, G.facets(g))
[(1, 1, 1), (1, 0.63, 1), (1, 0.6, 0.3)]
```

sage.groups.perm_gps.cubegroup.**cubie_faces**()

This provides a map from the 6 faces of the 27 cubies to the 48 facets of the larger cube.

-1,-1,-1 is left, top, front

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import cubie_faces
sage: sorted(cubie_faces().items())
[((-1, -1, -1), [6, 17, 11, 0, 0, 0]),
 ((-1, -1, 0), [4, 0, 10, 0, 0, 0]),
 ((-1, -1, 1), [1, 0, 9, 0, 35, 0]),
 ((-1, 0, -1), [0, 20, 13, 0, 0, 0]),
 ((-1, 0, 0), [0, 0, -5, 0, 0, 0]),
 ((-1, 0, 1), [0, 0, 12, 0, 37, 0]),
 ((-1, 1, -1), [0, 22, 16, 41, 0, 0]),
 ((-1, 1, 0), [0, 0, 15, 44, 0, 0]),
 ((-1, 1, 1), [0, 0, 14, 46, 40, 0]),
 ((0, -1, -1), [7, 18, 0, 0, 0, 0]),
 ((0, -1, 0), [-6, 0, 0, 0, 0, 0]),
 ((0, -1, 1), [2, 0, 0, 0, 34, 0]),
 ((0, 0, -1), [0, -4, 0, 0, 0, 0]),
 ((0, 0, 0), [0, 0, 0, 0, 0, 0]),
 ((0, 0, 1), [0, 0, 0, 0, -2, 0]),
 ((0, 1, -1), [0, 23, 0, 42, 0, 0]),
 ((0, 1, 0), [0, 0, 0, -1, 0, 0]),
 ((0, 1, 1), [0, 0, 0, 47, 39, 0]),
 ((1, -1, -1), [8, 19, 0, 0, 0, 25]),
 ((1, -1, 0), [5, 0, 0, 0, 0, 26]),
 ((1, -1, 1), [3, 0, 0, 0, 33, 27]),
 ((1, 0, -1), [0, 21, 0, 0, 0, 28]),
 ((1, 0, 0), [0, 0, 0, 0, 0, -3]),
 ((1, 0, 1), [0, 0, 0, 0, 36, 29]),
 ((1, 1, -1), [0, 24, 0, 43, 0, 30]),
 ((1, 1, 0), [0, 0, 0, 45, 0, 31]),
 ((1, 1, 1), [0, 0, 0, 48, 38, 32])]
```

sage.groups.perm_gps.cubegroup.**index2singmaster**(*facet*)

Translate index used (eg, 43) to Singmaster facet notation (eg, fdr).

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import *
sage: index2singmaster(41)
'dlf'
```

sage.groups.perm_gps.cubegroup.**inv_list**(*lst*)

Input a list of ints $1, \ldots, m$ (in any order), outputs inverse perm.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import inv_list
sage: L = [2,3,1]
sage: inv_list(L)
```

```
[3, 1, 2]
```

sage.groups.perm_gps.cubegroup.**plot3d_cubie**(*cnt*, *clrs*)

Plot the front, up and right face of a cubie centered at cnt and rgbcolors given by clrs (in the order FUR).

Type `P.show()` to view.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import *
sage: clrF = blue; clrU = red; clrR = green
sage: P = plot3d_cubie([1/2,1/2,1/2],[clrF,clrU,clrR])
```

sage.groups.perm_gps.cubegroup.**polygon_plot3d**(*points*, *tilt=30*, *turn=30*, ***kwargs*)

Plot a polygon viewed from an angle determined by `tilt`, `turn`, and vertices `points`.

> **Warning:** The ordering of the points is important to get "correct" and if you add several of these plots together, the one added first is also drawn first (ie, addition of Graphics objects is not commutative).

The following example produced a green-colored square with vertices at the points indicated.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import polygon_plot3d,green
sage: P = polygon_plot3d([[1,3,1],[2,3,1],[2,3,2],[1,3,2],[1,3,1]],rgbcolor=green)
```

sage.groups.perm_gps.cubegroup.**rotation_list**(*tilt*, *turn*)

Return a list $[\sin(\theta), \sin(\phi), \cos(\theta), \cos(\phi)]$ of rotations where θ is `tilt` and ϕ is `turn`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import rotation_list
sage: rotation_list(30, 45)
[0.49999999999999994, 0.7071067811865475, 0.8660254037844387, 0.7071067811865476]
```

sage.groups.perm_gps.cubegroup.**xproj**(*x*, *y*, *z*, *r*)

Return the x-projection of (x, y, z) rotated by r.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import rotation_list, xproj
sage: rot = rotation_list(30, 45)
sage: xproj(1,2,3,rot)
0.6123724356957945
```

sage.groups.perm_gps.cubegroup.**yproj**(*x*, *y*, *z*, *r*)

Return the y-projection of (x, y, z) rotated by r.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import rotation_list, yproj
sage: rot = rotation_list(30, 45)
sage: yproj(1,2,3,rot)
1.378497416975604
```

CHAPTER

THIRTYTWO

CONJUGACY CLASSES OF THE SYMMETRIC GROUP

AUTHORS:

- Vincent Delacroix, Travis Scrimshaw (2014-11-23)

class sage.groups.perm_gps.symgp_conjugacy_class.**PermutationsConjugacyClass**(*P*, *part*)

Bases: sage.groups.perm_gps.symgp_conjugacy_class.SymmetricGroupConjugacyClassMixin, sage.groups.conjugacy_classes.ConjugacyClass

A conjugacy class of the permutations of n.

INPUT:

- P – the permutations of n
- part – a partition or an element of P

set()

The set of all elements in the conjugacy class self.

EXAMPLES:

```
sage: G = Permutations(3)
sage: g = G([2, 1, 3])
sage: C = G.conjugacy_class(g)
sage: S = [[1, 3, 2], [2, 1, 3], [3, 2, 1]]
sage: C.set() == Set(G(x) for x in S)
True
```

class sage.groups.perm_gps.symgp_conjugacy_class.**SymmetricGroupConjugacyClass**(*group*, *part*)

Bases: sage.groups.perm_gps.symgp_conjugacy_class.SymmetricGroupConjugacyClassMixin, sage.groups.conjugacy_classes.ConjugacyClassGAP

A conjugacy class of the symmetric group.

INPUT:

- group – the symmetric group
- part – a partition or an element of group

set()

The set of all elements in the conjugacy class self.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2))
sage: C = G.conjugacy_class(g)
```

```
sage: S = [(2,3), (1,2), (1,3)]
sage: C.set() == Set(G(x) for x in S)
True
```

class `sage.groups.perm_gps.symgp_conjugacy_class.`**`SymmetricGroupConjugacyClassMixin`**(*domain*, *part*)

Bases: `object`

Mixin class which contains methods for conjugacy classes of the symmetric group.

`partition`()

Return the partition of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: g = G([(1,2), (3,4,5)])
sage: C = G.conjugacy_class(g)
```

`sage.groups.perm_gps.symgp_conjugacy_class.`**`conjugacy_class_iterator`**(*part*, *S=None*)

Return an iterator over the conjugacy class associated to the partition `part`.

The elements are given as a list of tuples, each tuple being a cycle.

INPUT:

- `part` – partition
- `S` – (optional, default: $\{1,2,\ldots,n\}$, where n is the size of `part`) a set

OUTPUT:

An iterator over the conjugacy class consisting of all permutations of the set `S` whose cycle type is `part`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.symgp_conjugacy_class import conjugacy_class_iterator
sage: for p in conjugacy_class_iterator([2,2]): print p
[(1, 2), (3, 4)]
[(1, 3), (2, 4)]
[(1, 4), (2, 3)]
```

In order to get permutations, one can use `imap` from the Python module `itertools`:

```
sage: from itertools import imap
sage: S = SymmetricGroup(5)
sage: for p in imap(S, conjugacy_class_iterator([3,2])): print p
(1,2)(3,4,5)
(1,2)(3,5,4)
(1,3)(2,4,5)
(1,3)(2,5,4)
...
(1,4,2)(3,5)
(1,2,3)(4,5)
(1,3,2)(4,5)
```

Check that the number of elements is the number of elements in the conjugacy class:

```
sage: s = lambda p: sum(1 for _ in conjugacy_class_iterator(p))
sage: all(s(p) == p.conjugacy_class_size() for p in Partitions(5))
True
```

It is also possible to specify any underlying set:

```
sage: it = conjugacy_class_iterator([2,2,2], 'abcdef')
sage: next(it)
[('a', 'c'), ('b', 'e'), ('d', 'f')]
sage: next(it)
[('a', 'c'), ('b', 'd'), ('e', 'f')]
```

`sage.groups.perm_gps.symgp_conjugacy_class.`**`default_representative`**`(`*part*, *G*`)`

Construct the default representative for the conjugacy class of cycle type `part` of a symmetric group `G`.

Let λ be a partition of n. We pick a representative by

$$(1, 2, \ldots, \lambda_1)(\lambda_1 + 1, \ldots, \lambda_1 + \lambda_2)(\lambda_1 + \lambda_2 + \cdots + \lambda_{\ell-1}, \ldots, n),$$

where ℓ is the length (or number of parts) of λ.

INPUT:

- `part` – partition
- `G` – a symmetric group

EXAMPLES:

```
sage: from sage.groups.perm_gps.symgp_conjugacy_class import default_representative
sage: S = SymmetricGroup(4)
sage: for p in Partitions(4):
....:     print default_representative(p, S)
(1,2,3,4)
(1,2,3)
(1,2)(3,4)
(1,2)
()
```

CHAPTER

THIRTYTHREE

LIBRARY OF INTERESTING GROUPS

Type `groups.matrix.<tab>` to access examples of groups implemented as permutation groups.

CHAPTER

THIRTYFOUR

BASE CLASSES FOR MATRIX GROUPS

Loading, saving, ... works:

```
sage: G = GL(2,5); G
General Linear Group of degree 2 over Finite Field of size 5
sage: TestSuite(G).run()

sage: g = G.1; g
[4 1]
[4 0]
sage: TestSuite(g).run()
```

We test that trac ticket #9437 is fixed:

```
sage: len(list(SL(2, Zmod(4))))
48
```

AUTHORS:

- William Stein: initial version
- David Joyner (2006-03-15): degree, base_ring, _contains_, list, random, order methods; examples
- William Stein (2006-12): rewrite
- David Joyner (2007-12): Added invariant_generators (with Martin Albrecht and Simon King)
- David Joyner (2008-08): Added module_composition_factors (interface to GAP's MeatAxe implementation) and as_permutation_group (returns isomorphic PermutationGroup).
- Simon King (2010-05): Improve invariant_generators by using GAP for the construction of the Reynolds operator in Singular.

class `sage.groups.matrix_gps.matrix_group.`**`MatrixGroup_base`**

Bases: `sage.groups.group.Group`

Base class for all matrix groups.

This base class just holds the base ring, but not the degree. So it can be a base for affine groups where the natural matrix is larger than the degree of the affine group. Makes no assumption about the group except that its elements have a `matrix()` method.

`as_matrix_group``()`

Return a new matrix group from the generators.

This will throw away any extra structure (encoded in a derived class) that a group of special matrices has.

EXAMPLES:

```
sage: G = SU(4,GF(5))
sage: G.as_matrix_group()
Matrix group over Finite Field in a of size 5^2 with 2 generators (
[      a       0       0       0]  [      1       0 4*a + 3       0]
[      0 2*a + 3       0       0]  [      1       0       0       0]
[      0       0 4*a + 1       0]  [      0 2*a + 4       0       1]
[      0       0       0     3*a], [      0 3*a + 1       0       0]
)

sage: G = GO(3,GF(5))
sage: G.as_matrix_group()
Matrix group over Finite Field of size 5 with 2 generators (
[2 0 0]  [0 1 0]
[0 3 0]  [1 4 4]
[0 0 1], [0 2 1]
)
```

class `sage.groups.matrix_gps.matrix_group.`**`MatrixGroup_gap`**(*degree*, *base_ring*, *libgap_group*, *ambient=None*, *category=None*)

Bases: `sage.groups.libgap_mixin.GroupMixinLibGAP`, `sage.groups.matrix_gps.matrix_group.Matr` `sage.groups.libgap_wrapper.ParentLibGAP`

Base class for matrix groups that implements GAP interface.

INPUT:

- `degree` – integer. The degree (matrix size) of the matrix group.
- `base_ring` – ring. The base ring of the matrices.
- `libgap_group` – the defining libgap group.
- `ambient` – A derived class of `ParentLibGAP` or `None` (default). The ambient class if `libgap_group` has been defined as a subgroup.

TESTS:

```
sage: from sage.groups.matrix_gps.matrix_group import MatrixGroup_gap
sage: MatrixGroup_gap(2, ZZ, libgap.eval('GL(2, Integers)'))
Matrix group over Integer Ring with 3 generators (
[0 1]  [-1  0]  [1 1]
[1 0], [ 0  1], [0 1]
)
```

`Element`

alias of `MatrixGroupElement_gap`

`list`()

List all elements of this group.

This method overrides the matrix group enumerator in GAP which is very slow, see http://tracker.gap-system.org/issues/369.

OUTPUT:

A tuple containing all group elements in a random but fixed order.

EXAMPLES:

```
sage: F = GF(3)
sage: gens = [matrix(F,2, [1,0, -1,1]), matrix(F, 2, [1,1,0,1])]
sage: G = MatrixGroup(gens)
sage: G.cardinality()
24
sage: v = G.list()
sage: len(v)
24
sage: v[:5]
(
[1 0]  [2 0]  [0 1]  [0 2]  [1 2]
[0 1], [0 2], [2 0], [1 0], [2 2]
)
sage: all(g in G for g in G.list())
True
```

An example over a ring (see trac 5241):

```
sage: M1 = matrix(ZZ,2,[[-1,0],[0,1]])
sage: M2 = matrix(ZZ,2,[[1,0],[0,-1]])
sage: M3 = matrix(ZZ,2,[[-1,0],[0,-1]])
sage: MG = MatrixGroup([M1, M2, M3])
sage: MG.list()
(
[1 0]  [ 1  0]  [-1  0]  [-1  0]
[0 1], [ 0 -1], [ 0  1], [ 0 -1]
)
sage: MG.list()[1]
[ 1  0]
[ 0 -1]
sage: MG.list()[1].parent()
Matrix group over Integer Ring with 3 generators (
[-1  0]  [ 1  0]  [-1  0]
[ 0  1], [ 0 -1], [ 0 -1]
)
```

An example over a field (see trac 10515):

```
sage: gens = [matrix(QQ,2,[1,0,0,1])]
sage: MatrixGroup(gens).list()
(
[1 0]
[0 1]
)
```

Another example over a ring (see trac 9437):

```
sage: len(SL(2, Zmod(4)).list())
48
```

An error is raised if the group is not finite:

```
sage: GL(2,ZZ).list()
Traceback (most recent call last):
...
NotImplementedError: group must be finite
```

structure_description(*G*, *latex=False*)

Return a string that tries to describe the structure of `G`.

This methods wraps GAP's `StructureDescription` method.

Requires the *optional* `database_gap` package.

For full details, including the form of the returned string and the algorithm to build it, see GAP's documentation.

INPUT:

- `latex` – a boolean (default: `False`). If `True` return a LaTeX formatted string.

OUTPUT:

- string

> **Warning:** From GAP's documentation: The string returned by `StructureDescription` is **not** an isomorphism invariant: non-isomorphic groups can have the same string value, and two isomorphic groups in different representations can produce different strings.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(6)
sage: G.structure_description()             # optional - database_gap
'C6'
sage: G.structure_description(latex=True)   # optional - database_gap
'C_{6}'
sage: G2 = G.direct_product(G, maps=False)
sage: LatexExpr(G2.structure_description(latex=True))    # optional - database_gap
C_{6} \times C_{6}
```

This method is mainly intended for small groups or groups with few normal subgroups. Even then there are some surprises:

```
sage: D3 = DihedralGroup(3)
sage: D3.structure_description()     # optional - database_gap
'S3'
```

We use the Sage notation for the degree of dihedral groups:

```
sage: D4 = DihedralGroup(4)
sage: D4.structure_description()     # optional - database_gap
'D4'
```

Works for finitely presented groups (trac ticket #17573):

```
sage: F.<x, y> = FreeGroup()
sage: G=F / [x^2*y^-1, x^3*y^2, x*y*x^-1*y^-1]
sage: G.structure_description()      # optional - database_gap
'C7'
```

And matrix groups (trac ticket #17573):

```
sage: groups.matrix.GL(4,2).structure_description() # optional - database_gap
'A8'
```

class `sage.groups.matrix_gps.matrix_group.`**`MatrixGroup_generic`**(*degree*, *base_ring*, *category=None*)

Bases: `sage.groups.matrix_gps.matrix_group.MatrixGroup_base`

Base class for matrix groups over generic base rings

You should not use this class directly. Instead, use one of the more specialized derived classes.

INPUT:

- `degree` – integer. The degree (matrix size) of the matrix group.
- `base_ring` – ring. The base ring of the matrices.

TESTS:

```
sage: G = GL(2, QQ)
sage: from sage.groups.matrix_gps.matrix_group import MatrixGroup_generic
sage: isinstance(G, MatrixGroup_generic)
True
```

Element
alias of `MatrixGroupElement_generic`

degree()
Return the degree of this matrix group.

OUTPUT:

Integer. The size (number of rows equals number of columns) of the matrices.

EXAMPLES:

```
sage: SU(5,5).degree()
5
```

hom(*x*)
Return the group homomorphism defined by `x`

INPUT:

- `x` – a list/tuple/iterable of matrix group elements.

OUTPUT:

The group homomorphism defined by `x`.

EXAMPLES:

```
sage: G = MatrixGroup([matrix(GF(5), [[1,3],[0,1]])])
sage: H = MatrixGroup([matrix(GF(5), [[1,2],[0,1]])])
sage: G.hom([H.gen(0)])
Homomorphism : Matrix group over Finite Field of size 5 with 1 generators (
[1 3]
[0 1]
) --> Matrix group over Finite Field of size 5 with 1 generators (
[1 2]
[0 1]
)
```

matrix_space()
Return the matrix space corresponding to this matrix group.

This is a matrix space over the field of definition of this matrix group.

EXAMPLES:

```
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS(1), MS([1,2,3,4])])
sage: G.matrix_space()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 5
sage: G.matrix_space() is MS
True
```

`sage.groups.matrix_gps.matrix_group.`**`is_MatrixGroup`**(*x*)

Test whether `x` is a matrix group.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.matrix_group import is_MatrixGroup
sage: is_MatrixGroup(MatrixSpace(QQ,3))
False
sage: is_MatrixGroup(Mat(QQ,3))
False
sage: is_MatrixGroup(GL(2,ZZ))
True
sage: is_MatrixGroup(MatrixGroup([matrix(2,[1,1,0,1])]))
True
```

CHAPTER

THIRTYFIVE

MATRIX GROUP ELEMENTS

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 3 with 2 generators (
[1 0]  [1 1]
[0 1], [0 1]
)
sage: g = G([[1,1],[0,1]])
sage: h = G([[1,2],[0,1]])
sage: g*h
[1 0]
[0 1]
```

You cannot add two matrices, since this is not a group operation. You can coerce matrices back to the matrix space and add them there:

```
sage: g + h
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +:
'FinitelyGeneratedMatrixGroup_gap_with_category.element_class' and
'FinitelyGeneratedMatrixGroup_gap_with_category.element_class'

sage: g.matrix() + h.matrix()
[2 0]
[0 2]
```

Similarly, you cannot multiply group elements by scalars but you can do it with the underlying matrices:

```
sage: 2*g
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '*': 'Integer Ring' and 'Matrix group over Finite Field
[1 0]  [1 1]
[0 1], [0 1]
)'
```

AUTHORS:

- David Joyner (2006-05): initial version David Joyner
- David Joyner (2006-05): various modifications to address William Stein's TODO's.
- William Stein (2006-12-09): many revisions.

- Volker Braun (2013-1) port to new Parent, libGAP.

class `sage.groups.matrix_gps.group_element.`**`MatrixGroupElement_base`**

Bases: `sage.structure.element.MultiplicativeGroupElement`

Base class for elements of matrix groups.

You should use one of the two subclasses:

- `MatrixGroupElement_sage` implements the group multiplication using Sage matrices.
- `MatrixGroupElement_gap` implements the group multiplication using libGAP matrices.

The base class only assumes that derived classes implement `matrix()`.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: g = G.random_element()
sage: type(g)
<class 'sage.groups.matrix_gps.group_element.FinitelyGeneratedMatrixGroup_gap_with_category.elem
```

`list()`

Return list representation of this matrix.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: g = G.0
sage: g
[1 0]
[0 1]
sage: g.list()
[[1, 0], [0, 1]]
```

class `sage.groups.matrix_gps.group_element.`**`MatrixGroupElement_gap`**(*parent*, *M*, *check=True*, *convert=True*)

Bases: `sage.groups.libgap_mixin.GroupElementMixinLibGAP`, `sage.groups.matrix_gps.group_element.MatrixGroupElement_base`, `sage.groups.libgap_wrapper.ElementLibGAP`

Element of a matrix group over a generic ring.

The group elements are implemented as Sage matrices.

INPUT:

- `M` – a matrix.
- `parent` – the parent.
- `check` – bool (default: `True`). If true does some type checking.
- `convert` – bool (default: `True`). If true convert `M` to the right matrix space.

TESTS:

```
sage: MS = MatrixSpace(GF(3),2,2)
sage: G = MatrixGroup(MS([[1,0],[0,1]]), MS([[1,1],[0,1]]))
sage: G.gen(0)
```

```
[1 0]
[0 1]
sage: g = G.random_element()
sage: TestSuite(g).run()
```

matrix()

Obtain the usual matrix (as an element of a matrix space) associated to this matrix group element.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.gen(0).matrix()
[1 0]
[0 1]
sage: _.parent()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 3
```

class sage.groups.matrix_gps.group_element.**MatrixGroupElement_generic**(*parent*, *M*, *check=True*, *convert=True*)

Bases: sage.groups.matrix_gps.group_element.MatrixGroupElement_base

Element of a matrix group over a generic ring.

The group elements are implemented as Sage matrices.

INPUT:

- M – a matrix.
- parent – the parent.
- **check – bool (default: True). If true does some** type checking.
- convert – bool (default: True). If true convert M to the right matrix space.

TESTS:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: g = G.random_element()
sage: TestSuite(g).run()
```

inverse()

Return the inverse group element

OUTPUT:

A matrix group element.

EXAMPLES:

```
sage: G = GL(2,3)
sage: g = G([1,2,1,0]); g
[1 2]
[1 0]
sage: g.__invert__()
[0 1]
```

```
[2 1]
sage: g * (~g)
[1 0]
[0 1]
```

matrix()

Obtain the usual matrix (as an element of a matrix space) associated to this matrix group element.

One reason to compute the associated matrix is that matrices support a huge range of functionality.

EXAMPLES:

```
sage: k = GF(7); G = MatrixGroup([matrix(k,2,[1,1,0,1]), matrix(k,2,[1,0,0,2])])
sage: g = G.0
sage: g.matrix()
[1 1]
[0 1]
sage: parent(g.matrix())
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 7
```

Matrices have extra functionality that matrix group elements do not have:

```
sage: g.matrix().charpoly('t')
t^2 + 5*t + 1
```

sage.groups.matrix_gps.group_element.**is_MatrixGroupElement**(*x*)

Test whether x is a matrix group element

INPUT:

- x – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.group_element import is_MatrixGroupElement
sage: is_MatrixGroupElement('helloooo')
False

sage: G = GL(2,3)
sage: is_MatrixGroupElement(G.an_element())
True
```

CHAPTER

THIRTYSIX

FINITELY GENERATED MATRIX GROUPS

This class is designed for computing with matrix groups defined by a finite set of generating matrices.

EXAMPLES:

```
sage: F = GF(3)
sage: gens = [matrix(F,2, [1,0, -1,1]), matrix(F,2, [1,1,0,1])]
sage: G = MatrixGroup(gens)
sage: G.conjugacy_class_representatives()
(
[1 0]  [0 2]  [0 1]  [2 0]  [0 2]  [0 1]  [0 2]
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
```

The finitely generated matrix groups can also be constructed as subgroups of matrix groups:

```
sage: SL2Z = SL(2,ZZ)
sage: S, T = SL2Z.gens()
sage: SL2Z.subgroup([T^2])
Matrix group over Integer Ring with 1 generators (
[1 2]
[0 1]
)
```

AUTHORS:

- William Stein: initial version
- David Joyner (2006-03-15): degree, base_ring, _contains_, list, random, order methods; examples
- William Stein (2006-12): rewrite
- David Joyner (2007-12): Added invariant_generators (with Martin Albrecht and Simon King)
- David Joyner (2008-08): Added module_composition_factors (interface to GAP's MeatAxe implementation) and as_permutation_group (returns isomorphic PermutationGroup).
- Simon King (2010-05): Improve invariant_generators by using GAP for the construction of the Reynolds operator in Singular.
- Volker Braun (2013-1) port to new Parent, libGAP.

class `sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_gap`(*degree*, *base_ring*, *libgap_group*, *ambient=None*, *category=None*)

Bases: `sage.groups.matrix_gps.matrix_group.MatrixGroup_gap`

Matrix group generated by a finite number of matrices.

EXAMPLES:

```
sage: m1 = matrix(GF(11), [[1,2],[3,4]])
sage: m2 = matrix(GF(11), [[1,3],[10,0]])
sage: G = MatrixGroup(m1, m2);  G
Matrix group over Finite Field of size 11 with 2 generators (
[1 2]  [ 1  3]
[3 4], [10  0]
)
sage: type(G)
<class 'sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_gap_with_category
sage: TestSuite(G).run()
```

as_permutation_group(*algorithm=None*)

Return a permutation group representation for the group.

In most cases occurring in practice, this is a permutation group of minimal degree (the degree begin determined from orbits under the group action). When these orbits are hard to compute, the procedure can be time-consuming and the degree may not be minimal.

INPUT:

- `algorithm` – `None` or `'smaller'`. In the latter case, try harder to find a permutation representation of small degree.

OUTPUT:

A permutation group isomorphic to `self`. The `algorithm='smaller'` option tries to return an isomorphic group of low degree, but is not guaranteed to find the smallest one.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2), 5, 5)
sage: A = MS([[0,0,0,0,1],[0,0,0,1,0],[0,0,1,0,0],[0,1,0,0,0],[1,0,0,0,0]])
sage: G = MatrixGroup([A])
sage: G.as_permutation_group()
Permutation Group with generators [(1,2)]
sage: MS = MatrixSpace( GF(7), 12, 12)
sage: GG = gap("ImfMatrixGroup( 12, 3 )")
sage: GG.GeneratorsOfGroup().Length()
3
sage: g1 = MS(eval(str(GG.GeneratorsOfGroup()[1]).replace("\n","")))
sage: g2 = MS(eval(str(GG.GeneratorsOfGroup()[2]).replace("\n","")))
sage: g3 = MS(eval(str(GG.GeneratorsOfGroup()[3]).replace("\n","")))
sage: G = MatrixGroup([g1, g2, g3])
sage: G.cardinality()
21499084800
```

```
sage: set_random_seed(0); current_randstate().set_seed_gap()
sage: P = G.as_permutation_group()
sage: P.cardinality()
21499084800
sage: P.degree()   # random output
144
sage: set_random_seed(3); current_randstate().set_seed_gap()
sage: Psmaller = G.as_permutation_group(algorithm="smaller")
sage: Psmaller.cardinality()
21499084800
sage: Psmaller.degree()   # random output
108
```

In this case, the "smaller" option returned an isomorphic group of lower degree. The above example used GAP's library of irreducible maximal finite ("imf") integer matrix groups to construct the MatrixGroup G over GF(7). The section "Irreducible Maximal Finite Integral Matrix Groups" in the GAP reference manual has more details.

TESTS:

```
sage: A= matrix(QQ, 2, [0, 1, 1, 0])
sage: B= matrix(QQ, 2, [1, 0, 0, 1])
sage: a, b= MatrixGroup([A, B]).as_permutation_group().gens()
sage: a.order(), b.order()
(2, 1)
```

invariant_generators()

Return invariant ring generators.

Computes generators for the polynomial ring $F[x_1, \ldots, x_n]^G$, where G in $GL(n, F)$ is a finite matrix group.

In the "good characteristic" case the polynomials returned form a minimal generating set for the algebra of G-invariant polynomials. In the "bad" case, the polynomials returned are primary and secondary invariants, forming a not necessarily minimal generating set for the algebra of G-invariant polynomials.

ALGORITHM:

Wraps Singular's `invariant_algebra_reynolds` and `invariant_ring` in `finvar.lib`.

EXAMPLES:

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[0,1],[-1,0]]),MS([[1,1],[2,3]])]
sage: G = MatrixGroup(gens)
sage: G.invariant_generators()
[x1^7*x2 - x1*x2^7,
 x1^12 - 2*x1^9*x2^3 - x1^6*x2^6 + 2*x1^3*x2^9 + x2^12,
 x1^18 + 2*x1^15*x2^3 + 3*x1^12*x2^6 + 3*x1^6*x2^12 - 2*x1^3*x2^15 + x2^18]

sage: q = 4; a = 2
sage: MS = MatrixSpace(QQ, 2, 2)
sage: gen1 = [[1/a,(q-1)/a],[1/a, -1/a]]; gen2 = [[1,0],[0,-1]]; gen3 = [[-1,0],[0,1]]
sage: G = MatrixGroup([MS(gen1),MS(gen2),MS(gen3)])
sage: G.cardinality()
12
sage: G.invariant_generators()
[x1^2 + 3*x2^2, x1^6 + 15*x1^4*x2^2 + 15*x1^2*x2^4 + 33*x2^6]

sage: F = CyclotomicField(8)
```

```
sage: z = F.gen()
sage: a = z+1/z
sage: b = z^2
sage: MS = MatrixSpace(F,2,2)
sage: g1 = MS([[1/a, 1/a], [1/a, -1/a]])
sage: g2 = MS([[-b, 0], [0, b]])
sage: G=MatrixGroup([g1,g2])
sage: G.invariant_generators()
[x1^4 + 2*x1^2*x2^2 + x2^4,
 x1^5*x2 - x1*x2^5,
 x1^8 + 28/9*x1^6*x2^2 + 70/9*x1^4*x2^4 + 28/9*x1^2*x2^6 + x2^8]
```

AUTHORS:

- David Joyner, Simon King and Martin Albrecht.

REFERENCES:

- Singular reference manual
- B. Sturmfels, "Algorithms in invariant theory", Springer-Verlag, 1993.
- S. King, "Minimal Generating Sets of non-modular invariant rings of finite groups", Arxiv math/0703035.

module_composition_factors(*algorithm=None*)

Return a list of triples consisting of [base field, dimension, irreducibility], for each of the Meataxe composition factors modules. The `algorithm="verbose"` option returns more information, but in Meataxe notation.

EXAMPLES:

```
sage: F=GF(3);MS=MatrixSpace(F,4,4)
sage: M=MS(0)
sage: M[0,1]=1;M[1,2]=1;M[2,3]=1;M[3,0]=1
sage: G = MatrixGroup([M])
sage: G.module_composition_factors()
[(Finite Field of size 3, 1, True),
 (Finite Field of size 3, 1, True),
 (Finite Field of size 3, 2, True)]
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[0,1],[-1,0]]),MS([[1,1],[2,3]])]
sage: G = MatrixGroup(gens)
sage: G.module_composition_factors()
[(Finite Field of size 7, 2, True)]
```

Type `G.module_composition_factors(algorithm='verbose')` to get a more verbose version.

For more on MeatAxe notation, see http://www.gap-system.org/Manuals/doc/ref/chap69.html

class sage.groups.matrix_gps.finitely_generated.**FinitelyGeneratedMatrixGroup_generic**(*degree, base_ring, generator_matrices, category=None*)

Bases: `sage.groups.matrix_gps.matrix_group.MatrixGroup_generic`

Matrix group generated by a finite number of matrices.

EXAMPLES:

```
sage: m1 = matrix(SR, [[1,2],[3,4]])
sage: m2 = matrix(SR, [[1,3],[-1,0]])
sage: G = MatrixGroup(m1, m2)
sage: TestSuite(G).run()
sage: type(G)
<class 'sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_generic_with_cate

sage: from sage.groups.matrix_gps.finitely_generated import              ....:     FinitelyGenera
sage: G = FinitelyGeneratedMatrixGroup_generic(2, QQ, [matrix(QQ,[[1,2],[3,4]])])
sage: G.gens()
(
[1 2]
[3 4]
)
```

gen (*i*)

Return the i-th generator

OUTPUT:

The i-th generator of the group.

EXAMPLES:

```
sage: H = GL(2, GF(3))
sage: h1, h2 = H([[1,0],[2,1]]), H([[1,1],[0,1]])
sage: G = H.subgroup([h1, h2])
sage: G.gen(0)
[1 0]
[2 1]
sage: G.gen(0).matrix() == h1.matrix()
True
```

gens ()

Return the generators of the matrix group.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]), MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: gens[0] in G
True
sage: gens = G.gens()
sage: gens[0] in G
True
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS(1), MS([1,2,3,4])])
sage: G
Matrix group over Finite Field of size 5 with 2 generators (
[1 0]  [1 2]
[0 1], [3 4]
)
sage: G.gens()
```

```
(
[1 0]  [1 2]
[0 1], [3 4]
)
```

ngens()

Return the number of generators

OUTPUT:

An integer. The number of generators.

EXAMPLES:

```
sage: H = GL(2, GF(3))
sage: h1, h2 = H([[1,0],[2,1]]), H([[1,1],[0,1]])
sage: G = H.subgroup([h1, h2])
sage: G.ngens()
2
```

sage.groups.matrix_gps.finitely_generated.**MatrixGroup**(**gens*, ***kwds*)

Return the matrix group with given generators.

INPUT:

- `*gens` – matrices, or a single list/tuple/iterable of matrices, or a matrix group.
- `check` – boolean keyword argument (optional, default: `True`). Whether to check that each matrix is invertible.

EXAMPLES:

```
sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 5 with 2 generators (
[1 2]  [1 1]
[4 1], [0 1]
)
```

In the second example, the generators are a matrix over $\mathbf{Z}$, a matrix over a finite field, and the integer 2. Sage determines that they both canonically map to matrices over the finite field, so creates that matrix group there:

```
sage: gens = [matrix(2,[1,2, -1, 1]), matrix(GF(7), 2, [1,1, 0,1]), 2]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 7 with 3 generators (
[1 2]  [1 1]  [2 0]
[6 1], [0 1], [0 2]
)
```

Each generator must be invertible:

```
sage: G = MatrixGroup([matrix(ZZ,2,[1,2,3,4])])
Traceback (most recent call last):
...
ValueError: each generator must be an invertible matrix

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: MatrixGroup([MS.0])
Traceback (most recent call last):
...
ValueError: each generator must be an invertible matrix
```

```
sage: MatrixGroup([MS.0], check=False)  # works formally but is mathematical nonsense
Matrix group over Finite Field of size 5 with 1 generators (
[1 0]
[0 0]
)
```

Some groups are not supported, or do not have much functionality implemented:

```
sage: G = SL(0, QQ)
Traceback (most recent call last):
...
ValueError: the degree must be at least 1

sage: SL2C = SL(2, CC);  SL2C
Special Linear Group of degree 2 over Complex Field with 53 bits of precision
sage: SL2C.gens()
Traceback (most recent call last):
...
AttributeError: 'LinearMatrixGroup_generic_with_category' object has no attribute 'gens'
```

sage.groups.matrix_gps.finitely_generated.**QuaternionMatrixGroupGF3**()

The quaternion group as a set of 2×2 matrices over $GF(3)$.

OUTPUT:

A matrix group consisting of 2×2 matrices with elements from the finite field of order 3. The group is the quaternion group, the nonabelian group of order 8 that is not isomorphic to the group of symmetries of a square (the dihedral group D_4).

Note: This group is most easily available via `groups.matrix.QuaternionGF3()`.

EXAMPLES:

The generators are the matrix representations of the elements commonly called I and J, while K is the product of I and J.

```
sage: from sage.groups.matrix_gps.finitely_generated import QuaternionMatrixGroupGF3
sage: Q = QuaternionMatrixGroupGF3()
sage: Q.order()
8
sage: aye = Q.gens()[0]; aye
[1 1]
[1 2]
sage: jay = Q.gens()[1]; jay
[2 1]
[1 1]
sage: kay = aye*jay; kay
[0 2]
[1 0]
```

TESTS:

```
sage: groups.matrix.QuaternionGF3()
Matrix group over Finite Field of size 3 with 2 generators (
[1 1]  [2 1]
[1 2], [1 1]
)

sage: Q = QuaternionMatrixGroupGF3()
```

```
sage: QP = Q.as_permutation_group()
sage: QP.is_isomorphic(QuaternionGroup())
True
sage: H = DihedralGroup(4)
sage: H.order()
8
sage: QP.is_abelian(), H.is_abelian()
(False, False)
sage: QP.is_isomorphic(H)
False
```

sage.groups.matrix_gps.finitely_generated.**normalize_square_matrices**(*matrices*)

Find a common space for all matrices.

OUTPUT:

A list of matrices, all elements of the same matrix space.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.finitely_generated import normalize_square_matrices
sage: m1 = [[1,2],[3,4]]
sage: m2 = [2, 3, 4, 5]
sage: m3 = matrix(QQ, [[1/2,1/3],[1/4,1/5]])
sage: m4 = MatrixGroup(m3).gen(0)
sage: normalize_square_matrices([m1, m2, m3, m4])
[
[1 2]  [2 3]  [1/2 1/3]  [1/2 1/3]
[3 4], [4 5], [1/4 1/5], [1/4 1/5]
]
```

CHAPTER

THIRTYSEVEN

HOMOMORPHISMS BETWEEN MATRIX GROUPS

AUTHORS:

- David Joyner and William Stein (2006-03): initial version
- David Joyner (2006-05): examples
- Simon King (2011-01): cleaning and improving code
- Volker Braun (2013-1) port to new Parent, libGAP.

class sage.groups.matrix_gps.morphism.**MatrixGroupMap**(*parent*)

Bases: sage.categories.morphism.Morphism

Set-theoretic map between matrix groups.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.morphism import MatrixGroupMap
sage: MatrixGroupMap(ZZ.Hom(ZZ))   # mathematical nonsense
MatrixGroup endomorphism of Integer Ring
```

class sage.groups.matrix_gps.morphism.**MatrixGroupMorphism**(*parent*)

Bases: sage.groups.matrix_gps.morphism.MatrixGroupMap

Set-theoretic map between matrix groups.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.morphism import MatrixGroupMap
sage: MatrixGroupMap(ZZ.Hom(ZZ))   # mathematical nonsense
MatrixGroup endomorphism of Integer Ring
```

class sage.groups.matrix_gps.morphism.**MatrixGroupMorphism_im_gens**(*homset*, *imgsH*, *check=True*)

Bases: sage.groups.matrix_gps.morphism.MatrixGroupMorphism

Group morphism specified by images of generators.

Some Python code for wrapping GAP's GroupHomomorphismByImages function but only for matrix groups. Can be expensive if G is large.

EXAMPLES:

```
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS([1,1,0,1])])
sage: H = MatrixGroup([MS([1,0,1,1])])
sage: phi = G.hom(H.gens())
sage: phi
Homomorphism : Matrix group over Finite Field of size 5 with 1 generators (
[1 1]
```

```
[0 1]
) --> Matrix group over Finite Field of size 5 with 1 generators (
[1 0]
[1 1]
)
sage: phi(MS([1,1,0,1]))
[1 0]
[1 1]
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: F.multiplicative_generator()
3
sage: G = MatrixGroup([MS([3,0,0,1])])
sage: a = G.gens()[0]^2
sage: phi = G.hom([a])
```

TESTS:

Check that trac ticket #19406 is fixed:

```
sage: G = GL(2, GF(3))
sage: H = GL(3, GF(2))
sage: mat1 = H([[-1,0,0],[0,0,-1],[0,-1,0]])
sage: mat2 = H([[1,1,1],[0,0,-1],[-1,0,0]])
sage: phi = G.hom([mat1, mat2])
Traceback (most recent call last):
...
TypeError: images do not define a group homomorphism
```

gap()

Return the underlying LibGAP group homomorphism

OUTPUT:

A LibGAP element.

EXAMPLES:

```
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS([1,1,0,1])])
sage: H = MatrixGroup([MS([1,0,1,1])])
sage: phi = G.hom(H.gens())
sage: phi.gap()
CompositionMapping( [ (6,7,8,10,9)(11,13,14,12,15)(16,19,20,18,17)(21,25,22,24,23) ]
-> [ [ [ Z(5)^0, 0*Z(5) ], [ Z(5)^0, Z(5)^0 ] ] ], <action isomorphism> )
sage: type(_)
<type 'sage.libs.gap.element.GapElement'>
```

image(*J*, **args*, ***kwds*)

The image of an element or a subgroup.

INPUT:

`J` – a subgroup or an element of the domain of `self`.

OUTPUT:

The image of `J` under `self`

NOTE:

`pushforward` is the method that is used when a map is called on anything that is not an element of its domain. For historical reasons, we keep the alias `image()` for this method.

EXAMPLES:

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: F.multiplicative_generator()
3
sage: G = MatrixGroup([MS([3,0,0,1])])
sage: a = G.gens()[0]^2
sage: phi = G.hom([a])
sage: phi.image(G.gens()[0]) # indirect doctest
[2 0]
[0 1]
sage: H = MatrixGroup([MS(a.list())])
sage: H
Matrix group over Finite Field of size 7 with 1 generators (
[2 0]
[0 1]
)
```

The following tests against trac ticket #10659:

```
sage: phi(H)   # indirect doctestest
Matrix group over Finite Field of size 7 with 1 generators (
[4 0]
[0 1]
)
```

kernel()

Return the kernel of `self`, i.e., a matrix group.

EXAMPLES:

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: F.multiplicative_generator()
3
sage: G = MatrixGroup([MS([3,0,0,1])])
sage: a = G.gens()[0]^2
sage: phi = G.hom([a])
sage: phi.kernel()
Matrix group over Finite Field of size 7 with 1 generators (
[6 0]
[0 1]
)
```

pushforward(*J*, **args*, ***kwds*)

The image of an element or a subgroup.

INPUT:

`J` – a subgroup or an element of the domain of `self`.

OUTPUT:

The image of `J` under `self`

NOTE:

`pushforward` is the method that is used when a map is called on anything that is not an element of its domain. For historical reasons, we keep the alias `image()` for this method.

EXAMPLES:

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: F.multiplicative_generator()
```

```
3
sage: G = MatrixGroup([MS([3,0,0,1])])
sage: a = G.gens()[0]^2
sage: phi = G.hom([a])
sage: phi.image(G.gens()[0]) # indirect doctest
[2 0]
[0 1]
sage: H = MatrixGroup([MS(a.list())])
sage: H
Matrix group over Finite Field of size 7 with 1 generators (
[2 0]
[0 1]
)
```

The following tests against trac ticket #10659:

```
sage: phi(H)   # indirect doctestest
Matrix group over Finite Field of size 7 with 1 generators (
[4 0]
[0 1]
)
```

sage.groups.matrix_gps.morphism.**to_libgap**(*x*)

Helper to convert x to a LibGAP matrix or matrix group element.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.morphism import to_libgap
sage: to_libgap(GL(2,3).gen(0))
[ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ]
sage: to_libgap(matrix(QQ, [[1,2],[3,4]]))
[ [ 1, 2 ], [ 3, 4 ] ]
```

CHAPTER

THIRTYEIGHT

MATRIX GROUP HOMSETS

AUTHORS:

- William Stein (2006-05-07): initial version
- Volker Braun (2013-1) port to new Parent, libGAP

class sage.groups.matrix_gps.homset.**MatrixGroupHomset**(*G*, *H*, *category=None*)

Bases: sage.groups.group_homset.GroupHomset_generic

Return the homset of two matrix groups.

INPUT:

- G – a matrix group
- H – a matrix group

OUTPUT:

The homset of two matrix groups.

EXAMPLES:

```
sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
sage: G = MatrixGroup(gens)
sage: from sage.groups.matrix_gps.homset import MatrixGroupHomset
sage: MatrixGroupHomset(G, G)
Set of Homomorphisms from
Matrix group over Finite Field of size 5 with 2 generators (
[1 2]  [1 1]
[4 1], [0 1]
) to Matrix group over Finite Field of size 5 with 2 generators (
[1 2]  [1 1]
[4 1], [0 1]
)
```

sage.groups.matrix_gps.homset.**is_MatrixGroupHomset**(*x*)

Test whether x is a homset.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.homset import is_MatrixGroupHomset
sage: is_MatrixGroupHomset(4)
False

sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
sage: G = MatrixGroup(gens)
```

```
sage: from sage.groups.matrix_gps.homset import MatrixGroupHomset
sage: M = MatrixGroupHomset(G, G)
sage: is_MatrixGroupHomset(M)
True
```

CHAPTER

THIRTYNINE

COXETER GROUPS AS MATRIX GROUPS

This implements a general Coxeter group as a matrix group by using the reflection representation.

AUTHORS:

- Travis Scrimshaw (2013-08-28): Initial version

class `sage.groups.matrix_gps.coxeter_group.`**`CoxeterMatrixGroup`**(*coxeter_matrix*, *base_ring*, *index_set*)

Bases: `sage.groups.matrix_gps.finitely_generated.FinitelyGeneratedMatrixGroup_generic`, `sage.structure.unique_representation.UniqueRepresentation`

A Coxeter group represented as a matrix group.

Let (W, S) be a Coxeter system. We construct a vector space V over $\mathbf{R}$ with a basis of $\{\alpha_s\}_{s \in S}$ and inner product

$$B(\alpha_s, \alpha_t) = -\cos\left(\frac{\pi}{m_{st}}\right)$$

where we have $B(\alpha_s, \alpha_t) = -1$ if $m_{st} = \infty$. Next we define a representation $\sigma_s : V \to V$ by

$$\sigma_s \lambda = \lambda - 2B(\alpha_s, \lambda)\alpha_s.$$

This representation is faithful so we can represent the Coxeter group W by the set of matrices σ_s acting on V.

INPUT:

- `data` – a Coxeter matrix or graph or a Cartan type
- `base_ring` – (default: the universal cyclotomic field) the base ring which contains all values $\cos(\pi/m_{ij})$ where $(m_{ij})_{ij}$ is the Coxeter matrix
- `index_set` – (optional) an indexing set for the generators

For more on creating Coxeter groups, see `CoxeterGroup()`.

Todo

Currently the label ∞ is implemented as -1 in the Coxeter matrix.

EXAMPLES:

We can create Coxeter groups from Coxeter matrices:

```
sage: W = CoxeterGroup([[1, 6, 3], [6, 1, 10], [3, 10, 1]])
sage: W
Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[ 1  6  3]
```

```
[ 6  1 10]
[ 3 10  1]
sage: W.gens()
(
[                  -1  -E(12)^7 + E(12)^11                    1]
[                   0                    1                    0]
[                   0                    0                    1],

[                   1                    0                    0]
[-E(12)^7 + E(12)^11                   -1      E(20) - E(20)^9]
[                   0                    0                    1],

[              1               0                0]
[              0               1                0]
[              1 E(20) - E(20)^9               -1]
)
sage: m = matrix([[1,3,3,3], [3,1,3,2], [3,3,1,2], [3,2,2,1]])
sage: W = CoxeterGroup(m)
sage: W.gens()
(
[-1  1  1  1]  [ 1  0  0  0]  [ 1  0  0  0]  [ 1  0  0  0]
[ 0  1  0  0]  [ 1 -1  1  0]  [ 0  1  0  0]  [ 0  1  0  0]
[ 0  0  1  0]  [ 0  0  1  0]  [ 1  1 -1  0]  [ 0  0  1  0]
[ 0  0  0  1], [ 0  0  0  1], [ 0  0  0  1], [ 1  0  0 -1]
)
sage: a,b,c,d = W.gens()
sage: (a*b*c)^3
[ 5  1 -5  7]
[ 5  0 -4  5]
[ 4  1 -4  4]
[ 0  0  0  1]
sage: (a*b)^3
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: b*d == d*b
True
sage: a*c*a == c*a*c
True
```

We can create the matrix representation over different base rings and with different index sets. Note that the base ring must contain all $2*\cos(\pi/m_{ij})$ where $(m_{ij})_{ij}$ is the Coxeter matrix:

```
sage: W = CoxeterGroup(m, base_ring=RR, index_set=['a','b','c','d'])
sage: W.base_ring()
Real Field with 53 bits of precision
sage: W.index_set()
('a', 'b', 'c', 'd')

sage: CoxeterGroup(m, base_ring=ZZ)
Coxeter group over Integer Ring with Coxeter matrix:
[1 3 3 3]
[3 1 3 2]
[3 3 1 2]
[3 2 2 1]
sage: CoxeterGroup([[1,4],[4,1]], base_ring=QQ)
Traceback (most recent call last):
```

```
...
TypeError: unable to convert sqrt(2) to a rational
```

Using the well-known conversion between Coxeter matrices and Coxeter graphs, we can input a Coxeter graph. Following the standard convention, edges with no label (i.e. labelled by `None`) are treated as 3:

```
sage: G = Graph([(0,3,None), (1,3,15), (2,3,7), (0,1,3)])
sage: W = CoxeterGroup(G); W
Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[ 1  3  2  3]
[ 3  1  2 15]
[ 2  2  1  7]
[ 3 15  7  1]
sage: G2 = W.coxeter_diagram()
sage: CoxeterGroup(G2) is W
True
```

Because there currently is no class for $\mathbf{Z} \cup \{\infty\}$, labels of ∞ are given by -1 in the Coxeter matrix:

```
sage: G = Graph([(0,1,None), (1,2,4), (0,2,oo)])
sage: W = CoxeterGroup(G)
sage: W.coxeter_matrix()
[ 1  3 -1]
[ 3  1  4]
[-1  4  1]
```

We can also create Coxeter groups from Cartan types using the `implementation` keyword:

```
sage: W = CoxeterGroup(['D',5], implementation="reflection")
sage: W
Finite Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3 2 2 2]
[3 1 3 2 2]
[2 3 1 3 3]
[2 2 3 1 2]
[2 2 3 2 1]
sage: W = CoxeterGroup(['H',3], implementation="reflection")
sage: W
Finite Coxeter group over Universal Cyclotomic Field with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1]
```

class Element (*parent*, *M*, *check=True*, *convert=True*)

Bases: `sage.groups.matrix_gps.group_element.MatrixGroupElement_generic`

A Coxeter group element.

canonical_matrix()

Return the matrix of `self` in the canonical faithful representation, which is `self` as a matrix.

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation="reflection")
sage: a,b,c = W.gens()
sage: elt = a*b*c
sage: elt.canonical_matrix()
[ 0  0 -1]
[ 1  0 -1]
[ 0  1 -1]
```

has_right_descent (*i*)

Return whether `i` is a right descent of `self`.

A Coxeter system (W, S) has a root system defined as $\{w(\alpha_s)\}_{w \in W}$ and we define the positive (resp. negative) roots $\alpha = \sum_{s \in S} c_s \alpha_s$ by all $c_s \geq 0$ (resp. $c_s \leq 0$). In particular, we note that if $\ell(ws) > \ell(w)$ then $w(\alpha_s) > 0$ and if $\ell(ws) < \ell(w)$ then $w(\alpha_s) < 0$. Thus $i \in I$ is a right descent if $w(\alpha_{s_i}) < 0$ or equivalently if the matrix representing w has all entries of the i-th column being non-positive.

INPUT:

- `i` – an element in the index set

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation="reflection")
sage: a,b,c = W.gens()
sage: elt = b*a*c
sage: map(lambda i: elt.has_right_descent(i), [1, 2, 3])
[True, False, True]
```

`CoxeterMatrixGroup.`**bilinear_form**()

Return the bilinear form associated to `self`.

Given a Coxeter group G with Coxeter matrix $M = (m_{ij})_{ij}$, the associated bilinear form $A = (a_{ij})_{ij}$ is given by

$$a_{ij} = -\cos\left(\frac{\pi}{m_{ij}}\right).$$

If A is positive definite, then G is of finite type (and so the associated Coxeter group is a finite group). If A is positive semidefinite, then G is affine type.

EXAMPLES:

```
sage: W = CoxeterGroup(['D',4])
sage: W.bilinear_form()
[   1 -1/2    0    0]
[-1/2    1 -1/2 -1/2]
[   0 -1/2    1    0]
[   0 -1/2    0    1]
```

`CoxeterMatrixGroup.`**canonical_representation**()

Return the canonical faithful representation of `self`, which is `self`.

EXAMPLES:

```
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.canonical_representation() is W
True
```

`CoxeterMatrixGroup.`**coxeter_diagram**()

Return the Coxeter diagram of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['H',3], implementation="reflection")
sage: G = W.coxeter_diagram(); G
Graph on 3 vertices
sage: G.edges()
[(1, 2, 3), (2, 3, 5)]
sage: CoxeterGroup(G) is W
True
sage: G = Graph([(0, 1, 3), (1, 2, oo)])
```

```
sage: W = CoxeterGroup(G)
sage: W.coxeter_diagram() == G
True
sage: CoxeterGroup(W.coxeter_diagram()) is W
True
```

CoxeterMatrixGroup.**coxeter_graph**(**args*, ***kwds*)

Deprecated: Use `coxeter_diagram()` instead. See trac ticket #17798 for details.

CoxeterMatrixGroup.**coxeter_matrix**()

Return the Coxeter matrix of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.coxeter_matrix()
[1 3]
[3 1]
sage: W = CoxeterGroup(['H',3])
sage: W.coxeter_matrix()
[1 3 2]
[3 1 5]
[2 5 1]
```

CoxeterMatrixGroup.**index_set**()

Return the index set of `self`.

EXAMPLES:

```
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.index_set()
(1, 2)
sage: W = CoxeterGroup([[1,3],[3,1]], index_set=['x', 'y'])
sage: W.index_set()
('x', 'y')
sage: W = CoxeterGroup(['H',3])
sage: W.index_set()
(1, 2, 3)
```

CoxeterMatrixGroup.**is_finite**()

Return `True` if this group is finite.

EXAMPLES:

```
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2],[3,1,l],[2,l,1]]).is_finite()]
....:
[2, 3, 4, 5]
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2,2],[3,1,l,2],[2,l,1,3],[2,2,3,1]]).is_finite()]
....:
[2, 3, 4]
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2,2,2], [3,1,3,3,2], [2,3,1,2,2],
....:                [2,3,2,1,l], [2,2,2,l,1]]).is_finite()]
....:
[2, 3]
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2,2,2], [3,1,2,3,3], [2,2,1,l,2],
....:                [2,3,l,1,2], [2,3,2,2,1]]).is_finite()]
```

```
....:
[2, 3]
sage: [l for l in range(2, 9) if
....:  CoxeterGroup([[1,3,2,2,2,2], [3,1,1,2,2,2], [2,1,1,3,1,2],
....:                [2,2,3,1,2,2], [2,2,1,2,1,3], [2,2,2,2,3,1]]).is_finite()]
....:
[2, 3]
```

`CoxeterMatrixGroup.order()`

Return the order of `self`.

If the Coxeter group is finite, this uses an iterator.

EXAMPLES:

```
sage: W = CoxeterGroup([[1,3],[3,1]])
sage: W.order()
6
sage: W = CoxeterGroup([[1,-1],[-1,1]])
sage: W.order()
+Infinity
```

`CoxeterMatrixGroup.simple_reflection(i)`

Return the simple reflection s_i.

INPUT:

- `i` – an element from the index set

EXAMPLES:

```
sage: W = CoxeterGroup(['A',3], implementation="reflection")
sage: W.simple_reflection(1)
[-1  1  0]
[ 0  1  0]
[ 0  0  1]
sage: W.simple_reflection(2)
[ 1  0  0]
[ 1 -1  1]
[ 0  0  1]
sage: W.simple_reflection(3)
[ 1  0  0]
[ 0  1  0]
[ 0  1 -1]
```

CHAPTER

FORTY

LINEAR GROUPS

EXAMPLES:

```
sage: GL(4,QQ)
General Linear Group of degree 4 over Rational Field
sage: GL(1,ZZ)
General Linear Group of degree 1 over Integer Ring
sage: GL(100,RR)
General Linear Group of degree 100 over Real Field with 53 bits of precision
sage: GL(3,GF(49,'a'))
General Linear Group of degree 3 over Finite Field in a of size 7^2

sage: SL(2, ZZ)
Special Linear Group of degree 2 over Integer Ring
sage: G = SL(2,GF(3)); G
Special Linear Group of degree 2 over Finite Field of size 3
sage: G.is_finite()
True
sage: G.conjugacy_class_representatives()
(
[1 0]  [0 2]  [0 1]  [2 0]  [0 2]  [0 1]  [0 2]
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
sage: G = SL(6,GF(5))
sage: G.gens()
(
[2 0 0 0 0 0]  [4 0 0 0 0 1]
[0 3 0 0 0 0]  [4 0 0 0 0 0]
[0 0 1 0 0 0]  [0 4 0 0 0 0]
[0 0 0 1 0 0]  [0 0 4 0 0 0]
[0 0 0 0 1 0]  [0 0 0 4 0 0]
[0 0 0 0 0 1], [0 0 0 0 4 0]
)
```

AUTHORS:

- William Stein: initial version
- David Joyner: degree, base_ring, random, order methods; examples
- David Joyner (2006-05): added center, more examples, renamed random attributes, bug fixes.
- William Stein (2006-12): total rewrite
- Volker Braun (2013-1) port to new Parent, libGAP, extreme refactoring.

REFERENCES:

- [KL] Peter Kleidman and Martin Liebeck. The subgroup structure of the finite classical groups. Cambridge University Press, 1990.
- [C] R. W. Carter. Simple groups of Lie type, volume 28 of Pure and Applied Mathematics. John Wiley and Sons, 1972.

`sage.groups.matrix_gps.linear.`**`GL`**`(`*n*, *R*, *var='a'*`)`

Return the general linear group.

The general linear group $GL(d, R)$ consists of all $dimesd$ matrices that are invertible over the ring R.

Note: This group is also available via `groups.matrix.GL()`.

INPUT:

- `n` – a positive integer.
- `R` – ring or an integer. If an integer is specified, the corresponding finite field is used.
- `var` – variable used to represent generator of the finite field, if needed.

EXAMPLES:

```
sage: G = GL(6,GF(5))
sage: G.order()
11064475422000000000000000
sage: G.base_ring()
Finite Field of size 5
sage: G.category()
Category of finite groups
sage: TestSuite(G).run()

sage: G = GL(6, QQ)
sage: G.category()
Category of groups
sage: TestSuite(G).run()
```

Here is the Cayley graph of (relatively small) finite General Linear Group:

```
sage: g = GL(2,3)
sage: d = g.cayley_graph(); d
Digraph on 48 vertices
sage: d.show(color_by_label=True, vertex_size=0.03, vertex_labels=False)
sage: d.show3d(color_by_label=True)

sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[2,0],[0,1]]), MS([[2,1],[2,0]])]
sage: G = MatrixGroup(gens)
sage: G.order()
48
sage: G.cardinality()
48
sage: H = GL(2,F)
sage: H.order()
48
sage: H == G
True
sage: H.gens() == G.gens()
True
sage: H.as_matrix_group() == H
True
```

```
sage: H.gens()
(
[2 0]  [2 1]
[0 1], [2 0]
)
```

TESTS:

```
sage: groups.matrix.GL(2, 3)
General Linear Group of degree 2 over Finite Field of size 3
```

class `sage.groups.matrix_gps.linear.`**`LinearMatrixGroup_gap`**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*)

Bases: `sage.groups.matrix_gps.named_group.NamedMatrixGroup_gap`, `sage.groups.matrix_gps.linear.LinearMatrixGroup_generic`

Base class for "named" matrix groups using LibGAP

INPUT:

- `degree` – integer. The degree (number of rows/columns of matrices).
- `base_ring` – rinrg. The base ring of the matrices.
- `special` – boolean. Whether the matrix group is special, that is, elements have determinant one.
- `latex_string` – string. The latex representation.
- `gap_command_string` – string. The GAP command to construct the matrix group.

EXAMPLES:

```
sage: G = GL(2, GF(3))
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_gap
sage: isinstance(G, NamedMatrixGroup_gap)
True
```

class `sage.groups.matrix_gps.linear.`**`LinearMatrixGroup_generic`**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*)

Bases: `sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic`

Base class for "named" matrix groups

INPUT:

- `degree` – integer. The degree (number of rows/columns of matrices).
- `base_ring` – rinrg. The base ring of the matrices.
- `special` – boolean. Whether the matrix group is special, that is, elements have determinant one.
- `latex_string` – string. The latex representation.

EXAMPLES:

```
sage: G = GL(2, QQ)
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_generic
sage: isinstance(G, NamedMatrixGroup_generic)
True
```

`sage.groups.matrix_gps.linear.`**`SL`**(*n*, *R*, *var='a'*)

Return the special linear group.

The special linear group $GL(d, R)$ consists of all $d \times d$ matrices that are invertible over the ring R with determinant one.

Note: This group is also available via `groups.matrix.SL()`.

INPUT:

- `n` – a positive integer.
- `R` – ring or an integer. If an integer is specified, the corresponding finite field is used.
- `var` – variable used to represent generator of the finite field, if needed.

EXAMPLES:

```
sage: SL(3, GF(2))
Special Linear Group of degree 3 over Finite Field of size 2
sage: G = SL(15, GF(7)); G
Special Linear Group of degree 15 over Finite Field of size 7
sage: G.category()
Category of finite groups
sage: G.order()
1956712595698146962015219062429586341124018007182049478916067369638713066737882363393519966
sage: len(G.gens())
2
sage: G = SL(2, ZZ); G
Special Linear Group of degree 2 over Integer Ring
sage: G.gens()
(
[ 0  1]  [1 1]
[-1  0], [0 1]
)
```

Next we compute generators for $\mathrm{SL}_3(\mathbf{Z})$

```
sage: G = SL(3,ZZ); G
Special Linear Group of degree 3 over Integer Ring
sage: G.gens()
(
[0 1 0]  [ 0  1  0]  [1 1 0]
[0 0 1]  [-1  0  0]  [0 1 0]
[1 0 0], [ 0  0  1], [0 0 1]
)
sage: TestSuite(G).run()
```

TESTS:

```
sage: groups.matrix.SL(2, 3)
Special Linear Group of degree 2 over Finite Field of size 3
```

CHAPTER

FORTYONE

ORTHOGONAL LINEAR GROUPS

The general orthogonal group $GO(n, R)$ consists of all $n \times n$ matrices over the ring R preserving an n-ary positive definite quadratic form. In cases where there are muliple non-isomorphic quadratic forms, additional data needs to be specified to disambiguate. The special orthogonal group is the normal subgroup of matrices of determinant one.

In characteristics different from 2, a quadratic form is equivalent to a bilinear symmetric form. Furthermore, over the real numbers a positive definite quadratic form is equivalent to the diagonal quadratic form, equivalent to the bilinear symmetric form defined by the identity matrix. Hence, the orthogonal group $GO(n, \mathbf{R})$ is the group of orthogonal matrices in the usual sense.

In the case of a finite field and if the degree n is even, then there are two inequivalent quadratic forms and a third parameter `e` must be specified to disambiguate these two possibilities. The index of $SO(e, d, q)$ in $GO(e, d, q)$ is 2 if q is odd, but $SO(e, d, q) = GO(e, d, q)$ if q is even.)

Warning: GAP and Sage use different notations:

- GAP notation: The optional `e` comes first, that is, `GO([e,] d, q)`, `SO([e,] d, q)`.
- Sage notation: The optional `e` comes last, the standard Python convention: `GO(d, GF(q), e=0)`, `SO(d, GF(q), e=0)`.

EXAMPLES:

```
sage: GO(3,7)
General Orthogonal Group of degree 3 over Finite Field of size 7

sage: G = SO( 4, GF(7), 1); G
Special Orthogonal Group of degree 4 and form parameter 1 over Finite Field of size 7
sage: G.random_element()   # random
[4 3 5 2]
[6 6 4 0]
[0 4 6 0]
[4 4 5 1]
```

TESTS:

```
sage: G = GO(3, GF(5))
sage: latex(G)
\text{GO}_{3}(\Bold{F}_{5})
sage: G = SO(3, GF(5))
sage: latex(G)
\text{SO}_{3}(\Bold{F}_{5})
sage: G = SO(4, GF(5), 1)
sage: latex(G)
\text{SO}_{4}(\Bold{F}_{5}, +)
```

AUTHORS:

- David Joyner (2006-03): initial version
- David Joyner (2006-05): added examples, _latex_, __str__, gens, as_matrix_group
- William Stein (2006-12-09): rewrite
- Volker Braun (2013-1) port to new Parent, libGAP, extreme refactoring.

sage.groups.matrix_gps.orthogonal.**GO**(*n*, *R*, *e=0*, *var='a'*)

Return the general orthogonal group.

The general orthogonal group $GO(n, R)$ consists of all $nimesn$ matrices over the ring R preserving an n-ary positive definite quadratic form. In cases where there are muliple non-isomorphic quadratic forms, additional data needs to be specified to disambiguate.

In the case of a finite field and if the degree n is even, then there are two inequivalent quadratic forms and a third parameter `e` must be specified to disambiguate these two possibilities.

Note: This group is also available via `groups.matrix.GO()`.

INPUT:

- `n` – integer. The degree.
- `R` – ring or an integer. If an integer is specified, the corresponding finite field is used.
- `e` – `+1` or `-1`, and ignored by default. Only relevant for finite fields and if the degree is even. A parameter that distinguishes inequivalent invariant forms.

OUTPUT:

The general orthogonal group of given degree, base ring, and choice of invariant form.

EXAMPLES:

```
sage: GO( 3, GF(7))
General Orthogonal Group of degree 3 over Finite Field of size 7
sage: GO( 3, GF(7)).order()
672
sage: GO( 3, GF(7)).gens()
(
[3 0 0]  [0 1 0]
[0 5 0]  [1 6 6]
[0 0 1], [0 2 1]
)
```

TESTS:

```
sage: groups.matrix.GO(2, 3, e=-1)
General Orthogonal Group of degree 2 and form parameter -1 over Finite Field of size 3
```

class sage.groups.matrix_gps.orthogonal.**OrthogonalMatrixGroup_gap**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*)

Bases: `sage.groups.matrix_gps.orthogonal.OrthogonalMatrixGroup_generic`, `sage.groups.matrix_gps.named_group.NamedMatrixGroup_gap`

Base class for "named" matrix groups using LibGAP

INPUT:

- `degree` – integer. The degree (number of rows/columns of matrices).
- `base_ring` – rinrg. The base ring of the matrices.
- `special` – boolean. Whether the matrix group is special, that is, elements have determinant one.
- `latex_string` – string. The latex representation.
- `gap_command_string` – string. The GAP command to construct the matrix group.

EXAMPLES:

```
sage: G = GL(2, GF(3))
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_gap
sage: isinstance(G, NamedMatrixGroup_gap)
True
```

invariant_bilinear_form()

Return the symmetric bilinear form preserved by the orthogonal group.

OUTPUT:

A matrix M such that, for every group element g, the identity $gmg^T = m$ holds. In characteristic different from two, this uniquely determines the orthogonal group.

EXAMPLES:

```
sage: G = GO(4, GF(7), -1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]

sage: G = GO(4, GF(7), +1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 6 0]
[0 0 0 2]

sage: G = GO(4, QQ)
sage: G.invariant_bilinear_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: G = SO(4, GF(7), -1)
sage: G.invariant_bilinear_form()
[0 1 0 0]
[1 0 0 0]
[0 0 2 0]
[0 0 0 2]
```

invariant_quadratic_form()

Return the quadratic form preserved by the orthogonal group.

OUTPUT:

The matrix Q defining "orthogonal" as follows. The matrix determines a quadratic form q on the natural vector space V, on which G acts, by $q(v) = vQv^t$. A matrix $M' is an element of the orthogonal group if `q(v) = q(vM)$ for all $v \in V$.

EXAMPLES:

```
sage: G = GO(4, GF(7), -1)
sage: G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 1 0]
[0 0 0 1]

sage: G = GO(4, GF(7), +1)
sage: G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 3 0]
[0 0 0 1]

sage: G = GO(4, QQ)
sage: G.invariant_quadratic_form()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: G = SO(4, GF(7), -1)
sage: G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 1 0]
[0 0 0 1]
```

class `sage.groups.matrix_gps.orthogonal.`**`OrthogonalMatrixGroup_generic`**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*)

Bases: `sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic`

Base class for "named" matrix groups

INPUT:

- `degree` – integer. The degree (number of rows/columns of matrices).
- `base_ring` – rinrg. The base ring of the matrices.
- `special` – boolean. Whether the matrix group is special, that is, elements have determinant one.
- `latex_string` – string. The latex representation.

EXAMPLES:

```
sage: G = GL(2, QQ)
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_generic
sage: isinstance(G, NamedMatrixGroup_generic)
True
```

invariant_bilinear_form()

Return the symmetric bilinear form preserved by the orthogonal group.

OUTPUT:

A matrix.

EXAMPLES:

```
sage: GO(2,3,+1).invariant_bilinear_form()
[0 1]
[1 0]
sage: GO(2,3,-1).invariant_bilinear_form()
[2 1]
[1 1]
```

invariant_quadratic_form()

Return the quadratic form preserved by the orthogonal group.

OUTPUT:

A matrix.

EXAMPLES:

```
sage: GO(2,3,+1).invariant_quadratic_form()
[0 1]
[0 0]
sage: GO(2,3,-1).invariant_quadratic_form()
[1 1]
[0 2]
```

sage.groups.matrix_gps.orthogonal.**SO**(*n*, *R*, *e=None*, *var='a'*)

Return the special orthogonal group.

The special orthogonal group $GO(n, R)$ consists of all $nimesn$ matrices with determint one over the ring R preserving an n-ary positive definite quadratic form. In cases where there are muliple non-isomorphic quadratic forms, additional data needs to be specified to disambiguate.

Note: This group is also available via `groups.matrix.SO()`.

INPUT:

- `n` – integer. The degree.
- `R` – ring or an integer. If an integer is specified, the corresponding finite field is used.
- `e` – $+1$ or -1, and ignored by default. Only relevant for finite fields and if the degree is even. A parameter that distinguishes inequivalent invariant forms.

OUTPUT:

The special orthogonal group of given degree, base ring, and choice of invariant form.

EXAMPLES:

```
sage: G = SO(3,GF(5))
sage: G
Special Orthogonal Group of degree 3 over Finite Field of size 5

sage: G = SO(3,GF(5))
sage: G.gens()
(
[2 0 0]  [3 2 3]  [1 4 4]
```

```
[0 3 0]   [0 2 0]   [4 0 0]
[0 0 1], [0 3 1], [2 0 4]
)
sage: G = SO(3,GF(5))
sage: G.as_matrix_group()
Matrix group over Finite Field of size 5 with 3 generators (
[2 0 0]   [3 2 3]   [1 4 4]
[0 3 0]   [0 2 0]   [4 0 0]
[0 0 1], [0 3 1], [2 0 4]
)
```

TESTS:

```
sage: groups.matrix.SO(2, 3, e=1)
Special Orthogonal Group of degree 2 and form parameter 1 over Finite Field of size 3
```

sage.groups.matrix_gps.orthogonal.**normalize_args_e**(*degree*, *ring*, *e*)

Normalize the arguments that relate the choice of quadratic form for special orthogonal groups over finite fields.

INPUT:

- `degree` – integer. The degree of the affine group, that is, the dimension of the affine space the group is acting on.
- `ring` – a ring. The base ring of the affine space.
- `e` – integer, one of $+1$, 0, -1. Only relevant for finite fields and if the degree is even. A parameter that distinguishes inequivalent invariant forms.

OUTPUT:

The integer `e` with values required by GAP.

TESTS:

```
sage: from sage.groups.matrix_gps.orthogonal import normalize_args_e
sage: normalize_args_e(2, GF(3), +1)
1
sage: normalize_args_e(3, GF(3), 0)
0
sage: normalize_args_e(3, GF(3), +1)
0
sage: normalize_args_e(2, GF(3), 0)
Traceback (most recent call last):
...
ValueError: must have e=-1 or e=1 for even degree
```

CHAPTER

FORTYTWO

SYMPLECTIC LINEAR GROUPS

EXAMPLES:

```
sage: G = Sp(4,GF(7));  G
Symplectic Group of degree 4 over Finite Field of size 7
sage: g = prod(G.gens());  g
[3 0 3 0]
[1 0 0 0]
[0 1 0 1]
[0 2 0 0]
sage: m = g.matrix()
sage: m * G.invariant_form() * m.transpose() == G.invariant_form()
True
sage: G.order()
276595200
```

AUTHORS:

- David Joyner (2006-03): initial version, modified from special_linear (by W. Stein)
- Volker Braun (2013-1) port to new Parent, libGAP, extreme refactoring.

`sage.groups.matrix_gps.symplectic.`**`Sp`**(*n*, *R*, *var='a'*)

Return the symplectic group.

The special linear group $GL(d, R)$ consists of all $d \times d$ matrices that are invertible over the ring R with determinant one.

Note: This group is also available via `groups.matrix.Sp()`.

INPUT:

- `n` – a positive integer.
- `R` – ring or an integer. If an integer is specified, the corresponding finite field is used.
- `var` – variable used to represent generator of the finite field, if needed.

EXAMPLES:

```
sage: Sp(4, 5)
Symplectic Group of degree 4 over Finite Field of size 5

sage: Sp(4, IntegerModRing(15))
Symplectic Group of degree 4 over Ring of integers modulo 15

sage: Sp(3, GF(7))
Traceback (most recent call last):
```

```
...
ValueError: the degree must be even
```

TESTS:

```
sage: groups.matrix.Sp(2, 3)
Symplectic Group of degree 2 over Finite Field of size 3

sage: G = Sp(4,5)
sage: TestSuite(G).run()
```

class sage.groups.matrix_gps.symplectic.**SymplecticMatrixGroup_gap**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*)

Bases: sage.groups.matrix_gps.symplectic.SymplecticMatrixGroup_generic, sage.groups.matrix_gps.named_group.NamedMatrixGroup_gap

Symplectic group in GAP

EXAMPLES:

```
sage: Sp(2,4)
Symplectic Group of degree 2 over Finite Field in a of size 2^2

sage: latex(Sp(4,5))
\text{Sp}_{4}(\Bold{F}_{5})
```

invariant_form()

Return the quadratic form preserved by the orthogonal group.

OUTPUT:

A matrix.

EXAMPLES:

```
sage: Sp(4, GF(3)).invariant_form()
[0 0 0 1]
[0 0 1 0]
[0 2 0 0]
[2 0 0 0]
```

class sage.groups.matrix_gps.symplectic.**SymplecticMatrixGroup_generic**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*)

Bases: sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic

Base class for "named" matrix groups

INPUT:

- degree – integer. The degree (number of rows/columns of matrices).
- base_ring – rinrg. The base ring of the matrices.
- special – boolean. Whether the matrix group is special, that is, elements have determinant one.

- `latex_string` – string. The latex representation.

EXAMPLES:

```
sage: G = GL(2, QQ)
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_generic
sage: isinstance(G, NamedMatrixGroup_generic)
True
```

invariant_form()

Return the quadratic form preserved by the orthogonal group.

OUTPUT:

A matrix.

EXAMPLES:

```
sage: Sp(4, QQ).invariant_form()
[0 0 0 1]
[0 0 1 0]
[0 1 0 0]
[1 0 0 0]
```

CHAPTER FORTYTHREE

UNITARY GROUPS $GU(N, Q)$ AND $SU(N, Q)$

These are $n \times n$ unitary matrices with entries in $GF(q^2)$.

EXAMPLES:

```
sage: G = SU(3,5)
sage: G.order()
378000
sage: G
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: G.gens()
(
[      a       0       0]  [4*a   4   1]
[      0 2*a + 2       0]  [  4   4   0]
[      0       0     3*a], [  1   0   0]
)
sage: G.base_ring()
Finite Field in a of size 5^2
```

AUTHORS:

- David Joyner (2006-03): initial version, modified from special_linear (by W. Stein)
- David Joyner (2006-05): minor additions (examples, _latex_, __str__, gens)
- William Stein (2006-12): rewrite
- Volker Braun (2013-1) port to new Parent, libGAP, extreme refactoring.

`sage.groups.matrix_gps.unitary.`**`GU`**`(`*n*, *R*, *var='a'*`)`

Return the general unitary group.

The general unitary group $GU(d, R)$ consists of all $d \times d$ matrices that preserve a nondegenerate sequilinear form over the ring R.

Note: For a finite field the matrices that preserve a sesquilinear form over F_q live over F_{q^2}. So `GU(n,q)` for integer `q` constructs the matrix group over the base ring `GF(q^2)`.

Note: This group is also available via `groups.matrix.GU()`.

INPUT:

- `n` – a positive integer.
- `R` – ring or an integer. If an integer is specified, the corresponding finite field is used.
- `var` – variable used to represent generator of the finite field, if needed.

OUTPUT:

Return the general unitary group.

EXAMPLES:

```
sage: G = GU(3, 7); G
General Unitary Group of degree 3 over Finite Field in a of size 7^2
sage: G.gens()
(
[  a   0   0]  [6*a   6   1]
[  0   1   0]  [  6   6   0]
[  0   0 5*a], [  1   0   0]
)
sage: GU(2,QQ)
General Unitary Group of degree 2 over Rational Field

sage: G = GU(3, 5, var='beta')
sage: G.base_ring()
Finite Field in beta of size 5^2
sage: G.gens()
(
[  beta      0      0]  [4*beta      4      1]
[     0      1      0]  [     4      4      0]
[     0      0 3*beta], [     1      0      0]
)
```

TESTS:

```
sage: groups.matrix.GU(2, 3)
General Unitary Group of degree 2 over Finite Field in a of size 3^2
```

`sage.groups.matrix_gps.unitary.`**`SU`**(*n*, *R*, *var='a'*)

The special unitary group $SU(d, R)$ consists of all $dimesd$ matrices that preserve a nondegenerate sequilinear form over the ring R and have determinant one.

Note: For a finite field the matrices that preserve a sesquilinear form over F_q live over F_{q^2}. So `SU(n,q)` for integer `q` constructs the matrix group over the base ring `GF(q^2)`.

Note: This group is also available via `groups.matrix.SU()`.

INPUT:

- `n` – a positive integer.
- `R` – ring or an integer. If an integer is specified, the corresponding finite field is used.
- `var` – variable used to represent generator of the finite field, if needed.

OUTPUT:

Return the special unitary group.

EXAMPLES:

```
sage: SU(3,5)
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: SU(3, GF(5))
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: SU(3,QQ)
Special Unitary Group of degree 3 over Rational Field
```

TESTS:

```
sage: groups.matrix.SU(2, 3)
Special Unitary Group of degree 2 over Finite Field in a of size 3^2
```

class `sage.groups.matrix_gps.unitary.`**`UnitaryMatrixGroup_gap`**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*)

Bases: `sage.groups.matrix_gps.unitary.UnitaryMatrixGroup_generic`, `sage.groups.matrix_gps.named_group.NamedMatrixGroup_gap`

Base class for "named" matrix groups using LibGAP

INPUT:

- `degree` – integer. The degree (number of rows/columns of matrices).
- `base_ring` – rinrg. The base ring of the matrices.
- `special` – boolean. Whether the matrix group is special, that is, elements have determinant one.
- `latex_string` – string. The latex representation.
- `gap_command_string` – string. The GAP command to construct the matrix group.

EXAMPLES:

```
sage: G = GL(2, GF(3))
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_gap
sage: isinstance(G, NamedMatrixGroup_gap)
True
```

class `sage.groups.matrix_gps.unitary.`**`UnitaryMatrixGroup_generic`**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*)

Bases: `sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic`

General Unitary Group over arbitrary rings.

EXAMPLES:

```
sage: G = GU(3, GF(7)); G
General Unitary Group of degree 3 over Finite Field in a of size 7^2
sage: latex(G)
\text{GU}_{3}(\Bold{F}_{7^{2}})

sage: G = SU(3, GF(5));  G
Special Unitary Group of degree 3 over Finite Field in a of size 5^2
sage: latex(G)
\text{SU}_{3}(\Bold{F}_{5^{2}})
```

`sage.groups.matrix_gps.unitary.`**`finite_field_sqrt`**(*ring*)

Helper function.

INPUT:

A ring.

OUTPUT:

Integer q such that `ring` is the finite field with q^2 elements.

EXAMPLES:

```
sage: from sage.groups.matrix_gps.unitary import finite_field_sqrt
sage: finite_field_sqrt(GF(4, 'a'))
2
```

CHAPTER

FORTYFOUR

AFFINE GROUPS

AUTHORS:

- Volker Braun: initial version

class `sage.groups.affine_gps.affine_group.`**`AffineGroup`**(*degree*, *ring*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.groups.group.Group`

An affine group.

The affine group $\mathrm{Aff}(A)$ (or general affine group) of an affine space A is the group of all invertible affine transformations from the space into itself.

If we let A_V be the affine space of a vector space V (essentially, forgetting what is the origin) then the affine group $\mathrm{Aff}(A_V)$ is the group generated by the general linear group $GL(V)$ together with the translations. Recall that the group of translations acting on A_V is just V itself. The general linear and translation subgroups do not quite commute, and in fact generate the semidirect product

$$\mathrm{Aff}(A_V) = GL(V) \ltimes V.$$

As such, the group elements can be represented by pairs (A, b) of a matrix and a vector. This pair then represents the transformation

$$x \mapsto Ax + b.$$

We can also represent affine transformations as linear transformations by considering $\dim(V) + 1$ dimensonal space. We take the affine transformation (A, b) to

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

and lifting $x = (x_1, \ldots, x_n)$ to $(x_1, \ldots, x_n, 1)$. Here the $(n + 1)$-th component is always 1, so the linear representations acts on the affine hyperplane $x_{n+1} = 1$ as affine transformations which can be seen directly from the matrix multiplication.

INPUT:

Something that defines an affine space. For example

- An affine space itself:
  - `A` – affine space
- A vector space:
  - `V` – a vector space
- Degree and base ring:

- degree – An integer. The degree of the affine group, that is, the dimension of the affine space the group is acting on.
- ring – A ring or an integer. The base ring of the affine space. If an integer is given, it must be a prime power and the corresponding finite field is constructed.
- var – (Defalut: 'a') Keyword argument to specify the finite field generator name in the case where ring is a prime power.

EXAMPLES:

```
sage: F = AffineGroup(3, QQ); F
Affine Group of degree 3 over Rational Field
sage: F(matrix(QQ,[[1,2,3],[4,5,6],[7,8,0]]), vector(QQ,[10,11,12]))
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
sage: F([[1,2,3],[4,5,6],[7,8,0]], [10,11,12])
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
sage: F([1,2,3,4,5,6,7,8,0], [10,11,12])
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]
```

Instead of specifying the complete matrix/vector information, you can also create special group elements:

```
sage: F.linear([1,2,3,4,5,6,7,8,0])
      [1 2 3]     [0]
x |-> [4 5 6] x + [0]
      [7 8 0]     [0]
sage: F.translation([1,2,3])
      [1 0 0]     [1]
x |-> [0 1 0] x + [2]
      [0 0 1]     [3]
```

Some additional ways to create affine groups:

```
sage: A = AffineSpace(2, GF(4,'a'));  A
Affine Space of dimension 2 over Finite Field in a of size 2^2
sage: G = AffineGroup(A); G
Affine Group of degree 2 over Finite Field in a of size 2^2
sage: G is AffineGroup(2,4) # shorthand
True

sage: V = ZZ^3;  V
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: AffineGroup(V)
Affine Group of degree 3 over Integer Ring
```

REFERENCES:

- Wikipedia article Affine_group

Element
alias of AffineGroupElement

degree()
Return the dimension of the affine space.

OUTPUT:

An integer.

EXAMPLES:

```
sage: G = AffineGroup(6, GF(5))
sage: g = G.an_element()
sage: G.degree()
6
sage: G.degree() == g.A().nrows() == g.A().ncols() == g.b().degree()
True
```

linear(*A*)

Construct the general linear transformation by A.

INPUT:

- A – anything that determines a matrix

OUTPUT:

The affine group element $x \mapsto Ax$.

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.linear([1,2,3,4,5,6,7,8,0])
      [1 2 3]     [0]
x |-> [4 0 1] x + [0]
      [2 3 0]     [0]
```

linear_space()

Return the space of the affine transformations represented as linear transformations.

We can represent affine transformations $Ax + b$ as linear transformations by

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

and lifting $x = (x_1, \ldots, x_n)$ to $(x_1, \ldots, x_n, 1)$.

See also:

- sage.groups.affine_gps.group_element.AffineGroupElement.matrix()

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.linear_space()
Full MatrixSpace of 4 by 4 dense matrices over Finite Field of size 5
```

matrix_space()

Return the space of matrices representing the general linear transformations.

OUTPUT:

The parent of the matrices A defining the affine group element $Ax + b$.

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.matrix_space()
Full MatrixSpace of 3 by 3 dense matrices over Finite Field of size 5
```

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: G = AffineGroup(4, GF(3))
sage: G.random_element()  # random
      [2 0 1 2]     [1]
      [2 1 1 2]     [2]
x |-> [1 0 2 2] x + [2]
      [1 1 1 1]     [2]
sage: G.random_element() in G
True
```

reflection(v)

Construct the Householder reflection.

A Householder reflection (transformation) is the affine transformation corresponding to an elementary reflection at the hyperplane perpendicular to v.

INPUT:

- v – a vector, or something that determines a vector.

OUTPUT:

The affine group element that is just the Householder transformation (a.k.a. Householder reflection, elementary reflection) at the hyperplane perpendicular to v.

EXAMPLES:

```
sage: G = AffineGroup(3, QQ)
sage: G.reflection([1,0,0])
      [-1  0  0]     [0]
x |-> [ 0  1  0] x + [0]
      [ 0  0  1]     [0]
sage: G.reflection([3,4,-5])
      [ 16/25 -12/25    3/5]     [0]
x |-> [-12/25   9/25    4/5] x + [0]
      [   3/5    4/5      0]     [0]
```

translation(b)

Construct the translation by b.

INPUT:

- b – anything that determines a vector

OUTPUT:

The affine group element $x \mapsto x + b$.

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.translation([1,4,8])
      [1 0 0]     [1]
x |-> [0 1 0] x + [4]
      [0 0 1]     [3]
```

vector_space()

Return the vector space of the underlying affine space.

EXAMPLES:

```
sage: G = AffineGroup(3, GF(5))
sage: G.vector_space()
Vector space of dimension 3 over Finite Field of size 5
```

CHAPTER

FORTYFIVE

EUCLIDEAN GROUPS

AUTHORS:

- Volker Braun: initial version

class `sage.groups.affine_gps.euclidean_group.`**`EuclideanGroup`**(*degree*, *ring*)

Bases: `sage.groups.affine_gps.affine_group.AffineGroup`

A Euclidean group.

The Euclidean group $E(A)$ (or general affine group) of an affine space A is the group of all invertible affine transformations from the space into itself preserving the Euclidean metric.

If we let A_V be the affine space of a vector space V (essentially, forgetting what is the origin) then the Euclidean group $E(A_V)$ is the group generated by the general linear group $SO(V)$ together with the translations. Recall that the group of translations acting on A_V is just V itself. The general linear and translation subgroups do not quite commute, and in fact generate the semidirect product

$$E(A_V) = SO(V) \ltimes V.$$

As such, the group elements can be represented by pairs (A, b) of a matrix and a vector. This pair then represents the transformation

$$x \mapsto Ax + b.$$

We can also represent this as a linear transformation in $\dim(V) + 1$ dimensional space as

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

and lifting $x = (x_1, \ldots, x_n)$ to $(x_1, \ldots, x_n, 1)$.

See also:

- `AffineGroup`

INPUT:

Something that defines an affine space. For example

- An affine space itself:
  - `A` – affine space
- A vector space:
  - `V` – a vector space
- Degree and base ring:

- `degree` – An integer. The degree of the affine group, that is, the dimension of the affine space the group is acting on.
- `ring` – A ring or an integer. The base ring of the affine space. If an integer is given, it must be a prime power and the corresponding finite field is constructed.
- `var` – (Defalut: `'a'`) Keyword argument to specify the finite field generator name in the case where `ring` is a prime power.

EXAMPLES:

```
sage: E3 = EuclideanGroup(3, QQ); E3
Euclidean Group of degree 3 over Rational Field
sage: E3(matrix(QQ,[(6/7, -2/7, 3/7), (-2/7, 3/7, 6/7), (3/7, 6/7, -2/7)]), vector(QQ,[10,11,12]
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
sage: E3([[6/7, -2/7, 3/7], [-2/7, 3/7, 6/7], [3/7, 6/7, -2/7]], [10,11,12])
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
sage: E3([6/7, -2/7, 3/7, -2/7, 3/7, 6/7, 3/7, 6/7, -2/7], [10,11,12])
      [ 6/7 -2/7  3/7]     [10]
x |-> [-2/7  3/7  6/7] x + [11]
      [ 3/7  6/7 -2/7]     [12]
```

Instead of specifying the complete matrix/vector information, you can also create special group elements:

```
sage: E3.linear([6/7, -2/7, 3/7, -2/7, 3/7, 6/7, 3/7, 6/7, -2/7])
      [ 6/7 -2/7  3/7]     [0]
x |-> [-2/7  3/7  6/7] x + [0]
      [ 3/7  6/7 -2/7]     [0]
sage: E3.reflection([4,5,6])
      [ 45/77 -40/77 -48/77]     [0]
x |-> [-40/77  27/77 -60/77] x + [0]
      [-48/77 -60/77   5/77]     [0]
sage: E3.translation([1,2,3])
      [1 0 0]     [1]
x |-> [0 1 0] x + [2]
      [0 0 1]     [3]
```

Some additional ways to create Euclidean groups:

```
sage: A = AffineSpace(2, GF(4,'a'));  A
Affine Space of dimension 2 over Finite Field in a of size 2^2
sage: G = EuclideanGroup(A); G
Euclidean Group of degree 2 over Finite Field in a of size 2^2
sage: G is EuclideanGroup(2,4) # shorthand
True

sage: V = ZZ^3;  V
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: EuclideanGroup(V)
Euclidean Group of degree 3 over Integer Ring

sage: EuclideanGroup(2, QQ)
Euclidean Group of degree 2 over Rational Field
```

TESTS:

```
sage: E6 = EuclideanGroup(6, QQ)
sage: E6 is E6
True
sage: V = QQ^6
sage: E6 is EuclideanGroup(V)
True
sage: G = EuclideanGroup(2, GF(5)); G
Euclidean Group of degree 2 over Finite Field of size 5
sage: TestSuite(G).run()
```

REFERENCES:

- Wikipedia article Euclidean_group

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: G = EuclideanGroup(4, GF(3))
sage: G.random_element()  # random
      [2 1 2 1]     [1]
      [1 2 2 1]     [0]
x |-> [2 2 2 2] x + [1]
      [1 1 2 2]     [2]
sage: G.random_element() in G
True
```

TESTS:

```
sage: G.random_element().A().is_unitary()
True
```

CHAPTER

FORTYSIX

ELEMENTS OF AFFINE GROUPS

The class in this module is used to represent the elements of `AffineGroup()` and its subgroups.

EXAMPLES:

```
sage: F = AffineGroup(3, QQ)
sage: F([1,2,3,4,5,6,7,8,0], [10,11,12])
      [1 2 3]     [10]
x |-> [4 5 6] x + [11]
      [7 8 0]     [12]

sage: G = AffineGroup(2, ZZ)
sage: g = G([[1,1],[0,1]], [1,0])
sage: h = G([[1,2],[0,1]], [0,1])
sage: g*h
      [1 3]     [2]
x |-> [0 1] x + [1]
sage: h*g
      [1 3]     [1]
x |-> [0 1] x + [1]
sage: g*h != h*g
True
```

AUTHORS:

- Volker Braun

class sage.groups.affine_gps.group_element.**AffineGroupElement**(*parent*, *A*, *b=0*, *convert=True*, *check=True*)

Bases: `sage.groups.matrix_gps.group_element.MatrixGroupElement_base`

An affine group element.

INPUT:

- `A` – an invertible matrix, or something defining a matrix if `convert==True`.
- `b`– a vector, or something defining a vector if `convert==True` (default: `0`, defining the zero vector).
- `parent` – the parent affine group.
- `convert` - bool (default: `True`). Whether to convert `A` into the correct matrix space and `b` into the correct vector space.
- **`check` - bool (default: `True`). Whether to do some** checks or just accept the input as valid.

As a special case, `A` can be a matrix obtained from `matrix()`, that is, one row and one column larger. In that case, the group element defining that matrix is reconstructed.

OUTPUT:

The affine group element $x \mapsto Ax + b$

EXAMPLES:

```
sage: G = AffineGroup(2, GF(3))
sage: g = G.random_element()
sage: type(g)
<class 'sage.groups.affine_gps.group_element.AffineGroup_with_category.element_class'>
sage: G(g.matrix()) == g
True
sage: G(2)
      [2 0]     [0]
x |-> [0 2] x + [0]
```

A()

Return the general linear part of an affine group element.

OUTPUT:

The matrix A of the affine group element $Ax + b$.

EXAMPLES:

```
sage: G = AffineGroup(3, QQ)
sage: g = G([1,2,3,4,5,6,7,8,0], [10,11,12])
sage: g.A()
[1 2 3]
[4 5 6]
[7 8 0]
```

b()

Return the translation part of an affine group element.

OUTPUT:

The vector b of the affine group element $Ax + b$.

EXAMPLES:

```
sage: G = AffineGroup(3, QQ)
sage: g = G([1,2,3,4,5,6,7,8,0], [10,11,12])
sage: g.b()
(10, 11, 12)
```

inverse()

Return the inverse group element.

OUTPUT:

Another affine group element.

EXAMPLES:

```
sage: G = AffineGroup(2, GF(3))
sage: g = G([1,2,3,4], [5,6])
sage: g
      [1 2]     [2]
x |-> [0 1] x + [0]
sage: ~g
      [1 1]     [1]
x |-> [0 1] x + [0]
sage: g * g.inverse()
```

```
       [1 0]      [0]
x |-> [0 1] x + [0]
sage: g * g.inverse() == g.inverse() * g == G(1)
True
```

matrix()

Return the standard matrix representation of `self`.

See also:

- `AffineGroup.linear_space()`

EXAMPLES:

```
sage: G = AffineGroup(3, GF(7))
sage: g = G([1,2,3,4,5,6,7,8,0], [10,11,12])
sage: g
       [1 2 3]      [3]
x |-> [4 5 6] x + [4]
       [0 1 0]      [5]
sage: g.matrix()
[1 2 3|3]
[4 5 6|4]
[0 1 0|5]
[-----+-]
[0 0 0|1]
sage: parent(g.matrix())
Full MatrixSpace of 4 by 4 dense matrices over Finite Field of size 7
sage: g.matrix() == matrix(g)
True
```

Composition of affine group elements equals multiplication of the matrices:

```
sage: g1 = G.random_element()
sage: g2 = G.random_element()
sage: g1.matrix() * g2.matrix() == (g1*g2).matrix()
True
```

CHAPTER

FORTYSEVEN

MISCELLANEOUS GROUPS

This is a collection of groups that may not fit into some of the other infinite families described elsewhere.

CHAPTER

FORTYEIGHT

SEMIMONOMIAL TRANSFORMATION GROUP

The semimonomial transformation group of degree n over a ring R is the semidirect product of the monomial transformation group of degree n (also known as the complete monomial group over the group of units $R^{\times}$ of R) and the group of ring automorphisms.

The multiplication of two elements $(\phi, \pi, \alpha)(\psi, \sigma, \beta)$ with

- $\phi, \psi \in {R^{\times}}^n$
- $\pi, \sigma \in S_n$ (with the multiplication $\pi\sigma$ done from left to right (like in GAP) – that is, $(\pi\sigma)(i) = \sigma(\pi(i))$ for all i.)
- $\alpha, \beta \in Aut(R)$

is defined by

$$(\phi, \pi, \alpha)(\psi, \sigma, \beta) = (\phi \cdot \psi^{\pi,\alpha}, \pi\sigma, \alpha \circ \beta)$$

where $\psi^{\pi,\alpha} = (\alpha(\psi_{\pi(1)-1}), \ldots, \alpha(\psi_{\pi(n)-1}))$ and the multiplication of vectors is defined elementwisely. (The indexing of vectors is 0-based here, so $\psi = (\psi_0, \psi_1, \ldots, \psi_{n-1})$.)

Todo

Up to now, this group is only implemented for finite fields because of the limited support of automorphisms for arbitrary rings.

AUTHORS:

- Thomas Feulner (2012-11-15): initial version

EXAMPLES:

```
sage: S = SemimonomialTransformationGroup(GF(4, 'a'), 4)
sage: G = S.gens()
sage: G[0]*G[1]
((a, 1, 1, 1); (1,2,3,4), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a)
```

TESTS:

```
sage: TestSuite(S).run()
sage: TestSuite(S.an_element()).run()
```

class `sage.groups.semimonomial_transformations.semimonomial_transformation_group.Semimonomial`

Bases: `sage.categories.action.Action`

The action of SemimonomialTransformationGroup on matrices over the same ring whose number of columns is equal to the degree. See SemimonomialActionVec for the definition of the action on the row vectors of such a matrix.

class `sage.groups.semimonomial_transformations.semimonomial_transformation_group.Semimonomial`

Bases: `sage.categories.action.Action`

The natural action of the semimonomial group on vectors.

The action is defined by: $(\phi, \pi, \alpha) * (v_0, \ldots, v_{n-1}) := (\alpha(v_{\pi(1)-1}) \cdot \phi_0^{-1}, \ldots, \alpha(v_{\pi(n)-1}) \cdot \phi_{n-1}^{-1})$. (The indexing of vectors is 0-based here, so $\psi = (\psi_0, \psi_1, \ldots, \psi_{n-1})$.)

class `sage.groups.semimonomial_transformations.semimonomial_transformation_group.Semimonomial`

Bases: `sage.groups.group.FiniteGroup`, `sage.structure.unique_representation.UniqueRepresent`

A semimonomial transformation group over a ring.

The semimonomial transformation group of degree n over a ring R is the semidirect product of the monomial transformation group of degree n (also known as the complete monomial group over the group of units $R^\times$ of R) and the group of ring automorphisms.

The multiplication of two elements $(\phi, \pi, \alpha)(\psi, \sigma, \beta)$ with

- $\phi, \psi \in {R^\times}^n$
- $\pi, \sigma \in S_n$ (with the multiplication $\pi\sigma$ done from left to right (like in GAP) – that is, $(\pi\sigma)(i) = \sigma(\pi(i))$ for all i.)
- $\alpha, \beta \in Aut(R)$

is defined by

$$(\phi, \pi, \alpha)(\psi, \sigma, \beta) = (\phi \cdot \psi^{\pi,\alpha}, \pi\sigma, \alpha \circ \beta)$$

where $\psi^{\pi,\alpha} = (\alpha(\psi_{\pi(1)-1}), \ldots, \alpha(\psi_{\pi(n)-1}))$ and the multiplication of vectors is defined elementwisely. (The indexing of vectors is 0-based here, so $\psi = (\psi_0, \psi_1, \ldots, \psi_{n-1})$.)

Todo

Up to now, this group is only implemented for finite fields because of the limited support of automorphisms for arbitrary rings.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: S = SemimonomialTransformationGroup(F, 4)
sage: g = S(v = [2, a, 1, 2])
sage: h = S(perm = Permutation('(1,2,3,4)'), autom=F.hom([a**3]))
sage: g*h
((2, a, 1, 2); (1,2,3,4), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a +
sage: h*g
((2*a + 1, 1, 2, 2); (1,2,3,4), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |-->
sage: S(g)
((2, a, 1, 2); (), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> a)
sage: S(1)
((1, 1, 1, 1); (), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> a)
```

Element
alias of `SemimonomialTransformation`

base_ring()

Returns the underlying ring of `self`.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 3).base_ring() is F
True
```

degree()

Returns the degree of `self`.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 3).degree()
3
```

gens()

Return a tuple of generators of `self`.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 3).gens()
[((a, 1, 1); (), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a), ((1, 1, 1); (1,2,3), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a), ((1, 1, 1); (1,2), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a), ((1, 1, 1); (), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a + 1)]
```

order()

Returns the number of elements of `self`.

EXAMPLES:

```
sage: F.<a> = GF(4)
sage: SemimonomialTransformationGroup(F, 5).order() == (4-1)**5 * factorial(5) * 2
True
```

CHAPTER

FORTYNINE

ELEMENTS OF A SEMIMONOMIAL TRANSFORMATION GROUP.

The semimonomial transformation group of degree n over a ring R is the semidirect product of the monomial transformation group of degree n (also known as the complete monomial group over the group of units $R^{\times}$ of R) and the group of ring automorphisms.

The multiplication of two elements $(\phi, \pi, \alpha)(\psi, \sigma, \beta)$ with

- $\phi, \psi \in {R^{\times}}^n$
- $\pi, \sigma \in S_n$ (with the multiplication $\pi\sigma$ done from left to right (like in GAP) – that is, $(\pi\sigma)(i) = \sigma(\pi(i))$ for all i.)
- $\alpha, \beta \in Aut(R)$

is defined by

$$(\phi, \pi, \alpha)(\psi, \sigma, \beta) = (\phi \cdot \psi^{\pi,\alpha}, \pi\sigma, \alpha \circ \beta)$$

with $\psi^{\pi,\alpha} = (\alpha(\psi_{\pi(1)-1}), \ldots, \alpha(\psi_{\pi(n)-1}))$ and an elementwisely defined multiplication of vectors. (The indexing of vectors is 0-based here, so $\psi = (\psi_0, \psi_1, \ldots, \psi_{n-1})$.)

The parent is `SemimonomialTransformationGroup`.

AUTHORS:

- Thomas Feulner (2012-11-15): initial version
- **Thomas Feulner (2013-12-27): trac ticket #15576 dissolve dependency on**
  Permutations().global_options()['mul']

EXAMPLES:

```
sage: S = SemimonomialTransformationGroup(GF(4, 'a'), 4)
sage: G = S.gens()
sage: G[0]*G[1]
((a, 1, 1, 1); (1,2,3,4), Ring endomorphism of Finite Field in a of size 2^2
  Defn: a |--> a)
```

TESTS:

```
sage: TestSuite(G[0]).run()
```

class `sage.groups.semimonomial_transformations.semimonomial_transformation.SemimonomialTransf`

Bases: `sage.structure.element.MultiplicativeGroupElement`

An element in the semimonomial group over a ring R. See `SemimonomialTransformationGroup` for the details on the multiplication of two elements.

The init method should never be called directly. Use the call via the parent `SemimonomialTransformationGroup`. instead.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: S = SemimonomialTransformationGroup(F, 4)
sage: g = S(v = [2, a, 1, 2])
sage: h = S(perm = Permutation('(1,2,3,4)'), autom=F.hom([a**3]))
sage: g*h
((2, a, 1, 2); (1,2,3,4), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a +
sage: h*g
((2*a + 1, 1, 2, 2); (1,2,3,4), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |-->
sage: S(g)
((2, a, 1, 2); (), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> a)
sage: S(1) # the one element in the group
((1, 1, 1, 1); (), Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> a)
```

get_autom()

Returns the component corresponding to $Aut(R)$ of `self`.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_autom()
Ring endomorphism of Finite Field in a of size 3^2 Defn: a |--> 2*a + 1
```

get_perm()

Returns the component corresponding to S_n of `self`.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_perm()
[4, 1, 2, 3]
```

get_v()

Returns the component corresponding to R^{imes^n} of `self`.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_v()
(a, 1, 1, 1)
```

get_v_inverse()

Returns the (elementwise) inverse of the component corresponding to R^{imes^n} of `self`.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: SemimonomialTransformationGroup(F, 4).an_element().get_v_inverse()
(a + 2, 1, 1, 1)
```

invert_v()

Elementwisely inverts all entries of `self` which correspond to the component R^{imes^n}.

The other components of `self` keep unchanged.

EXAMPLES:

```
sage: F.<a> = GF(9)
sage: x = copy(SemimonomialTransformationGroup(F, 4).an_element())
sage: x.invert_v();
sage: x.get_v() == SemimonomialTransformationGroup(F, 4).an_element().get_v_inverse()
True
```

CHAPTER

FIFTY

CLASS FUNCTIONS OF GROUPS.

This module implements a wrapper of GAP's ClassFunction function.

NOTE: The ordering of the columns of the character table of a group corresponds to the ordering of the list. However, in general there is no way to canonically list (or index) the conjugacy classes of a group. Therefore the ordering of the columns of the character table of a group is somewhat random.

AUTHORS:

- Franco Saliola (November 2008): initial version
- Volker Braun (October 2010): Bugfixes, exterior and symmetric power.

`sage.groups.class_function.`**`ClassFunction`**`(`*group*, *values*`)`

Construct a class function.

INPUT:

- `group` – a group.
- `values` – list/tuple/iterable of numbers. The values of the class function on the conjugacy classes, in that order.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: G.conjugacy_classes()
[Conjugacy class of () in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,2,3,4) in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,3)(2,4) in Cyclic group of order 4 as a permutation group,
 Conjugacy class of (1,4,3,2) in Cyclic group of order 4 as a permutation group]
sage: values  = [1, -1, 1, -1]
sage: chi = ClassFunction(G, values); chi
Character of Cyclic group of order 4 as a permutation group
```

class `sage.groups.class_function.`**`ClassFunction_gap`**`(`*G*, *values*`)`

Bases: `sage.structure.sage_object.SageObject`

A wrapper of GAP's ClassFunction function.

Note: It is *not* checked whether the given values describes a character, since GAP does not do this.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: values  = [1, -1, 1, -1]
sage: chi = ClassFunction(G, values); chi
Character of Cyclic group of order 4 as a permutation group
```

```
sage: loads(dumps(chi)) == chi
True
```

central_character()

Returns the central character of self.

EXAMPLES:

```
sage: t = SymmetricGroup(4).trivial_character()
sage: t.central_character().values()
[1, 6, 3, 8, 6]
```

decompose()

Returns a list of the characters that appear in the decomposition of chi.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, 2, -3])
sage: chi.decompose()
((3, Character of Symmetric group of order 5! as a permutation group),
 (2, Character of Symmetric group of order 5! as a permutation group))
```

degree()

Returns the degree of the character self.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: irr = S5.irreducible_characters()
sage: [x.degree() for x in irr]
[1, 4, 5, 6, 5, 4, 1]
```

determinant_character()

Returns the determinant character of self.

EXAMPLES:

```
sage: t = ClassFunction(SymmetricGroup(4), [1, -1, 1, 1, -1])
sage: t.determinant_character().values()
[1, -1, 1, 1, -1]
```

domain()

Returns the domain of the self.

OUTPUT:

The underlying group of the class function.

EXAMPLES:

```
sage: ClassFunction(SymmetricGroup(4), [1,-1,1,1,-1]).domain()
Symmetric group of order 4! as a permutation group
```

exterior_power(*n*)

Returns the anti-symmetrized product of self with itself n times.

INPUT:

- n – a positive integer.

OUTPUT:

The n-th anti-symmetrized power of self as a ClassFunction.

EXAMPLES:

```
sage: chi = ClassFunction(SymmetricGroup(4), [3, 1, -1, 0, -1])
sage: p = chi.exterior_power(3)    # the highest anti-symmetric power for a 3-d character
sage: p
Character of Symmetric group of order 4! as a permutation group
sage: p.values()
[1, -1, 1, 1, -1]
sage: p == chi.determinant_character()
True
```

induct(*G*)

Return the induced character.

INPUT:

- G – A supergroup of the underlying group of self.

OUTPUT:

A ClassFunction of G defined by induction. Induction is the adjoint functor to restriction, see restrict().

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: H = G.subgroup([(1,2,3), (1,2), (4,5)])
sage: xi = H.trivial_character(); xi
Character of Subgroup of (Symmetric group of order 5! as a permutation group) generated by [
sage: xi.induct(G)
Character of Symmetric group of order 5! as a permutation group
sage: xi.induct(G).values()
[10, 4, 2, 1, 1, 0, 0]
```

irreducible_constituents()

Returns a list of the characters that appear in the decomposition of chi.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, 2, -3])
sage: irr = chi.irreducible_constituents(); irr
(Character of Symmetric group of order 5! as a permutation group,
 Character of Symmetric group of order 5! as a permutation group)
sage: map(list, irr)
[[4, -2, 0, 1, 1, 0, -1], [5, -1, 1, -1, -1, 1, 0]]
sage: G = GL(2,3)
sage: chi = ClassFunction(G, [-1, -1, -1, -1, -1, -1, -1, -1])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [1, 1, 1, 1, 1, 1, 1, 1])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [2, 2, 2, 2, 2, 2, 2, 2])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [-1, -1, -1, -1, 3, -1, -1, 1])
sage: ic = chi.irreducible_constituents(); ic
(Character of General Linear Group of degree 2 over Finite Field of size 3,
 Character of General Linear Group of degree 2 over Finite Field of size 3)
```

```
sage: map(list, ic)
[[2, -1, 2, -1, 2, 0, 0, 0], [3, 0, 3, 0, -1, 1, 1, -1]]
```

is_irreducible()

Returns True if self cannot be written as the sum of two nonzero characters of self.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [x.is_irreducible() for x in irr]
[True, True, True, True, True]
```

norm()

Returns the norm of self.

EXAMPLES:

```
sage: A5 = AlternatingGroup(5)
sage: [x.norm() for x in A5.irreducible_characters()]
[1, 1, 1, 1, 1]
```

restrict(*H*)

Return the restricted character.

INPUT:

- `H` – a subgroup of the underlying group of `self`.

OUTPUT:

A `ClassFunction` of `H` defined by restriction.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: chi = ClassFunction(G, [3, -3, -1, 0, 0, -1, 3]); chi
Character of Symmetric group of order 5! as a permutation group
sage: H = G.subgroup([(1,2,3), (1,2), (4,5)])
sage: chi.restrict(H)
Character of Subgroup of (Symmetric group of order 5! as a permutation group) generated by [
sage: chi.restrict(H).values()
[3, -3, -3, -1, 0, 0]
```

scalar_product(*other*)

Returns the scalar product of self with other.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [[x.scalar_product(y) for x in irr] for y in irr]
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

symmetric_power(*n*)

Returns the symmetrized product of self with itself `n` times.

INPUT:

•n – a positive integer.

OUTPUT:

The n-th symmetrized power of self as a ClassFunction.

EXAMPLES:

```
sage: chi = ClassFunction(SymmetricGroup(4), [3, 1, -1, 0, -1])
sage: p = chi.symmetric_power(3)
sage: p
Character of Symmetric group of order 4! as a permutation group
sage: p.values()
[10, 2, -2, 1, 0]
```

tensor_product(*other*)

EXAMPLES:

```
sage: S3 = SymmetricGroup(3)
sage: chi1, chi2, chi3 = S3.irreducible_characters()
sage: chi1.tensor_product(chi3).values()
[1, -1, 1]
```

values()

Return the list of values of self on the conjugacy classes.

EXAMPLES:

```
sage: G = GL(2,3)
sage: [x.values() for x in G.irreducible_characters()] #random
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, -1, -1, -1],
 [2, -1, 2, -1, 2, 0, 0, 0],
 [2, 1, -2, -1, 0, -zeta8^3 - zeta8, zeta8^3 + zeta8, 0],
 [2, 1, -2, -1, 0, zeta8^3 + zeta8, -zeta8^3 - zeta8, 0],
 [3, 0, 3, 0, -1, -1, -1, 1],
 [3, 0, 3, 0, -1, 1, 1, -1],
 [4, -1, -4, 1, 0, 0, 0, 0]]
```

TESTS:

```
sage: G = GL(2,3)
sage: k = CyclotomicField(8)
sage: zeta8 = k.gen()
sage: v = [tuple(x.values()) for x in G.irreducible_characters()]
sage: set(v) == set([(1, 1, 1, 1, 1, 1, 1, 1), (1, 1, 1, 1, 1, -1, -1, -1), (2, -1, 2, -1, 2
True
```

class sage.groups.class_function.**ClassFunction_libgap**(*G*, *values*)

Bases: sage.structure.sage_object.SageObject

A wrapper of GAP's ClassFunction function.

Note: It is *not* checked whether the given values describes a character, since GAP does not do this.

EXAMPLES:

```
sage: G = SO(3,3)
sage: values  = [1, -1, -1, 1, 2]
sage: chi = ClassFunction(G, values); chi
Character of Special Orthogonal Group of degree 3 over Finite Field of size 3
```

```
sage: loads(dumps(chi)) == chi
True
```

central_character()

Return the central character of `self`.

EXAMPLES:

```
sage: t = SymmetricGroup(4).trivial_character()
sage: t.central_character().values()
[1, 6, 3, 8, 6]
```

decompose()

Return a list of the characters that appear in the decomposition of `self`.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, 2, -3])
sage: chi.decompose()
((3, Character of Symmetric group of order 5! as a permutation group),
 (2, Character of Symmetric group of order 5! as a permutation group))
```

degree()

Return the degree of the character `self`.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: irr = S5.irreducible_characters()
sage: [x.degree() for x in irr]
[1, 4, 5, 6, 5, 4, 1]
```

determinant_character()

Return the determinant character of `self`.

EXAMPLES:

```
sage: t = ClassFunction(SymmetricGroup(4), [1, -1, 1, 1, -1])
sage: t.determinant_character().values()
[1, -1, 1, 1, -1]
```

domain()

Return the domain of `self`.

OUTPUT:

The underlying group of the class function.

EXAMPLES:

```
sage: ClassFunction(SymmetricGroup(4), [1,-1,1,1,-1]).domain()
Symmetric group of order 4! as a permutation group
```

exterior_power(*n*)

Return the anti-symmetrized product of `self` with itself `n` times.

INPUT:

- `n` – a positive integer

OUTPUT:

The n-th anti-symmetrized power of self as a ClassFunction.

EXAMPLES:

```
sage: chi = ClassFunction(SymmetricGroup(4), [3, 1, -1, 0, -1])
sage: p = chi.exterior_power(3)   # the highest anti-symmetric power for a 3-d character
sage: p
Character of Symmetric group of order 4! as a permutation group
sage: p.values()
[1, -1, 1, 1, -1]
sage: p == chi.determinant_character()
True
```

gap()

Return the underlying LibGAP element.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: values  = [1, -1, 1, -1]
sage: chi = ClassFunction(G, values);  chi
Character of Cyclic group of order 4 as a permutation group
sage: type(chi)
<class 'sage.groups.class_function.ClassFunction_gap'>
sage: gap(chi)
ClassFunction( CharacterTable( Group( [ (1,2,3,4) ] ) ), [ 1, -1, 1, -1 ] )
sage: type(_)
<class 'sage.interfaces.gap.GapElement'>
```

induct(*G*)

Return the induced character.

INPUT:

- G – A supergroup of the underlying group of self.

OUTPUT:

A ClassFunction of G defined by induction. Induction is the adjoint functor to restriction, see restrict().

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: H = G.subgroup([(1,2,3), (1,2), (4,5)])
sage: xi = H.trivial_character(); xi
Character of Subgroup of (Symmetric group of order 5! as a permutation group) generated by [
sage: xi.induct(G)
Character of Symmetric group of order 5! as a permutation group
sage: xi.induct(G).values()
[10, 4, 2, 1, 1, 0, 0]
```

irreducible_constituents()

Return a list of the characters that appear in the decomposition of self.

EXAMPLES:

```
sage: S5 = SymmetricGroup(5)
sage: chi = ClassFunction(S5, [22, -8, 2, 1, 1, 2, -3])
sage: irr = chi.irreducible_constituents(); irr
(Character of Symmetric group of order 5! as a permutation group,
 Character of Symmetric group of order 5! as a permutation group)
sage: map(list, irr)
```

```
[[4, -2, 0, 1, 1, 0, -1], [5, -1, 1, -1, -1, 1, 0]]

sage: G = GL(2,3)
sage: chi = ClassFunction(G, [-1, -1, -1, -1, -1, -1, -1, -1])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [1, 1, 1, 1, 1, 1, 1, 1])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [2, 2, 2, 2, 2, 2, 2, 2])
sage: chi.irreducible_constituents()
(Character of General Linear Group of degree 2 over Finite Field of size 3,)
sage: chi = ClassFunction(G, [-1, -1, -1, -1, 3, -1, -1, 1])
sage: ic = chi.irreducible_constituents(); ic
(Character of General Linear Group of degree 2 over Finite Field of size 3,
 Character of General Linear Group of degree 2 over Finite Field of size 3)
sage: map(list, ic)
[[2, -1, 2, -1, 2, 0, 0, 0], [3, 0, 3, 0, -1, 1, 1, -1]]
```

is_irreducible()

Return `True` if `self` cannot be written as the sum of two nonzero characters of `self`.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [x.is_irreducible() for x in irr]
[True, True, True, True, True]
```

norm()

Return the norm of `self`.

EXAMPLES:

```
sage: A5 = AlternatingGroup(5)
sage: [x.norm() for x in A5.irreducible_characters()]
[1, 1, 1, 1, 1]
```

restrict(*H*)

Return the restricted character.

INPUT:

- `H` – a subgroup of the underlying group of `self`.

OUTPUT:

A `ClassFunction` of `H` defined by restriction.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: chi = ClassFunction(G, [3, -3, -1, 0, 0, -1, 3]); chi
Character of Symmetric group of order 5! as a permutation group
sage: H = G.subgroup([(1,2,3), (1,2), (4,5)])
sage: chi.restrict(H)
Character of Subgroup of (Symmetric group of order 5! as a permutation group) generated by [
sage: chi.restrict(H).values()
[3, -3, -3, -1, 0, 0]
```

scalar_product(*other*)

Return the scalar product of `self` with `other`.

EXAMPLES:

```
sage: S4 = SymmetricGroup(4)
sage: irr = S4.irreducible_characters()
sage: [[x.scalar_product(y) for x in irr] for y in irr]
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

symmetric_power(*n*)

Return the symmetrized product of `self` with itself `n` times.

INPUT:

- `n` – a positive integer

OUTPUT:

The `n`-th symmetrized power of `self` as a `ClassFunction`.

EXAMPLES:

```
sage: chi = ClassFunction(SymmetricGroup(4), [3, 1, -1, 0, -1])
sage: p = chi.symmetric_power(3)
sage: p
Character of Symmetric group of order 4! as a permutation group
sage: p.values()
[10, 2, -2, 1, 0]
```

tensor_product(*other*)

Return the tensor product of `self` and `other`.

EXAMPLES:

```
sage: S3 = SymmetricGroup(3)
sage: chi1, chi2, chi3 = S3.irreducible_characters()
sage: chi1.tensor_product(chi3).values()
[1, -1, 1]
```

values()

Return the list of values of self on the conjugacy classes.

EXAMPLES:

```
sage: G = GL(2,3)
sage: [x.values() for x in G.irreducible_characters()] #random
[[1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, -1, -1, -1],
 [2, -1, 2, -1, 2, 0, 0, 0],
 [2, 1, -2, -1, 0, -zeta8^3 - zeta8, zeta8^3 + zeta8, 0],
 [2, 1, -2, -1, 0, zeta8^3 + zeta8, -zeta8^3 - zeta8, 0],
 [3, 0, 3, 0, -1, -1, -1, 1],
 [3, 0, 3, 0, -1, 1, 1, -1],
 [4, -1, -4, 1, 0, 0, 0, 0]]
```

TESTS:

```
sage: G = GL(2,3)
sage: k = CyclotomicField(8)
sage: zeta8 = k.gen()
sage: v = [tuple(x.values()) for x in G.irreducible_characters()]
sage: set(v) == set([(1, 1, 1, 1, 1, 1, 1, 1), (1, 1, 1, 1, 1, -1, -1, -1), (2, -1, 2, -1, 2
True
```

CHAPTER

FIFTYONE

CONJUGACY CLASSES OF GROUPS

This module implements a wrapper of GAP's `ConjugacyClass` function.

There are two main classes, `ConjugacyClass` and `ConjugacyClassGAP`. All generic methods should go into `ConjugacyClass`, whereas `ConjugacyClassGAP` should only contain wrappers for GAP functions. `ConjugacyClass` contains some fallback methods in case some group cannot be defined as a GAP object.

Todo

- Implement a non-naive fallback method for computing all the elements of the conjugacy class when the group is not defined in GAP, as the one in Butler's paper.
- Define a sage method for gap matrices so that groups of matrices can use the quicker GAP algorithm rather than the naive one.

EXAMPLES:

Conjugacy classes for groups of permutations:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: G.conjugacy_class(g)
Conjugacy class of cycle type [4] in Symmetric group of order 4! as a permutation group
```

Conjugacy classes for groups of matrices:

```
sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
sage: H = MatrixGroup(gens)
sage: h = H(matrix(F,2,[1,2, -1, 1]))
sage: H.conjugacy_class(h)
Conjugacy class of [1 2]
[4 1] in Matrix group over Finite Field of size 5 with 2 generators (
[1 2]  [1 1]
[4 1], [0 1]
)
```

TESTS:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2,3))
sage: C = ConjugacyClass(G,g)
sage: TestSuite(C).run()
```

class `sage.groups.conjugacy_classes.`**`ConjugacyClass`**`(`*group*, *element*`)`

Bases: `sage.structure.parent.Parent`

Generic conjugacy classes for elements in a group.

This is the default fall-back implementation to be used whenever GAP cannot handle the group.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: ConjugacyClass(G,g)
Conjugacy class of (1,2,3,4) in Symmetric group of order 4! as a
permutation group
```

an_element()

Return a representative of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2,3))
sage: C = ConjugacyClass(G,g)
sage: C.representative()
(1,2,3)
```

is_rational()

Checks if `self` is rational (closed for powers).

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: c = ConjugacyClass(G,g)
sage: c.is_rational()
False
```

is_real()

Checks if `self` is real (closed for inverses).

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: c = ConjugacyClass(G,g)
sage: c.is_real()
True
```

list()

Return a list with all the elements of `self`.

EXAMPLES:

Groups of permutations:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2,3))
sage: c = ConjugacyClass(G,g)
sage: L = c.list()
sage: Set(L) == Set([G((1,3,2)), G((1,2,3))])
True
```

representative()

Return a representative of `self`.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2,3))
sage: C = ConjugacyClass(G,g)
sage: C.representative()
(1,2,3)
```

set()

Return the set of elements of the conjugacy class.

EXAMPLES:

Groups of permutations:

```
sage: G = SymmetricGroup(3)
sage: g = G((1,2))
sage: C = ConjugacyClass(G,g)
sage: S = [(2,3), (1,2), (1,3)]
sage: C.set() == Set(G(x) for x in S)
True
```

Groups of matrices over finite fields:

```
sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
sage: H = MatrixGroup(gens)
sage: h = H(matrix(F,2,[1,2, -1, 1]))
sage: C = ConjugacyClass(H,h)
sage: S = [[[3, 2], [2, 4]], [[0, 1], [2, 2]], [[3, 4], [1, 4]],\
      [[0, 3], [4, 2]], [[1, 2], [4, 1]], [[2, 1], [2, 0]],\
      [[4, 1], [4, 3]], [[4, 4], [1, 3]], [[2, 4], [3, 0]],\
      [[1, 4], [2, 1]], [[3, 3], [3, 4]], [[2, 3], [4, 0]],\
      [[0, 2], [1, 2]], [[1, 3], [1, 1]], [[4, 3], [3, 3]],\
      [[4, 2], [2, 3]], [[0, 4], [3, 2]], [[1, 1], [3, 1]],\
      [[2, 2], [1, 0]], [[3, 1], [4, 4]]]
sage: C.set() == Set(H(x) for x in S)
True
```

It is not implemented for infinite groups:

```
sage: a = matrix(ZZ,2,[1,1,0,1])
sage: b = matrix(ZZ,2,[1,0,1,1])
sage: G = MatrixGroup([a,b])        # takes 1s
sage: g = G(a)
sage: C = ConjugacyClass(G, g)
sage: C.set()
Traceback (most recent call last):
...
NotImplementedError: Listing the elements of conjugacy classes is not implemented for infini
```

class sage.groups.conjugacy_classes.**ConjugacyClassGAP**(*group*, *element*)

Bases: sage.groups.conjugacy_classes.ConjugacyClass

Class for a conjugacy class for groups defined over GAP. Intended for wrapping GAP methods on conjugacy classes.

INPUT:

- group – the group in which the conjugacy class is taken
- element – the element generating the conjugacy class

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: ConjugacyClassGAP(G,g)
Conjugacy class of (1,2,3,4) in Symmetric group of order 4! as a
permutation group
```

cardinality()

Return the size of this conjugacy class.

EXAMPLES:

```
sage: W = WeylGroup(['C',6])
sage: cc = W.conjugacy_class(W.an_element())
sage: cc.cardinality()
3840
sage: type(cc.cardinality())
<type 'sage.rings.integer.Integer'>
```

set()

Return a Sage `Set` with all the elements of the conjugacy class.

By default attempts to use GAP construction of the conjugacy class. If GAP method is not implemented for the given group, and the group is finite, falls back to a naive algorithm.

Warning: The naive algorithm can be really slow and memory intensive.

EXAMPLES:

Groups of permutations:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4))
sage: C = ConjugacyClassGAP(G,g)
sage: S = [(1,3,2,4), (1,4,3,2), (1,3,4,2), (1,2,3,4), (1,4,2,3), (1,2,4,3)]
sage: C.set() == Set(G(x) for x in S)
True
```

CHAPTER

FIFTYTWO

MISSING TITLE

CHAPTER

FIFTYTHREE

MISSING TITLE

CHAPTER

FIFTYFOUR

INTERNALS

54.1 Base for Classical Matrix Groups

This module implements the base class for matrix groups that have various famous names, like the general linear group.

EXAMPLES:

```
sage: SL(2, ZZ)
Special Linear Group of degree 2 over Integer Ring
sage: G = SL(2,GF(3)); G
Special Linear Group of degree 2 over Finite Field of size 3
sage: G.is_finite()
True
sage: G.conjugacy_class_representatives()
(
[1 0]  [0 2]  [0 1]  [2 0]  [0 2]  [0 1]  [0 2]
[0 1], [1 1], [2 1], [0 2], [1 2], [2 2], [1 0]
)
sage: G = SL(6,GF(5))
sage: G.gens()
(
[2 0 0 0 0 0]  [4 0 0 0 0 1]
[0 3 0 0 0 0]  [4 0 0 0 0 0]
[0 0 1 0 0 0]  [0 4 0 0 0 0]
[0 0 0 1 0 0]  [0 0 4 0 0 0]
[0 0 0 0 1 0]  [0 0 0 4 0 0]
[0 0 0 0 0 1], [0 0 0 0 4 0]
)
```

class `sage.groups.matrix_gps.named_group.`**`NamedMatrixGroup_gap`**(*degree*, *base_ring*, *special*, *sage_name*, *latex_string*, *gap_command_string*)

Bases: `sage.groups.matrix_gps.named_group.NamedMatrixGroup_generic`, `sage.groups.matrix_gps.matrix_group.MatrixGroup_gap`

Base class for "named" matrix groups using LibGAP

INPUT:

- `degree` – integer. The degree (number of rows/columns of matrices).
- `base_ring` – rinrg. The base ring of the matrices.
- `special` – boolean. Whether the matrix group is special, that is, elements have determinant one.

- `latex_string` – string. The latex representation.
- `gap_command_string` – string. The GAP command to construct the matrix group.

EXAMPLES:

```
sage: G = GL(2, GF(3))
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_gap
sage: isinstance(G, NamedMatrixGroup_gap)
True
```

class `sage.groups.matrix_gps.named_group.`**`NamedMatrixGroup_generic`**`(`*degree*, *base_ring*, *special*, *sage_name*, *latex_string*`)`

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.groups.matrix_gps.matrix_group.MatrixGroup_generic`

Base class for "named" matrix groups

INPUT:

- `degree` – integer. The degree (number of rows/columns of matrices).
- `base_ring` – rinrg. The base ring of the matrices.
- `special` – boolean. Whether the matrix group is special, that is, elements have determinant one.
- `latex_string` – string. The latex representation.

EXAMPLES:

```
sage: G = GL(2, QQ)
sage: from sage.groups.matrix_gps.named_group import NamedMatrixGroup_generic
sage: isinstance(G, NamedMatrixGroup_generic)
True
```

`sage.groups.matrix_gps.named_group.`**`normalize_args_vectorspace`**`(`**args*, ***kwds*`)`

Normalize the arguments that relate to a vector space.

INPUT:

Something that defines an affine space. For example

- An affine space itself:
  - `A` – affine space
- A vector space:
  - `V` – a vector space
- Degree and base ring:
  - `degree` – integer. The degree of the affine group, that is, the dimension of the affine space the group is acting on.
  - `ring` – a ring or an integer. The base ring of the affine space. If an integer is given, it must be a prime power and the corresponding finite field is constructed.
  - `var='a'` – optional keyword argument to specify the finite field generator name in the case where `ring` is a prime power.

OUTPUT:

A pair `(degree, ring)`.

TESTS:

```
sage: from sage.groups.matrix_gps.named_group import normalize_args_vectorspace
sage: A = AffineSpace(2, GF(4,'a'));  A
Affine Space of dimension 2 over Finite Field in a of size 2^2
sage: normalize_args_vectorspace(A)
(2, Finite Field in a of size 2^2)

sage: normalize_args_vectorspace(2,4)   # shorthand
(2, Finite Field in a of size 2^2)

sage: V = ZZ^3;  V
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: normalize_args_vectorspace(V)
(3, Integer Ring)

sage: normalize_args_vectorspace(2, QQ)
(2, Rational Field)
```

CHAPTER

FIFTYFIVE

INDICES AND TABLES

- Index
- Module Index
- Search Page

[JohnsonPG90] D.L. Johnson. *Presentations of Groups*. Cambridge University Press. (1990).

[Bigelow] Bigelow, Stephen J. The Lawrence-Krammer representation. Arxiv math/0204057v1

[Big99] Stephen J. Bigelow. The Burau representation is not faithful for $n = 5$. Geom. Topol., 3:397–404, 1999.

[JonesNotes] Vaughan Jones. The Jones Polynomial. https://math.berkeley.edu/~vfr/jones.pdf

[Lic] William B. Raymond Lickorish. An Introduction to Knot Theory, volume 175 of Graduate Texts in Mathematics. Springer-Verlag, New York, 1997. ISBN 0-387-98254-X

[Dynnikov07] I. Dynnikov and B. Wiest, On the complexity of braids, J. Europ. Math. Soc. 9 (2007)

[Dehornoy] P. Dehornoy, Le probleme d'isotopie des tresses, in lecons de mathematiques d'aujourd'hui vol. 4

[Wan10] Zhenghan Wang. Tolological quantum computation. Providence, RI: American Mathematical Society (AMS), 2010. ISBN 978-0-8218-4930-9

[Charney2006] Ruth Charney. *An introduction to right-angled Artin groups*. http://people.brandeis.edu/~charney/papers/RAAGfinal.pdf, Arxiv math/0610668.

[BB1997] Mladen Bestvina and Noel Brady. *Morse theory and finiteness properties of groups*. Invent. Math. **129** (1997). No. 3, 445-470. www.math.ou.edu/~nbrady/papers/morse.ps.

[Droms1987] Carl Droms. *Isomorphisms of graph groups*. Proc. of the Amer. Math. Soc. **100** (1987). No 3. http://educ.jmu.edu/~dromscg/vita/preprints/Isomorphisms.pdf

[CP2001] John Crisp and Luis Paris. *The solution to a conjecture of Tits on the subgroup generated by the squares of the generators of an Artin group*. Invent. Math. **145** (2001). No 1, 19-36. Arxiv math/0003133.

[VW1994] Leonard Van Wyk. *Graph groups are biautomatic*. J. Pure Appl. Alg. **94** (1994). no. 3, 341-352.

[THOMAS-WOODS] A.D. Thomas and G.V. Wood, Group Tables (Exeter: Shiva Publishing, 1980)

[CONRAD2009] Groups of order 12. Keith Conrad, accessed 21 October 2009.

[GORENSTEIN1980] Daniel Gorenstein, Finite Groups (New York: Chelsea Publishing, 1980)

[GORENSTEIN] Daniel Gorenstein, Finite Groups (New York: Chelsea Publishing, 1980)

PYTHON MODULE INDEX

g

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y