Sage Reference Manual: Tensors on free modules of finite rank

Release 7.3

The Sage Development Team

Aug 04, 2016

1 Free modules of finite rank **3**

2 Free module bases **35**

3 Tensors **39**
3.1 Tensor products of free modules . . . 39
3.2 Tensors on free modules . . . 45
3.3 Index notation for tensors . . . 74

4 Alternating forms **77**
4.1 Exterior powers of dual free modules . . . 77
4.2 Alternating forms on free modules . . . 81

5 Morphisms **91**
5.1 Sets of morphisms between free modules . . . 91
5.2 Free module morphisms . . . 95
5.3 General linear group of a free module . . . 101
5.4 Free module automorphisms . . . 106

6 Components as indexed sets of ring elements **121**

7 Formatting utilities **159**

8 Indices and Tables **163**

This work is part of the SageManifolds project but it does not depend upon other SageManifolds classes. In other words, it constitutes a self-consistent subset that can be used independently of SageManifolds.

CHAPTER

ONE

FREE MODULES OF FINITE RANK

The class *FiniteRankFreeModule* implements free modules of finite rank over a commutative ring.

A *free module of finite rank* over a commutative ring R is a module M over R that admits a *finite basis*, i.e. a finite familly of linearly independent generators. Since R is commutative, it has the invariant basis number property, so that the rank of the free module M is defined uniquely, as the cardinality of any basis of M.

No distinguished basis of M is assumed. On the contrary, many bases can be introduced on the free module along with change-of-basis rules (as module automorphisms). Each module element has then various representations over the various bases.

Note: The class *FiniteRankFreeModule* does not inherit from class `FreeModule_generic` nor from class `CombinatorialFreeModule`, since both classes deal with modules with a *distinguished basis* (see details *below*). Accordingly, the class *FiniteRankFreeModule* inherits directly from the generic class `Parent` with the category set to `Modules` (and not to `ModulesWithBasis`).

Todo

- implement submodules
- create a FreeModules category (cf. the *TODO* statement in the documentation of `Modules`: *Implement a ''FreeModules(R)'' category, when so prompted by a concrete use case*)

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version

REFERENCES:

- Chap. 10 of R. Godement : *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)
- Chap. 3 of S. Lang : *Algebra*, 3rd ed., Springer (New York) (2002)

EXAMPLES:

Let us define a free module of rank 2 over $\mathbf{Z}$:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M') ; M
Rank-2 free module M over the Integer Ring
sage: M.category()
Category of finite dimensional modules over Integer Ring
```

We introduce a first basis on `M`:

```
sage: e = M.basis('e') ; e
Basis (e_0,e_1) on the Rank-2 free module M over the Integer Ring
```

The elements of the basis are of course module elements:

```
sage: e[0]
Element e_0 of the Rank-2 free module M over the Integer Ring
sage: e[1]
Element e_1 of the Rank-2 free module M over the Integer Ring
sage: e[0].parent()
Rank-2 free module M over the Integer Ring
```

We define a module element by its components w.r.t. basis `e` :

```
sage: u = M([2,-3], basis=e, name='u')
sage: u.display(e)
u = 2 e_0 - 3 e_1
```

Module elements can be also be created by arithmetic expressions:

```
sage: v = -2*u + 4*e[0] ; v
Element of the Rank-2 free module M over the Integer Ring
sage: v.display(e)
6 e_1
sage: u == 2*e[0] - 3*e[1]
True
```

We define a second basis on `M` from a family of linearly independent elements:

```
sage: f = M.basis('f', from_family=(e[0]-e[1], -2*e[0]+3*e[1])) ; f
Basis (f_0,f_1) on the Rank-2 free module M over the Integer Ring
sage: f[0].display(e)
f_0 = e_0 - e_1
sage: f[1].display(e)
f_1 = -2 e_0 + 3 e_1
```

We may of course express the elements of basis `e` in terms of basis `f` :

```
sage: e[0].display(f)
e_0 = 3 f_0 + f_1
sage: e[1].display(f)
e_1 = 2 f_0 + f_1
```

as well as any module element:

```
sage: u.display(f)
u = -f_1
sage: v.display(f)
12 f_0 + 6 f_1
```

The two bases are related by a module automorphism:

```
sage: a = M.change_of_basis(e,f) ; a
Automorphism of the Rank-2 free module M over the Integer Ring
sage: a.parent()
General linear group of the Rank-2 free module M over the Integer Ring
sage: a.matrix(e)
```

```
[ 1 -2]
[-1  3]
```

Let us check that basis `f` is indeed the image of basis `e` by `a` :

```
sage: f[0] == a(e[0])
True
sage: f[1] == a(e[1])
True
```

The reverse change of basis is of course the inverse automorphism:

```
sage: M.change_of_basis(f,e) == a^(-1)
True
```

We introduce a new module element via its components w.r.t. basis `f` :

```
sage: v = M([2,4], basis=f, name='v')
sage: v.display(f)
v = 2 f_0 + 4 f_1
```

The sum of the two module elements `u` and `v` can be performed even if they have been defined on different bases, thanks to the known relation between the two bases:

```
sage: s = u + v ; s
Element u+v of the Rank-2 free module M over the Integer Ring
```

We can display the result in either basis:

```
sage: s.display(e)
u+v = -4 e_0 + 7 e_1
sage: s.display(f)
u+v = 2 f_0 + 3 f_1
```

Tensor products of elements are implemented:

```
sage: t = u*v ; t
Type-(2,0) tensor u*v on the Rank-2 free module M over the Integer Ring
sage: t.parent()
Free module of type-(2,0) tensors on the
 Rank-2 free module M over the Integer Ring
sage: t.display(e)
u*v = -12 e_0*e_0 + 20 e_0*e_1 + 18 e_1*e_0 - 30 e_1*e_1
sage: t.display(f)
u*v = -2 f_1*f_0 - 4 f_1*f_1
```

We can access to tensor components w.r.t. to a given basis via the square bracket operator:

```
sage: t[e,0,1]
20
sage: t[f,1,0]
-2
sage: u[e,0]
2
sage: u[e,:]
[2, -3]
sage: u[f,:]
[0, -1]
```

The parent of the automorphism `a` is the group $\mathrm{GL}(M)$, but `a` can also be considered as a tensor of type $(1,1)$ on `M`:

```
sage: a.parent()
General linear group of the Rank-2 free module M over the Integer Ring
sage: a.tensor_type()
(1, 1)
sage: a.display(e)
e_0*e^0 - 2 e_0*e^1 - e_1*e^0 + 3 e_1*e^1
sage: a.display(f)
f_0*f^0 - 2 f_0*f^1 - f_1*f^0 + 3 f_1*f^1
```

As such, we can form its tensor product with `t`, yielding a tensor of type $(3,1)$:

```
sage: t*a
Type-(3,1) tensor on the Rank-2 free module M over the Integer Ring
sage: (t*a).display(e)
-12 e_0*e_0*e_0*e^0 + 24 e_0*e_0*e_0*e^1 + 12 e_0*e_0*e_1*e^0
 - 36 e_0*e_0*e_1*e^1 + 20 e_0*e_1*e_0*e^0 - 40 e_0*e_1*e_0*e^1
 - 20 e_0*e_1*e_1*e^0 + 60 e_0*e_1*e_1*e^1 + 18 e_1*e_0*e_0*e^0
 - 36 e_1*e_0*e_0*e^1 - 18 e_1*e_0*e_1*e^0 + 54 e_1*e_0*e_1*e^1
 - 30 e_1*e_1*e_0*e^0 + 60 e_1*e_1*e_0*e^1 + 30 e_1*e_1*e_1*e^0
 - 90 e_1*e_1*e_1*e^1
```

The parent of $t \otimes a$ is itself a free module of finite rank over $\mathbf{Z}$:

```
sage: T = (t*a).parent() ; T
Free module of type-(3,1) tensors on the Rank-2 free module M over the
 Integer Ring
sage: T.base_ring()
Integer Ring
sage: T.rank()
16
```

Differences between `FiniteRankFreeModule` and `FreeModule` (or `VectorSpace`)

To illustrate the differences, let us create two free modules of rank 3 over $\mathbf{Z}$, one with `FiniteRankFreeModule` and the other one with `FreeModule`:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M') ; M
Rank-3 free module M over the Integer Ring
sage: N = FreeModule(ZZ, 3) ; N
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

The main difference is that `FreeModule` returns a free module with a distinguished basis, while `FiniteRankFreeModule` does not:

```
sage: N.basis()
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: M.bases()
[]
sage: M.print_bases()
No basis has been defined on the Rank-3 free module M over the Integer Ring
```

This is also revealed by the category of each module:

```
sage: M.category()
Category of finite dimensional modules over Integer Ring
sage: N.category()
Category of finite dimensional modules with basis over
 (euclidean domains and infinite enumerated sets and metric spaces)
```

In other words, the module created by `FreeModule` is actually $\mathbf{Z}^3$, while, in the absence of any distinguished basis, no *canonical* isomorphism relates the module created by `FiniteRankFreeModule` to $\mathbf{Z}^3$:

```
sage: N is ZZ^3
True
sage: M is ZZ^3
False
sage: M == ZZ^3
False
```

Because it is $\mathbf{Z}^3$, `N` is unique, while there may be various modules of the same rank over the same ring created by `FiniteRankFreeModule` ; they are then distinguished by their names (actually by the complete sequence of arguments of `FiniteRankFreeModule`):

```
sage: N1 = FreeModule(ZZ, 3) ; N1
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: N1 is N  # FreeModule(ZZ, 3) is unique
True
sage: M1 = FiniteRankFreeModule(ZZ, 3, name='M_1') ; M1
Rank-3 free module M_1 over the Integer Ring
sage: M1 is M  # M1 and M are different rank-3 modules over ZZ
False
sage: M1b = FiniteRankFreeModule(ZZ, 3, name='M_1') ; M1b
Rank-3 free module M_1 over the Integer Ring
sage: M1b is M1  # because M1b and M1 have the same name
True
```

As illustrated above, various bases can be introduced on the module created by `FiniteRankFreeModule` :

```
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: f = M.basis('f', from_family=(-e[0], e[1]-e[2], -2*e[1]+3*e[2])) ; f
Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring
sage: M.bases()
[Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring,
 Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring]
```

Each element of a basis is accessible via its index:

```
sage: e[0]
Element e_0 of the Rank-3 free module M over the Integer Ring
sage: e[0].parent()
Rank-3 free module M over the Integer Ring
sage: f[1]
Element f_1 of the Rank-3 free module M over the Integer Ring
sage: f[1].parent()
Rank-3 free module M over the Integer Ring
```

while on module `N` , the element of the (unique) basis is accessible directly from the module symbol:

```
sage: N.0
(1, 0, 0)
sage: N.1
(0, 1, 0)
sage: N.0.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

The arithmetic of elements is similar; the difference lies in the display: a basis has to be specified for elements of `M`, while elements of `N` are displayed directly as elements of $\mathbf{Z}^3$:

```
sage: u = 2*e[0] - 3*e[2] ; u
Element of the Rank-3 free module M over the Integer Ring
sage: u.display(e)
2 e_0 - 3 e_2
sage: u.display(f)
-2 f_0 - 6 f_1 - 3 f_2
sage: u[e,:]
[2, 0, -3]
sage: u[f,:]
[-2, -6, -3]
sage: v = 2*N.0 - 3*N.2 ; v
(2, 0, -3)
```

For the case of `M`, in order to avoid to specify the basis if the user is always working with the same basis (e.g. only one basis has been defined), the concept of *default basis* has been introduced:

```
sage: M.default_basis()
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: M.print_bases()
Bases defined on the Rank-3 free module M over the Integer Ring:
 - (e_0,e_1,e_2) (default basis)
 - (f_0,f_1,f_2)
```

This is different from the *distinguished basis* of `N`: it simply means that the mention of the basis can be omitted in function arguments:

```
sage: u.display()  # equivalent to u.display(e)
2 e_0 - 3 e_2
sage: u[:]         # equivalent to u[e,:]
[2, 0, -3]
```

At any time, the default basis can be changed:

```
sage: M.set_default_basis(f)
sage: u.display()
-2 f_0 - 6 f_1 - 3 f_2
```

Another difference between `FiniteRankFreeModule` and `FreeModule` is that for the former the range of indices can be specified (by default, it starts from 0):

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1) ; M
Rank-3 free module M over the Integer Ring
sage: e = M.basis('e') ; e  # compare with (e_0,e_1,e_2) above
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
sage: e[1], e[2], e[3]
(Element e_1 of the Rank-3 free module M over the Integer Ring,
 Element e_2 of the Rank-3 free module M over the Integer Ring,
 Element e_3 of the Rank-3 free module M over the Integer Ring)
```

All the above holds for `VectorSpace` instead of `FreeModule` : the object created by `VectorSpace` is actually a Cartesian power of the base field:

```
sage: V = VectorSpace(QQ,3) ; V
Vector space of dimension 3 over Rational Field
sage: V.category()
Category of finite dimensional vector spaces with basis
 over (quotient fields and metric spaces)
sage: V is QQ^3
True
sage: V.basis()
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
```

To create a vector space without any distinguished basis, one has to use `FiniteRankFreeModule` :

```
sage: V = FiniteRankFreeModule(QQ, 3, name='V') ; V
3-dimensional vector space V over the Rational Field
sage: V.category()
Category of finite dimensional vector spaces over Rational Field
sage: V.bases()
[]
sage: V.print_bases()
No basis has been defined on the 3-dimensional vector space V over the
 Rational Field
```

The class *FiniteRankFreeModule* has been created for the needs of the SageManifolds project, where free modules do not have any distinguished basis. Too kinds of free modules occur in the context of differentiable manifolds (see here for more details):

- the tangent vector space at any point of the manifold
- the set of vector fields on a parallelizable open subset U of the manifold, which is a free module over the algebra of scalar fields on U.

For instance, without any specific coordinate choice, no basis can be distinguished in a tangent space.

On the other side, the modules created by `FreeModule` have much more algebraic functionalities than those created by `FiniteRankFreeModule` . In particular, submodules have not been implemented yet in *FiniteRankFreeModule* . Moreover, modules resulting from `FreeModule` are tailored to the specific kind of their base ring:

- free module over a commutative ring that is not an integral domain ($\mathbf{Z}/6\mathbf{Z}$):

```
sage: R = IntegerModRing(6) ; R
Ring of integers modulo 6
sage: FreeModule(R, 3)
Ambient free module of rank 3 over Ring of integers modulo 6
sage: type(FreeModule(R, 3))
<class 'sage.modules.free_module.FreeModule_ambient_with_category'>
```

- free module over an integral domain that is not principal ($\mathbf{Z}[X]$):

```
sage: R.<X> = ZZ[] ; R
Univariate Polynomial Ring in X over Integer Ring
```

```
sage: FreeModule(R, 3)
Ambient free module of rank 3 over the integral domain Univariate
 Polynomial Ring in X over Integer Ring
sage: type(FreeModule(R, 3))
<class 'sage.modules.free_module.FreeModule_ambient_domain_with_category'>
```

- free module over a principal ideal domain ($\mathbf{Z}$):

```
sage: R = ZZ ; R
Integer Ring
sage: FreeModule(R,3)
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: type(FreeModule(R, 3))
<class 'sage.modules.free_module.FreeModule_ambient_pid_with_category'>
```

On the contrary, all objects constructed with `FiniteRankFreeModule` belong to the same class:

```
sage: R = IntegerModRing(6)
sage: type(FiniteRankFreeModule(R, 3))
<class 'sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule_with_category
 ↪'>
sage: R.<X> = ZZ[]
sage: type(FiniteRankFreeModule(R, 3))
<class 'sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule_with_category
 ↪'>
sage: R = ZZ
sage: type(FiniteRankFreeModule(R, 3))
<class 'sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule_with_category
 ↪'>
```

Differences between `FiniteRankFreeModule` and `CombinatorialFreeModule`

An alternative to construct free modules in Sage is `CombinatorialFreeModule` . However, as `FreeModule` , it leads to a module with a distinguished basis:

```
sage: N = CombinatorialFreeModule(ZZ, [1,2,3]) ; N
Free module generated by {1, 2, 3} over Integer Ring
sage: N.category()
Category of finite dimensional modules with basis over Integer Ring
```

The distinguished basis is returned by the method `basis()` :

```
sage: b = N.basis() ; b
Finite family {1: B[1], 2: B[2], 3: B[3]}
sage: b[1]
B[1]
sage: b[1].parent()
Free module generated by {1, 2, 3} over Integer Ring
```

For the free module `M` created above with `FiniteRankFreeModule` , the method `basis` has at least one argument: the symbol string that specifies which basis is required:

```
sage: e = M.basis('e') ; e
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
sage: e[1]
Element e_1 of the Rank-3 free module M over the Integer Ring
```

```
sage: e[1].parent()
Rank-3 free module M over the Integer Ring
```

The arithmetic of elements is similar:

```
sage: u = 2*e[1] - 5*e[3] ; u
Element of the Rank-3 free module M over the Integer Ring
sage: v = 2*b[1] - 5*b[3] ; v
2*B[1] - 5*B[3]
```

One notices that elements of `N` are displayed directly in terms of their expansions on the distinguished basis. For elements of `M`, one has to use the method *`display()`* in order to specify the basis:

```
sage: u.display(e)
2 e_1 - 5 e_3
```

The components on the basis are returned by the square bracket operator for `M` and by the method `coefficient` for `N`:

```
sage: [u[e,i] for i in {1,2,3}]
[2, 0, -5]
sage: u[e,:]  # a shortcut for the above
[2, 0, -5]
sage: [v.coefficient(i) for i in {1,2,3}]
[2, 0, -5]
```

class sage.tensor.modules.finite_rank_free_module. **FiniteRankFreeModule** (*ring*, *rank*, *name=None*, *latex_name=None*, *start_index=0*, *output_formatter=None*, *category=None*)

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

Free module of finite rank over a commutative ring.

A *free module of finite rank* over a commutative ring R is a module M over R that admits a *finite basis*, i.e. a finite familly of linearly independent generators. Since R is commutative, it has the invariant basis number property, so that the rank of the free module M is defined uniquely, as the cardinality of any basis of M.

No distinguished basis of M is assumed. On the contrary, many bases can be introduced on the free module along with change-of-basis rules (as module automorphisms). Each module element has then various representations over the various bases.

Note: The class *`FiniteRankFreeModule`* does not inherit from class `FreeModule_generic` nor from class `CombinatorialFreeModule`, since both classes deal with modules with a *distinguished basis* (see details *above*). Moreover, following the recommendation exposed in trac ticket #16427 the class *`FiniteRankFreeModule`* inherits directly from `Parent` (with the category set to `Modules`) and not from the Cython class `Module`.

The class *FiniteRankFreeModule* is a Sage *parent* class, the corresponding *element* class being *FiniteRankFreeModuleElement*.

INPUT:

- `ring` – commutative ring R over which the free module is constructed
- `rank` – positive integer; rank of the free module
- `name` – (default: `None`) string; name given to the free module
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the freemodule; if none is provided, it is set to `name`
- `start_index` – (default: 0) integer; lower bound of the range of indices in bases defined on the free module
- `output_formatter` – (default: `None`) function or unbound method called to format the output of the tensor components; `output_formatter` must take 1 or 2 arguments: the first argument must be an element of the ring R and the second one, if any, some format specification

EXAMPLES:

Free module of rank 3 over $\mathbf{Z}$:

```
sage: FiniteRankFreeModule._clear_cache_() # for doctests only
sage: M = FiniteRankFreeModule(ZZ, 3) ; M
Rank-3 free module over the Integer Ring
sage: M = FiniteRankFreeModule(ZZ, 3, name='M') ; M  # declaration with a name
Rank-3 free module M over the Integer Ring
sage: M.category()
Category of finite dimensional modules over Integer Ring
sage: M.base_ring()
Integer Ring
sage: M.rank()
3
```

If the base ring is a field, the free module is in the category of vector spaces:

```
sage: V = FiniteRankFreeModule(QQ, 3, name='V') ; V
3-dimensional vector space V over the Rational Field
sage: V.category()
Category of finite dimensional vector spaces over Rational Field
```

The LaTeX output is adjusted via the parameter `latex_name`:

```
sage: latex(M)  # the default is the symbol provided in the string ``name``
M
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', latex_name=r'\mathcal{M}')
sage: latex(M)
\mathcal{M}
```

The free module M has no distinguished basis:

```
sage: M in ModulesWithBasis(ZZ)
False
sage: M in Modules(ZZ)
True
```

In particular, no basis is initialized at the module construction:

```
sage: M.print_bases()
No basis has been defined on the Rank-3 free module M over the Integer Ring
sage: M.bases()
[]
```

Bases have to be introduced by means of the method *basis()* , the first defined basis being considered as the *default basis*, meaning it can be skipped in function arguments required a basis (this can be changed by means of the method *set_default_basis()*):

```
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: M.default_basis()
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
```

A second basis can be created from a family of linearly independent elements expressed in terms of basis `e` :

```
sage: f = M.basis('f', from_family=(-e[0], e[1]+e[2], 2*e[1]+3*e[2]))
sage: f
Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring
sage: M.print_bases()
Bases defined on the Rank-3 free module M over the Integer Ring:
 - (e_0,e_1,e_2) (default basis)
 - (f_0,f_1,f_2)
sage: M.bases()
[Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring,
 Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring]
```

M is a *parent* object, whose elements are instances of *FiniteRankFreeModuleElement* (actually a dynamically generated subclass of it):

```
sage: v = M.an_element() ; v
Element of the Rank-3 free module M over the Integer Ring
sage: from sage.tensor.modules.free_module_tensor import␣
 ↪FiniteRankFreeModuleElement
sage: isinstance(v, FiniteRankFreeModuleElement)
True
sage: v in M
True
sage: M.is_parent_of(v)
True
sage: v.display() # expansion w.r.t. the default basis (e)
e_0 + e_1 + e_2
sage: v.display(f)
-f_0 + f_1
```

The test suite of the category of modules is passed:

```
sage: TestSuite(M).run()
```

Constructing an element of `M` from (the integer) 0 yields the zero element of `M` :

```
sage: M(0)
Element zero of the Rank-3 free module M over the Integer Ring
sage: M(0) is M.zero()
True
```

Non-zero elements are constructed by providing their components in a given basis:

```
sage: v = M([-1,0,3]) ; v  # components in the default basis (e)
Element of the Rank-3 free module M over the Integer Ring
sage: v.display() # expansion w.r.t. the default basis (e)
-e_0 + 3 e_2
sage: v.display(f)
f_0 - 6 f_1 + 3 f_2
sage: v = M([-1,0,3], basis=f) ; v  # components in a specific basis
Element of the Rank-3 free module M over the Integer Ring
sage: v.display(f)
-f_0 + 3 f_2
sage: v.display()
e_0 + 6 e_1 + 9 e_2
sage: v = M([-1,0,3], basis=f, name='v') ; v
Element v of the Rank-3 free module M over the Integer Ring
sage: v.display(f)
v = -f_0 + 3 f_2
sage: v.display()
v = e_0 + 6 e_1 + 9 e_2
```

An alternative is to construct the element from an empty list of componentsand to set the nonzero components afterwards:

```
sage: v = M([], name='v')
sage: v[e,0] = -1
sage: v[e,2] = 3
sage: v.display(e)
v = -e_0 + 3 e_2
```

Indices on the free module, such as indices labelling the element of a basis, are provided by the generator method *irange()* . By default, they range from 0 to the module's rank minus one:

```
sage: list(M.irange())
[0, 1, 2]
```

This can be changed via the parameter `start_index` in the module construction:

```
sage: M1 = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: list(M1.irange())
[1, 2, 3]
```

The parameter `output_formatter` in the constructor of the free module is used to set the output format of tensor components:

```
sage: N = FiniteRankFreeModule(QQ, 3, output_formatter=Rational.numerical_approx)
sage: e = N.basis('e')
sage: v = N([1/3, 0, -2], basis=e)
sage: v[e,:]
[0.333333333333333, 0.000000000000000, -2.00000000000000]
sage: v.display(e)  # default format (53 bits of precision)
0.333333333333333 e_0 - 2.00000000000000 e_2
sage: v.display(e, format_spec=10)  # 10 bits of precision
0.33 e_0 - 2.0 e_2
```

Element
alias of `FiniteRankFreeModuleElement`

alternating_form (*degree*, *name=None*, *latex_name=None*)
Construct an alternating form on the free module.

INPUT:

- `degree` – the degree of the alternating form (i.e. its tensor rank)
- `name` – (default: `None`) string; name given to the alternating form
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the alternating form; if none is provided, the LaTeX symbol is set to `name`

OUTPUT:

- instance of *FreeModuleAltForm*

EXAMPLES:

Alternating forms on a rank-3 module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: a = M.alternating_form(2, 'a') ; a
Alternating form a of degree 2 on the
 Rank-3 free module M over the Integer Ring
```

The nonzero components in a given basis have to be set in a second step, thereby fully specifying the alternating form:

```
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: a.set_comp(e)[0,1] = 2
sage: a.set_comp(e)[1,2] = -3
sage: a.display(e)
a = 2 e^0/\e^1 - 3 e^1/\e^2
```

An alternating form of degree 1 is a linear form:

```
sage: a = M.alternating_form(1, 'a') ; a
Linear form a on the Rank-3 free module M over the Integer Ring
```

To construct such a form, it is preferable to call the method *linear_form()* instead:

```
sage: a = M.linear_form('a') ; a
Linear form a on the Rank-3 free module M over the Integer Ring
```

See *FreeModuleAltForm* for more documentation.

automorphism (*matrix=None*, *basis=None*, *name=None*, *latex_name=None*)

Construct a module automorphism of `self`.

Denoting `self` by M, an automorphism of `self` is an element of the general linear group $\mathrm{GL}(M)$.

INPUT:

- `matrix` – (default: `None`) matrix of size rank(M)*rank(M) representing the automorphism with respect to `basis`; this entry can actually be any material from which a matrix of elements of `self` base ring can be constructed; the *columns* of `matrix` must be the components w.r.t. `basis` of the images of the elements of `basis`. If `matrix` is `None`, the automorphism has to be initialized afterwards by method *set_comp()* or via the operator [].
- `basis` – (default: `None`) basis of `self` defining the matrix representation; if `None` the default basis of `self` is assumed.
- `name` – (default: `None`) string; name given to the automorphism

- `latex_name` – (default: `None`) string; LaTeX symbol to denote the automorphism; if none is provided, the LaTeX symbol is set to `name`

OUTPUT:

- instance of *FreeModuleAutomorphism*

EXAMPLES:

Automorphism of a rank-2 free $\mathbf{Z}$-module:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M')
sage: e = M.basis('e')
sage: a = M.automorphism(matrix=[[1,2],[1,3]], basis=e, name='a') ; a
Automorphism a of the Rank-2 free module M over the Integer Ring
sage: a.parent()
General linear group of the Rank-2 free module M over the Integer Ring
sage: a.matrix(e)
[1 2]
[1 3]
```

An automorphism is a tensor of type (1,1):

```
sage: a.tensor_type()
(1, 1)
sage: a.display(e)
a = e_0*e^0 + 2 e_0*e^1 + e_1*e^0 + 3 e_1*e^1
```

The automorphism components can be specified in a second step, as components of a type-(1,1) tensor:

```
sage: a1 = M.automorphism(name='a')
sage: a1[e,:] = [[1,2],[1,3]]
sage: a1.matrix(e)
[1 2]
[1 3]
sage: a1 == a
True
```

Component by component specification:

```
sage: a2 = M.automorphism(name='a')
sage: a2[0,0] = 1  # component set in the module's default basis (e)
sage: a2[0,1] = 2
sage: a2[1,0] = 1
sage: a2[1,1] = 3
sage: a2.matrix(e)
[1 2]
[1 3]
sage: a2 == a
True
```

See *FreeModuleAutomorphism* for more documentation.

bases ( )

Return the list of bases that have been defined on the free module `self`.

Use the method *print_bases()* to get a formatted output with more information.

OUTPUT:

- list of instances of class *FreeModuleBasis*

EXAMPLES:

Bases on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M_3', start_index=1)
sage: M.bases()
[]
sage: e = M.basis('e')
sage: M.bases()
[Basis (e_1,e_2,e_3) on the Rank-3 free module M_3 over the Integer Ring]
sage: f = M.basis('f')
sage: M.bases()
[Basis (e_1,e_2,e_3) on the Rank-3 free module M_3 over the Integer Ring,
 Basis (f_1,f_2,f_3) on the Rank-3 free module M_3 over the Integer Ring]
```

basis (*symbol*, *latex_symbol=None*, *from_family=None*)

Define or return a basis of the free module `self` .

Let M denotes the free module `self` and n its rank.

The basis can be defined from a set of n linearly independent elements of M by means of the argument `from_family` . If `from_family` is not specified, the basis is created from scratch and, at this stage, is unrelated to bases that could have been defined previously on M. It can be related afterwards by means of the method *set_change_of_basis()* .

If the basis specified by the given symbol already exists, it is simply returned, whatever the value of the arguments `latex_symbol` or `from_family` .

Note that another way to construct a basis of `self` is to use the method *new_basis()* on an existing basis, with the automorphism relating the two bases as an argument.

INPUT:

- `symbol` – string; a letter (of a few letters) to denote a generic element of the basis
- `latex_symbol` – (default: `None`) string; symbol to denote a generic element of the basis; if `None` , the value of `symbol` is used
- `from_family` – (default: `None`) a tuple of n linearly independent elements of the free module `self` (n being the rank of `self`)

OUTPUT:

- instance of *FreeModuleBasis* representing a basis on `self`

EXAMPLES:

Bases on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: e[0]
Element e_0 of the Rank-3 free module M over the Integer Ring
sage: latex(e)
\left(e_0,e_1,e_2\right)
```

The LaTeX symbol can be set explicitely, as the second argument of *basis()* :

```
sage: eps = M.basis('eps', r'\epsilon') ; eps
Basis (eps_0,eps_1,eps_2) on the Rank-3 free module M
 over the Integer Ring
```

```
sage: latex(eps)
\left(\epsilon_0,\epsilon_1,\epsilon_2\right)
```

If the provided symbol is that of an already defined basis, the latter is returned (no new basis is created):

```
sage: M.basis('e') is e
True
sage: M.basis('eps') is eps
True
```

The individual elements of the basis are labelled according the parameter `start_index` provided at the free module construction:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e') ; e
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
sage: e[1]
Element e_1 of the Rank-3 free module M over the Integer Ring
```

Construction of a basis from a family of linearly independent module elements:

```
sage: f1 = -e[2]
sage: f2 = 4*e[1] + 3*e[3]
sage: f3 = 7*e[1] + 5*e[3]
sage: f = M.basis('f', from_family=(f1,f2,f3))
sage: f[1].display()
f_1 = -e_2
sage: f[2].display()
f_2 = 4 e_1 + 3 e_3
sage: f[3].display()
f_3 = 7 e_1 + 5 e_3
```

The change-of-basis automorphisms have been registered:

```
sage: M.change_of_basis(e,f).matrix(e)
[ 0  4  7]
[-1  0  0]
[ 0  3  5]
sage: M.change_of_basis(f,e).matrix(e)
[ 0 -1  0]
[-5  0  7]
[ 3  0 -4]
sage: M.change_of_basis(f,e) == M.change_of_basis(e,f).inverse()
True
```

Check of the change-of-basis e –> f:

```
sage: a = M.change_of_basis(e,f) ; a
Automorphism of the Rank-3 free module M over the Integer Ring
sage: all( f[i] == a(e[i]) for i in M.irange() )
True
```

For more documentation on bases see *FreeModuleBasis*.

change_of_basis (*basis1*, *basis2*)

Return a module automorphism linking two bases defined on the free module `self`.

If the automorphism has not been recorded yet (in the internal dictionary `self._basis_changes`), it is computed by transitivity, i.e. by performing products of recorded changes of basis.

INPUT:

- `basis1` – a basis of `self`, denoted (e_i) below
- `basis2` – a basis of `self`, denoted (f_i) below

OUTPUT:

- instance of *FreeModuleAutomorphism* describing the automorphism P that relates the basis (e_i) to the basis (f_i) according to $f_i = P(e_i)$

EXAMPLES:

Changes of basis on a rank-2 free module:

```
sage: FiniteRankFreeModule._clear_cache_() # for doctests only
sage: M = FiniteRankFreeModule(ZZ, 2, name='M', start_index=1)
sage: e = M.basis('e')
sage: f = M.basis('f', from_family=(e[1]+2*e[2], e[1]+3*e[2]))
sage: P = M.change_of_basis(e,f) ; P
Automorphism of the Rank-2 free module M over the Integer Ring
sage: P.matrix(e)
[1 1]
[2 3]
```

Note that the columns of this matrix contain the components of the elements of basis `f` w.r.t. to basis `e`:

```
sage: f[1].display(e)
f_1 = e_1 + 2 e_2
sage: f[2].display(e)
f_2 = e_1 + 3 e_2
```

The change of basis is cached:

```
sage: P is M.change_of_basis(e,f)
True
```

Check of the change-of-basis automorphism:

```
sage: f[1] == P(e[1])
True
sage: f[2] == P(e[2])
True
```

Check of the reverse change of basis:

```
sage: M.change_of_basis(f,e) == P^(-1)
True
```

We have of course:

```
sage: M.change_of_basis(e,e)
Identity map of the Rank-2 free module M over the Integer Ring
sage: M.change_of_basis(e,e) is M.identity_map()
True
```

Let us introduce a third basis on `M`:

```
sage: h = M.basis('h', from_family=(3*e[1]+4*e[2], 5*e[1]+7*e[2]))
```

The change of basis `e -> h` has been recorded directly from the definition of `h` :

```
sage: Q = M.change_of_basis(e,h) ; Q.matrix(e)
[3 5]
[4 7]
```

The change of basis `f -> h` is computed by transitivity, i.e. from the changes of basis `f -> e` and `e -> h` :

```
sage: R = M.change_of_basis(f,h) ; R
Automorphism of the Rank-2 free module M over the Integer Ring
sage: R.matrix(e)
[-1  2]
[-2  3]
sage: R.matrix(f)
[ 5  8]
[-2 -3]
```

Let us check that `R` is indeed the change of basis `f -> h` :

```
sage: h[1] == R(f[1])
True
sage: h[2] == R(f[2])
True
```

A related check is:

```
sage: R == Q*P^(-1)
True
```

default_basis ( )

Return the default basis of the free module `self` .

The *default basis* is simply a basis whose name can be skipped in methods requiring a basis as an argument. By default, it is the first basis introduced on the module. It can be changed by the method *set_default_basis()* .

OUTPUT:

- instance of *FreeModuleBasis*

EXAMPLES:

At the module construction, no default basis is assumed:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M', start_index=1)
sage: M.default_basis()
No default basis has been defined on the
 Rank-2 free module M over the Integer Ring
```

The first defined basis becomes the default one:

```
sage: e = M.basis('e') ; e
Basis (e_1,e_2) on the Rank-2 free module M over the Integer Ring
sage: M.default_basis()
Basis (e_1,e_2) on the Rank-2 free module M over the Integer Ring
sage: f =  M.basis('f') ; f
Basis (f_1,f_2) on the Rank-2 free module M over the Integer Ring
sage: M.default_basis()
Basis (e_1,e_2) on the Rank-2 free module M over the Integer Ring
```

dual ( )

Return the dual module of `self` .

EXAMPLE:

Dual of a free module over **Z**:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: M.dual()
Dual of the Rank-3 free module M over the Integer Ring
sage: latex(M.dual())
M^*
```

The dual is a free module of the same rank as M:

```
sage: isinstance(M.dual(), FiniteRankFreeModule)
True
sage: M.dual().rank()
3
```

It is formed by alternating forms of degree 1, i.e. linear forms:

```
sage: M.dual() is M.dual_exterior_power(1)
True
sage: M.dual().an_element()
Linear form on the Rank-3 free module M over the Integer Ring
sage: a = M.linear_form()
sage: a in M.dual()
True
```

The elements of a dual basis belong of course to the dual module:

```
sage: e = M.basis('e')
sage: e.dual_basis()[0] in M.dual()
True
```

dual_exterior_power (*p*)

Return the p-th exterior power of the dual of `self` .

If M stands for the free module `self` , the *p-th exterior power* of the dual of M is the set $\Lambda^p(M^*)$ of all *alternating forms of degree* p on M, i.e. of all multilinear maps

$$\underbrace{M \times \cdots \times M}_{p \text{ times}} \longrightarrow R$$

that vanish whenever any of two of their arguments are equal. $\Lambda^p(M^*)$ is a free module of rank $\binom{n}{p}$ over the same ring as M, where n is the rank of M.

INPUT:

- `p` – non-negative integer

OUTPUT:

- for $p \geq 1$, instance of *ExtPowerFreeModule* representing the free module $\Lambda^p(M^*)$; for $p = 0$, the base ring R is returned instead

EXAMPLES:

Exterior powers of the dual of a free **Z**-module of rank 3:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: M.dual_exterior_power(0)  # return the base ring
Integer Ring
sage: M.dual_exterior_power(1)  # return the dual module
Dual of the Rank-3 free module M over the Integer Ring
sage: M.dual_exterior_power(1) is M.dual()
True
sage: M.dual_exterior_power(2)
2nd exterior power of the dual of the Rank-3 free module M over the Integer␣
↪Ring
sage: M.dual_exterior_power(2).an_element()
Alternating form of degree 2 on the Rank-3 free module M over the Integer Ring
sage: M.dual_exterior_power(2).an_element().display()
e^0/\e^1
sage: M.dual_exterior_power(3)
3rd exterior power of the dual of the Rank-3 free module M over the Integer␣
↪Ring
sage: M.dual_exterior_power(3).an_element()
Alternating form of degree 3 on the Rank-3 free module M over the Integer Ring
sage: M.dual_exterior_power(3).an_element().display()
e^0/\e^1/\e^2
```

See *ExtPowerFreeModule* for more documentation.

endomorphism (*matrix_rep*, *basis=None*, *name=None*, *latex_name=None*)

Contruct an endomorphism of the free module `self` .

The returned object is a module morphism $\phi : M \to M$, where M is `self` .

INPUT:

- `matrix_rep` – matrix of size rank(M)*rank(M) representing the endomorphism with respect to `basis` ; this entry can actually be any material from which a matrix of elements of `self` base ring can be constructed; the *columns* of `matrix_rep` must be the components w.r.t. `basis` of the images of the elements of `basis` .
- `basis` – (default: `None`) basis of `self` defining the matrix representation; if None the default basis of `self` is assumed.
- `name` – (default: `None`) string; name given to the endomorphism
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the endomorphism; if none is provided, `name` will be used.

OUTPUT:

- the endomorphism $\phi : M \to M$ corresponding to the given specifications, as an instance of *FiniteRankFreeModuleMorphism*

EXAMPLES:

Construction of an endomorphism with minimal data (module's default basis and no name):

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M')
sage: e = M.basis('e')
sage: phi = M.endomorphism([[1,-2], [-3,4]]) ; phi
Generic endomorphism of Rank-2 free module M over the Integer Ring
sage: phi.matrix()  # matrix w.r.t the default basis
[ 1 -2]
[-3  4]
```

Construction with full list of arguments (matrix given a basis different from the default one):

```
sage: a = M.automorphism() ; a[0,1], a[1,0] = 1, -1
sage: ep = e.new_basis(a, 'ep', latex_symbol="e'")
sage: phi = M.endomorphism([[1,-2], [-3,4]], basis=ep, name='phi',
....:                      latex_name=r'\phi')
sage: phi
Generic endomorphism of Rank-2 free module M over the Integer Ring
sage: phi.matrix(ep)  # the input matrix
[ 1 -2]
[-3  4]
sage: phi.matrix()  # matrix w.r.t the default basis
[4 3]
[2 1]
```

See *FiniteRankFreeModuleMorphism* for more documentation.

general_linear_group ( )

Return the general linear group of `self` .

If `self` is the free module M, the *general linear group* is the group $\mathrm{GL}(M)$ of automorphisms of M.

OUTPUT:

- instance of class *FreeModuleLinearGroup* representing $\mathrm{GL}(M)$

EXAMPLES:

The general linear group of a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: GL = M.general_linear_group() ; GL
General linear group of the Rank-3 free module M over the Integer Ring
sage: GL.category()
Category of groups
sage: type(GL)
<class 'sage.tensor.modules.free_module_linear_group.FreeModuleLinearGroup_
↪with_category'>
```

There is a unique instance of the general linear group:

```
sage: M.general_linear_group() is GL
True
```

The group identity element:

```
sage: GL.one()
Identity map of the Rank-3 free module M over the Integer Ring
sage: GL.one().matrix(e)
[1 0 0]
[0 1 0]
[0 0 1]
```

An element:

```
sage: GL.an_element()
Automorphism of the Rank-3 free module M over the Integer Ring
sage: GL.an_element().matrix(e)
[ 1  0  0]
```

```
[ 0 -1  0]
[ 0  0  1]
```

See *FreeModuleLinearGroup* for more documentation.

hom (*codomain*, *matrix_rep*, *bases=None*, *name=None*, *latex_name=None*)
Homomorphism from `self` to a free module.

Define a module homomorphism

$$\phi : M \longrightarrow N,$$

where M is `self` and N is a free module of finite rank over the same ring R as `self` .

Note: This method is a redefinition of `sage.structure.parent.Parent.hom()` because the latter assumes that `self` has some privileged generators, while an instance of *FiniteRankFreeModule* has no privileged basis.

INPUT:

- `codomain` – the target module N
- `matrix_rep` – matrix of size rank(N)*rank(M) representing the homomorphism with respect to the pair of bases defined by `bases` ; this entry can actually be any material from which a matrix of elements of R can be constructed; the *columns* of `matrix_rep` must be the components w.r.t. `basis_N` of the images of the elements of `basis_M` .
- `bases` – (default: `None`) pair `(basis_M,basis_N)` defining the matrix representation, `basis_M` being a basis of `self` and `basis_N` a basis of module N ; if None the pair formed by the default bases of each module is assumed.
- `name` – (default: `None`) string; name given to the homomorphism
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the homomorphism; if None, `name` will be used.

OUTPUT:

- the homomorphism $\phi : M \to N$ corresponding to the given specifications, as an instance of *FiniteRankFreeModuleMorphism*

EXAMPLES:

Homomorphism between two free modules over **Z**:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: N = FiniteRankFreeModule(ZZ, 2, name='N')
sage: e = M.basis('e')
sage: f = N.basis('f')
sage: phi = M.hom(N, [[-1,2,0], [5,1,2]]) ; phi
Generic morphism:
  From: Rank-3 free module M over the Integer Ring
  To:   Rank-2 free module N over the Integer Ring
```

Homomorphism defined by a matrix w.r.t. bases that are not the default ones:

```
sage: ep = M.basis('ep', latex_symbol=r"e'")
sage: fp = N.basis('fp', latex_symbol=r"f'")
sage: phi = M.hom(N, [[3,2,1], [1,2,3]], bases=(ep, fp)) ; phi
```

```
Generic morphism:
  From: Rank-3 free module M over the Integer Ring
  To:   Rank-2 free module N over the Integer Ring
```

Call with all arguments specified:

```
sage: phi = M.hom(N, [[3,2,1], [1,2,3]], bases=(ep, fp),
....:             name='phi', latex_name=r'\phi')
```

The parent:

```
sage: phi.parent() is Hom(M,N)
True
```

See class *FiniteRankFreeModuleMorphism* for more documentation.

identity_map (*name='Id'*, *latex_name=None*)

Return the identity map of the free module `self` .

INPUT:

- `name` – (string; default: 'Id') name given to the identity identity map
- `latex_name` – (string; default: `None`) LaTeX symbol to denote the identity map; if none is provided, the LaTeX symbol is set to 'mathrm{Id}' if `name` is 'Id' and to `name` otherwise

OUTPUT:

- the identity map of `self` as an instance of *FreeModuleAutomorphism*

EXAMPLES:

Identity map of a rank-3 **Z**-module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: Id = M.identity_map() ; Id
Identity map of the Rank-3 free module M over the Integer Ring
sage: Id.parent()
General linear group of the Rank-3 free module M over the Integer Ring
sage: Id.matrix(e)
[1 0 0]
[0 1 0]
[0 0 1]
```

The default LaTeX symbol:

```
sage: latex(Id)
\mathrm{Id}
```

It can be changed by means of the method *set_name()* :

```
sage: Id.set_name(latex_name=r'\mathrm{1}_M')
sage: latex(Id)
\mathrm{1}_M
```

The identity map is actually the identity element of GL(M):

```
sage: Id is M.general_linear_group().one()
True
```

It is also a tensor of type-(1,1) on M:

```
sage: Id.tensor_type()
(1, 1)
sage: Id.comp(e)
Kronecker delta of size 3x3
sage: Id[:]
[1 0 0]
[0 1 0]
[0 0 1]
```

Example with a LaTeX symbol different from the default one and set at the creation of the object:

```
sage: N = FiniteRankFreeModule(ZZ, 3, name='N')
sage: f = N.basis('f')
sage: Id = N.identity_map(name='Id_N', latex_name=r'\mathrm{Id}_N')
sage: Id
Identity map of the Rank-3 free module N over the Integer Ring
sage: latex(Id)
\mathrm{Id}_N
```

irange (*start=None*)

Single index generator, labelling the elements of a basis of `self` .

INPUT:

- `start` – (default: `None`) integer; initial value of the index; if none is provided, `self._sindex` is assumed

OUTPUT:

- an iterable index, starting from `start` and ending at `self._sindex + self.rank() -1`

EXAMPLES:

Index range on a rank-3 module:

```
sage: M = FiniteRankFreeModule(ZZ, 3)
sage: list(M.irange())
[0, 1, 2]
sage: list(M.irange(start=1))
[1, 2]
```

The default starting value corresponds to the parameter `start_index` provided at the module construction (the default value being 0):

```
sage: M1 = FiniteRankFreeModule(ZZ, 3, start_index=1)
sage: list(M1.irange())
[1, 2, 3]
sage: M2 = FiniteRankFreeModule(ZZ, 3, start_index=-4)
sage: list(M2.irange())
[-4, -3, -2]
```

linear_form (*name=None*, *latex_name=None*)

Construct a linear form on the free module `self` .

A *linear form* on a free module M over a ring R is a map $M \to R$ that is linear. It can be viewed as a tensor of type $(0, 1)$ on M.

INPUT:

- `name` – (default: `None`) string; name given to the linear form
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the linear form; if none is provided, the LaTeX symbol is set to `name`

OUTPUT:

- instance of *FreeModuleAltForm*

EXAMPLES:

Linear form on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: a = M.linear_form('A') ; a
Linear form A on the Rank-3 free module M over the Integer Ring
sage: a[:] = [2,-1,3]  # components w.r.t. the module's default basis (e)
sage: a.display()
A = 2 e^0 - e^1 + 3 e^2
```

A linear form maps module elements to ring elements:

```
sage: v = M([1,1,1])
sage: a(v)
4
```

Test of linearity:

```
sage: u = M([-5,-2,7])
sage: a(3*u - 4*v) == 3*a(u) - 4*a(v)
True
```

See *FreeModuleAltForm* for more documentation.

print_bases()

Display the bases that have been defined on the free module `self`.

Use the method *bases()* to get the raw list of bases.

EXAMPLES:

Bases on a rank-4 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 4, name='M', start_index=1)
sage: M.print_bases()
No basis has been defined on the
 Rank-4 free module M over the Integer Ring
sage: e = M.basis('e')
sage: M.print_bases()
Bases defined on the Rank-4 free module M over the Integer Ring:
 - (e_1,e_2,e_3,e_4) (default basis)
sage: f = M.basis('f')
sage: M.print_bases()
Bases defined on the Rank-4 free module M over the Integer Ring:
 - (e_1,e_2,e_3,e_4) (default basis)
 - (f_1,f_2,f_3,f_4)
sage: M.set_default_basis(f)
sage: M.print_bases()
Bases defined on the Rank-4 free module M over the Integer Ring:
 - (e_1,e_2,e_3,e_4)
 - (f_1,f_2,f_3,f_4) (default basis)
```

rank ( )

Return the rank of the free module `self` .

Since the ring over which `self` is built is assumed to be commutative (and hence has the invariant basis number property), the rank is defined uniquely, as the cardinality of any basis of `self` .

EXAMPLES:

Rank of free modules over **Z**:

```
sage: M = FiniteRankFreeModule(ZZ, 3)
sage: M.rank()
3
sage: M.tensor_module(0,1).rank()
3
sage: M.tensor_module(0,2).rank()
9
sage: M.tensor_module(1,0).rank()
3
sage: M.tensor_module(1,1).rank()
9
sage: M.tensor_module(1,2).rank()
27
sage: M.tensor_module(2,2).rank()
81
```

set_change_of_basis (*basis1*, *basis2*, *change_of_basis*, *compute_inverse=True*)

Relates two bases by an automorphism of `self` .

This updates the internal dictionary `self._basis_changes` .

INPUT:

- `basis1` – basis 1, denoted (e_i) below
- `basis2` – basis 2, denoted (f_i) below
- `change_of_basis` – instance of class *FreeModuleAutomorphism* describing the automorphism P that relates the basis (e_i) to the basis (f_i) according to $f_i = P(e_i)$
- `compute_inverse` (default: `True`) – if set to `True` , the inverse automorphism is computed and the change from basis (f_i) to (e_i) is set to it in the internal dictionary `self._basis_changes`

EXAMPLES:

Defining a change of basis on a rank-2 free module:

```
sage: M = FiniteRankFreeModule(QQ, 2, name='M')
sage: e = M.basis('e')
sage: f = M.basis('f')
sage: a = M.automorphism()
sage: a[:] = [[1, 2], [-1, 3]]
sage: M.set_change_of_basis(e, f, a)
```

The change of basis and its inverse have been recorded:

```
sage: M.change_of_basis(e,f).matrix(e)
[ 1  2]
[-1  3]
sage: M.change_of_basis(f,e).matrix(e)
```

```
[ 3/5 -2/5]
[ 1/5  1/5]
```

and are effective:

```
sage: f[0].display(e)
f_0 = e_0 - e_1
sage: e[0].display(f)
e_0 = 3/5 f_0 + 1/5 f_1
```

set_default_basis (*basis*)

Sets the default basis of `self` .

The *default basis* is simply a basis whose name can be skipped in methods requiring a basis as an argument. By default, it is the first basis introduced on the module.

INPUT:

- `basis` – instance of *FreeModuleBasis* representing a basis on `self`

EXAMPLES:

Changing the default basis on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e') ; e
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
sage: f =  M.basis('f') ; f
Basis (f_1,f_2,f_3) on the Rank-3 free module M over the Integer Ring
sage: M.default_basis()
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
sage: M.set_default_basis(f)
sage: M.default_basis()
Basis (f_1,f_2,f_3) on the Rank-3 free module M over the Integer Ring
```

sym_bilinear_form (*name=None*, *latex_name=None*)

Construct a symmetric bilinear form on the free module `self` .

INPUT:

- `name` – (default: `None`) string; name given to the symmetric bilinear form
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the symmetric bilinear form; if none is provided, the LaTeX symbol is set to `name`

OUTPUT:

- instance of *FreeModuleTensor* of tensor type $(0, 2)$ and symmetric

EXAMPLES:

Symmetric bilinear form on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: a = M.sym_bilinear_form('A') ; a
Symmetric bilinear form A on the
 Rank-3 free module M over the Integer Ring
```

A symmetric bilinear form is a type-(0,2) tensor that is symmetric:

```
sage: a.parent()
Free module of type-(0,2) tensors on the
 Rank-3 free module M over the Integer Ring
sage: a.tensor_type()
(0, 2)
sage: a.tensor_rank()
2
sage: a.symmetries()
symmetry: (0, 1);  no antisymmetry
```

Components with respect to a given basis:

```
sage: e = M.basis('e')
sage: a[0,0], a[0,1], a[0,2] = 1, 2, 3
sage: a[1,1], a[1,2] = 4, 5
sage: a[2,2] = 6
```

Only independent components have been set; the other ones are deduced by symmetry:

```
sage: a[1,0], a[2,0], a[2,1]
(2, 3, 5)
sage: a[:]
[1 2 3]
[2 4 5]
[3 5 6]
```

A symmetric bilinear form acts on pairs of module elements:

```
sage: u = M([2,-1,3]) ; v = M([-2,4,1])
sage: a(u,v)
61
sage: a(v,u) == a(u,v)
True
```

The sum of two symmetric bilinear forms is another symmetric bilinear form:

```
sage: b = M.sym_bilinear_form('B')
sage: b[0,0], b[0,1], b[1,2] = -2, 1, -3
sage: s = a + b ; s
Symmetric bilinear form A+B on the
 Rank-3 free module M over the Integer Ring
sage: a[:], b[:], s[:]
(
[1 2 3]  [-2  1  0]  [-1  3  3]
[2 4 5]  [ 1  0 -3]  [ 3  4  2]
[3 5 6], [ 0 -3  0], [ 3  2  6]
)
```

Adding a symmetric bilinear from with a non-symmetric one results in a generic type-$(0,2)$ tensor:

```
sage: c = M.tensor((0,2), name='C')
sage: c[0,1] = 4
sage: s = a + c ; s
Type-(0,2) tensor A+C on the Rank-3 free module M over the Integer Ring
sage: s.symmetries()
no symmetry;  no antisymmetry
sage: s[:]
[1 6 3]
```

```
[2 4 5]
[3 5 6]
```

See *FreeModuleTensor* for more documentation.

tensor (*tensor_type*, *name=None*, *latex_name=None*, *sym=None*, *antisym=None*)

Construct a tensor on the free module `self` .

INPUT:

- `tensor_type` – pair `(k,l)` with `k` being the contravariant rank and `l` the covariant rank
- `name` – (default: `None`) string; name given to the tensor
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the tensor; if none is provided, the LaTeX symbol is set to `name`
- `sym` – (default: `None`) a symmetry or a list of symmetries among the tensor arguments: each symmetry is described by a tuple containing the positions of the involved arguments, with the convention `position = 0` for the first argument. For instance:
  - `sym = (0,1)` for a symmetry between the 1st and 2nd arguments
  - `sym = [(0,2),(1,3,4)]` for a symmetry between the 1st and 3rd arguments and a symmetry between the 2nd, 4th and 5th arguments.
- `antisym` – (default: `None`) antisymmetry or list of antisymmetries among the arguments, with the same convention as for `sym`

OUTPUT:

- instance of *FreeModuleTensor* representing the tensor defined on `self` with the provided characteristics

EXAMPLES:

Tensors on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: t = M.tensor((1,0), name='t') ; t
Element t of the Rank-3 free module M over the Integer Ring
sage: t = M.tensor((0,1), name='t') ; t
Linear form t on the Rank-3 free module M over the Integer Ring
sage: t = M.tensor((1,1), name='t') ; t
Type-(1,1) tensor t on the Rank-3 free module M over the Integer Ring
sage: t = M.tensor((0,2), name='t', sym=(0,1)) ; t
Symmetric bilinear form t on the
 Rank-3 free module M over the Integer Ring
sage: t = M.tensor((0,2), name='t', antisym=(0,1)) ; t
Alternating form t of degree 2 on the
 Rank-3 free module M over the Integer Ring
sage: t = M.tensor((1,2), name='t') ; t
Type-(1,2) tensor t on the Rank-3 free module M over the Integer Ring
```

See *FreeModuleTensor* for more examples and documentation.

tensor_from_comp (*tensor_type*, *comp*, *name=None*, *latex_name=None*)

Construct a tensor on `self` from a set of components.

The tensor symmetries are deduced from those of the components.

INPUT:

- `tensor_type` – pair `(k,l)` with `k` being the contravariant rank and `l` the covariant rank
- `comp` – instance of *Components* representing the tensor components in a given basis
- `name` – (default: `None`) string; name given to the tensor
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the tensor; if none is provided, the LaTeX symbol is set to `name`

OUTPUT:

- instance of *FreeModuleTensor* representing the tensor defined on `self` with the provided characteristics.

EXAMPLES:

Construction of a tensor of rank 1:

```
sage: from sage.tensor.modules.comp import Components, CompWithSym,␣
 ↪CompFullySym, CompFullyAntiSym
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: c = Components(ZZ, e, 1)
sage: c[:]
[0, 0, 0]
sage: c[:] = [-1,4,2]
sage: t = M.tensor_from_comp((1,0), c)
sage: t
Element of the Rank-3 free module M over the Integer Ring
sage: t.display(e)
-e_0 + 4 e_1 + 2 e_2
sage: t = M.tensor_from_comp((0,1), c) ; t
Linear form on the Rank-3 free module M over the Integer Ring
sage: t.display(e)
-e^0 + 4 e^1 + 2 e^2
```

Construction of a tensor of rank 2:

```
sage: c = CompFullySym(ZZ, e, 2)
sage: c[0,0], c[1,2] = 4, 5
sage: t = M.tensor_from_comp((0,2), c) ; t
Symmetric bilinear form on the
 Rank-3 free module M over the Integer Ring
sage: t.symmetries()
symmetry: (0, 1);  no antisymmetry
sage: t.display(e)
4 e^0*e^0 + 5 e^1*e^2 + 5 e^2*e^1
sage: c = CompFullyAntiSym(ZZ, e, 2)
sage: c[0,1], c[1,2] = 4, 5
sage: t = M.tensor_from_comp((0,2), c) ; t
Alternating form of degree 2 on the
 Rank-3 free module M over the Integer Ring
sage: t.display(e)
4 e^0/\e^1 + 5 e^1/\e^2
```

`tensor_module` (k, l)

Return the free module of all tensors of type (k, l) defined on `self`.

INPUT:

- `k` – non-negative integer; the contravariant rank, the tensor type being (k, l)

- `l` – non-negative integer; the covariant rank, the tensor type being (k, l)

OUTPUT:

- instance of *TensorFreeModule* representing the free module $T^{(k,l)}(M)$ of type-(k, l) tensors on the free module `self`

EXAMPLES:

Tensor modules over a free module over **Z**:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: T = M.tensor_module(1,2) ; T
Free module of type-(1,2) tensors on the Rank-3 free module M
 over the Integer Ring
sage: T.an_element()
Type-(1,2) tensor on the Rank-3 free module M over the Integer Ring
```

Tensor modules are unique:

```
sage: M.tensor_module(1,2) is T
True
```

The base module is itself the module of all type-$(1, 0)$ tensors:

```
sage: M.tensor_module(1,0) is M
True
```

See *TensorFreeModule* for more documentation.

zero ( )

Return the zero element of `self` .

EXAMPLES:

Zero elements of free modules over **Z**:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: M.zero()
Element zero of the Rank-3 free module M over the Integer Ring
sage: M.zero().parent() is M
True
sage: M.zero() is M(0)
True
sage: T = M.tensor_module(1,1)
sage: T.zero()
Type-(1,1) tensor zero on the Rank-3 free module M over the Integer Ring
sage: T.zero().parent() is T
True
sage: T.zero() is T(0)
True
```

Components of the zero element with respect to some basis:

```
sage: e = M.basis('e')
sage: M.zero()[e,:]
[0, 0, 0]
sage: all(M.zero()[e,i] == M.base_ring().zero() for i in M.irange())
True
sage: T.zero()[e,:]
[0 0 0]
```

```
[0 0 0]
[0 0 0]
sage: M.tensor_module(1,2).zero()[e,:]
[[[0, 0, 0], [0, 0, 0], [0, 0, 0]],
 [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
 [[0, 0, 0], [0, 0, 0], [0, 0, 0]]]
```

CHAPTER

TWO

FREE MODULE BASES

The class *FreeModuleBasis* implements bases on a free module M of finite rank over a commutative ring, while the class *FreeModuleCoBasis* implements the dual bases (i.e. bases of the dual module M^*).

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version

REFERENCES:

- Chap. 10 of R. Godement : *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)
- Chap. 3 of S. Lang : *Algebra*, 3rd ed., Springer (New York) (2002)

class sage.tensor.modules.free_module_basis. **Basis_abstract** (*fmodule*, *symbol*, *latex_symbol*, *latex_name*)

Bases: sage.structure.unique_representation.UniqueRepresentation , sage.structure.sage_object.SageObject

Abstract base class for (dual) bases of free modules.

free_module ( )

Return the free module of self .

EXAMPLES:

```
sage: M = FiniteRankFreeModule(QQ, 2, name='M', start_index=1)
sage: e = M.basis('e')
sage: e.free_module() is M
True
```

class sage.tensor.modules.free_module_basis. **FreeModuleBasis** (*fmodule*, *symbol*, *latex_symbol=None*)

Bases: *sage.tensor.modules.free_module_basis.Basis_abstract*

Basis of a free module over a commutative ring R.

INPUT:

- fmodule – free module M (as an instance of *FiniteRankFreeModule*)
- symbol – string; a letter (of a few letters) to denote a generic element of the basis
- latex_symbol – (default: None) string; symbol to denote a generic element of the basis; if None , the value of symbol is used

EXAMPLES:

A basis on a rank-3 free module over **Z**:

```
sage: M0 = FiniteRankFreeModule(ZZ, 3, name='M_0')
sage: from sage.tensor.modules.free_module_basis import FreeModuleBasis
sage: e = FreeModuleBasis(M0, 'e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M_0 over the Integer Ring
```

Instead of importing FreeModuleBasis in the global name space, it is recommended to use the module's method *basis()* :

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
```

The individual elements constituting the basis are accessed via the square bracket operator:

```
sage: e[0]
Element e_0 of the Rank-3 free module M over the Integer Ring
sage: e[0] in M
True
```

The LaTeX symbol can be set explicitely, as the second argument of *basis()* :

```
sage: latex(e)
\left(e_0,e_1,e_2\right)
sage: eps = M.basis('eps', r'\epsilon') ; eps
Basis (eps_0,eps_1,eps_2) on the Rank-3 free module M over the Integer
 Ring
sage: latex(eps)
\left(\epsilon_0,\epsilon_1,\epsilon_2\right)
```

The individual elements of the basis are labelled according the parameter `start_index` provided at the free module construction:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e') ; e
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
sage: e[1]
Element e_1 of the Rank-3 free module M over the Integer Ring
```

dual_basis ( )

Return the basis dual to `self` .

OUTPUT:

- instance of *FreeModuleCoBasis* representing the dual of `self`

EXAMPLES:

Dual basis on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e') ; e
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
sage: f = e.dual_basis() ; f
Dual basis (e^1,e^2,e^3) on the Rank-3 free module M over the Integer Ring
```

Let us check that the elements of f are elements of the dual of M:

```
sage: f[1] in M.dual()
True
```

```
sage: f[1]
Linear form e^1 on the Rank-3 free module M over the Integer Ring
```

and that f is indeed the dual of e:

```
sage: f[1](e[1]), f[1](e[2]), f[1](e[3])
(1, 0, 0)
sage: f[2](e[1]), f[2](e[2]), f[2](e[3])
(0, 1, 0)
sage: f[3](e[1]), f[3](e[2]), f[3](e[3])
(0, 0, 1)
```

new_basis (*change_of_basis*, *symbol*, *latex_symbol=None*)

Define a new module basis from `self` .

The new basis is defined by means of a module automorphism.

INPUT:

- `change_of_basis` – instance of *FreeModuleAutomorphism* describing the automorphism P that relates the current basis (e_i) (described by `self`) to the new basis (n_i) according to $n_i = P(e_i)$
- `symbol` – string; a letter (of a few letters) to denote a generic element of the basis
- `latex_symbol` – (default: `None`) string; symbol to denote a generic element of the basis; if `None` , the value of `symbol` is used

OUTPUT:

- the new basis (n_i), as an instance of *FreeModuleBasis*

EXAMPLES:

Change of basis on a vector space of dimension 2:

```
sage: M = FiniteRankFreeModule(QQ, 2, name='M', start_index=1)
sage: e = M.basis('e')
sage: a = M.automorphism()
sage: a[:] = [[1, 2], [-1, 3]]
sage: f = e.new_basis(a, 'f') ; f
Basis (f_1,f_2) on the 2-dimensional vector space M over the
 Rational Field
sage: f[1].display()
f_1 = e_1 - e_2
sage: f[2].display()
f_2 = 2 e_1 + 3 e_2
sage: e[1].display(f)
e_1 = 3/5 f_1 + 1/5 f_2
sage: e[2].display(f)
e_2 = -2/5 f_1 + 1/5 f_2
```

class sage.tensor.modules.free_module_basis. **FreeModuleCoBasis** (*basis*, *symbol*, *latex_symbol=None*)

Bases: *sage.tensor.modules.free_module_basis.Basis_abstract*

Dual basis of a free module over a commutative ring.

INPUT:

- `basis` – basis of a free module M of which `self` is the dual (must be an instance of *FreeModuleBasis*)

- `symbol` – a letter (of a few letters) to denote a generic element of the cobasis
- `latex_symbol` – (default: `None`) symbol to denote a generic element of the cobasis; if `None`, the value of `symbol` is used

EXAMPLES:

Dual basis on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e') ; e
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
sage: from sage.tensor.modules.free_module_basis import FreeModuleCoBasis
sage: f = FreeModuleCoBasis(e, 'f') ; f
Dual basis (f^1,f^2,f^3) on the Rank-3 free module M over the Integer Ring
```

Let us check that the elements of `f` are in the dual of `M`:

```
sage: f[1] in M.dual()
True
sage: f[1]
Linear form f^1 on the Rank-3 free module M over the Integer Ring
```

and that `f` is indeed the dual of `e`:

```
sage: f[1](e[1]), f[1](e[2]), f[1](e[3])
(1, 0, 0)
sage: f[2](e[1]), f[2](e[2]), f[2](e[3])
(0, 1, 0)
sage: f[3](e[1]), f[3](e[2]), f[3](e[3])
(0, 0, 1)
```

CHAPTER

THREE

TENSORS

3.1 Tensor products of free modules

The class `TensorFreeModule` implements tensor products of the type

$$T^{(k,l)}(M) = \underbrace{M \otimes \cdots \otimes M}_{k \text{ times}} \otimes \underbrace{M^* \otimes \cdots \otimes M^*}_{l \text{ times}},$$

where M is a free module of finite rank over a commutative ring R and $M^* = \mathrm{Hom}_R(M, R)$ is the dual of M. Note that $T^{(1,0)}(M) = M$ and $T^{(0,1)}(M) = M^*$.

Thanks to the canonical isomorphism $M^{**} \simeq M$ (which holds since M is a free module of finite rank), $T^{(k,l)}(M)$ can be identified with the set of tensors of type (k, l) defined as multilinear maps

$$\underbrace{M^* \times \cdots \times M^*}_{k \text{ times}} \times \underbrace{M \times \cdots \times M}_{l \text{ times}} \longrightarrow R$$

Accordingly, `TensorFreeModule` is a Sage *parent* class, whose *element* class is `FreeModuleTensor`.

$T^{(k,l)}(M)$ is itself a free module over R, of rank n^{k+l}, n being the rank of M. Accordingly the class `TensorFreeModule` inherits from the class `FiniteRankFreeModule`.

Todo

implement more general tensor products, i.e. tensor product of the type $M_1 \otimes \cdots \otimes M_n$, where the M_i's are n free modules of finite rank over the same ring R.

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version

REFERENCES:

- K. Conrad: *Tensor products*, http://www.math.uconn.edu/~kconrad/blurbs/
- Chap. 21 (Exer. 4) of R. Godement: *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)
- Chap. 16 of S. Lang: *Algebra*, 3rd ed., Springer (New York) (2002)

class sage.tensor.modules.tensor_free_module. **TensorFreeModule** (*fmodule*, *tensor_type*, *name=None*, *latex_name=None*)

Bases: `sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule`

Class for the free modules over a commutative ring R that are tensor products of a given free module M over R with itself and its dual M^*:

$$T^{(k,l)}(M) = \underbrace{M \otimes \cdots \otimes M}_{k \text{ times}} \otimes \underbrace{M^* \otimes \cdots \otimes M^*}_{l \text{ times}}$$

As recalled above, $T^{(k,l)}(M)$ can be canonically identified with the set of tensors of type (k, l) on M.

This is a Sage *parent* class, whose *element* class is `FreeModuleTensor`.

INPUT:

- `fmodule` – free module M of finite rank over a commutative ring R, as an instance of `FiniteRankFreeModule`
- `tensor_type` – pair `(k,l)` with `k` being the contravariant rank and `l` the covariant rank
- `name` – (default: `None`) string; name given to the tensor module
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the tensor module; if none is provided, it is set to `name`

EXAMPLES:

Set of tensors of type $(1, 2)$ on a free $\mathbf{Z}$-module of rank 3:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: from sage.tensor.modules.tensor_free_module import TensorFreeModule
sage: T = TensorFreeModule(M, (1,2)) ; T
Free module of type-(1,2) tensors on the
 Rank-3 free module M over the Integer Ring
```

Instead of importing TensorFreeModule in the global name space, it is recommended to use the module's method `tensor_module()`:

```
sage: T = M.tensor_module(1,2) ; T
Free module of type-(1,2) tensors on the
 Rank-3 free module M over the Integer Ring
sage: latex(T)
T^{(1, 2)}\left(M\right)
```

The module `M` itself is considered as the set of tensors of type $(1, 0)$:

```
sage: M is M.tensor_module(1,0)
True
```

`T` is a module (actually a free module) over $\mathbf{Z}$:

```
sage: T.category()
Category of finite dimensional modules over Integer Ring
sage: T in Modules(ZZ)
True
sage: T.rank()
27
sage: T.base_ring()
Integer Ring
sage: T.base_module()
Rank-3 free module M over the Integer Ring
```

`T` is a *parent* object, whose elements are instances of `FreeModuleTensor`:

```
sage: t = T.an_element() ; t
Type-(1,2) tensor on the Rank-3 free module M over the Integer Ring
sage: from sage.tensor.modules.free_module_tensor import FreeModuleTensor
sage: isinstance(t, FreeModuleTensor)
True
sage: t in T
True
sage: T.is_parent_of(t)
True
```

Elements can be constructed from `T` . In particular, 0 yields the zero element of `T` :

```
sage: T(0)
Type-(1,2) tensor zero on the Rank-3 free module M over the Integer Ring
sage: T(0) is T.zero()
True
```

while non-zero elements are constructed by providing their components in a given basis:

```
sage: e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: comp = [[[i-j+k for k in range(3)] for j in range(3)] for i in range(3)]
sage: t = T(comp, basis=e, name='t') ; t
Type-(1,2) tensor t on the Rank-3 free module M over the Integer Ring
sage: t.comp(e)[:]
[[[0, 1, 2], [-1, 0, 1], [-2, -1, 0]],
 [[1, 2, 3], [0, 1, 2], [-1, 0, 1]],
 [[2, 3, 4], [1, 2, 3], [0, 1, 2]]]
sage: t.display(e)
t = e_0*e^0*e^1 + 2 e_0*e^0*e^2 - e_0*e^1*e^0 + e_0*e^1*e^2
 - 2 e_0*e^2*e^0 - e_0*e^2*e^1 + e_1*e^0*e^0 + 2 e_1*e^0*e^1
 + 3 e_1*e^0*e^2 + e_1*e^1*e^1 + 2 e_1*e^1*e^2 - e_1*e^2*e^0
 + e_1*e^2*e^2 + 2 e_2*e^0*e^0 + 3 e_2*e^0*e^1 + 4 e_2*e^0*e^2
 + e_2*e^1*e^0 + 2 e_2*e^1*e^1 + 3 e_2*e^1*e^2 + e_2*e^2*e^1
 + 2 e_2*e^2*e^2
```

An alternative is to construct the tensor from an empty list of components and to set the nonzero components afterwards:

```
sage: t = T([], name='t')
sage: t.set_comp(e)[0,1,1] = -3
sage: t.set_comp(e)[2,0,1] = 4
sage: t.display(e)
t = -3 e_0*e^1*e^1 + 4 e_2*e^0*e^1
```

See the documentation of *FreeModuleTensor* for the full list of arguments that can be provided to the __call__ operator. For instance, to contruct a tensor symmetric with respect to the last two indices:

```
sage: t = T([], name='t', sym=(1,2))
sage: t.set_comp(e)[0,1,1] = -3
sage: t.set_comp(e)[2,0,1] = 4
sage: t.display(e)  # notice that t^2_{10} has be set equal to t^2_{01} by␣
↪symmetry
t = -3 e_0*e^1*e^1 + 4 e_2*e^0*e^1 + 4 e_2*e^1*e^0
```

The tensor modules over a given module M are unique:

```
sage: T is M.tensor_module(1,2)
True
```

There is a coercion map from $\Lambda^p(M^*)$, the set of alternating forms of degree p, to $T^{(0,p)}(M)$:

```
sage: L2 = M.dual_exterior_power(2) ; L2
2nd exterior power of the dual of the Rank-3 free module M over the
 Integer Ring
sage: T02 = M.tensor_module(0,2) ; T02
Free module of type-(0,2) tensors on the Rank-3 free module M over the
 Integer Ring
sage: T02.has_coerce_map_from(L2)
True
```

Of course, for $p \geq 2$, there is no coercion in the reverse direction, since not every tensor of type (0,p) is alternating:

```
sage: L2.has_coerce_map_from(T02)
False
```

The coercion map $\Lambda^2(M^*) \to T^{(0,2)}(M)$ in action:

```
sage: a = M.alternating_form(2, name='a') ; a
Alternating form a of degree 2 on the Rank-3 free module M over the
 Integer Ring
sage: a[0,1], a[1,2] = 4, -3
sage: a.display(e)
a = 4 e^0/\e^1 - 3 e^1/\e^2
sage: a.parent() is L2
True
sage: ta = T02(a) ; ta
Type-(0,2) tensor a on the Rank-3 free module M over the Integer Ring
sage: ta.display(e)
a = 4 e^0*e^1 - 4 e^1*e^0 - 3 e^1*e^2 + 3 e^2*e^1
sage: ta.symmetries() # the antisymmetry is of course preserved
no symmetry;  antisymmetry: (0, 1)
```

For the degree $p = 1$, there is a coercion in both directions:

```
sage: L1 = M.dual_exterior_power(1) ; L1
Dual of the Rank-3 free module M over the Integer Ring
sage: T01 = M.tensor_module(0,1) ; T01
Free module of type-(0,1) tensors on the Rank-3 free module M over the
 Integer Ring
sage: T01.has_coerce_map_from(L1)
True
sage: L1.has_coerce_map_from(T01)
True
```

The coercion map $\Lambda^1(M^*) \to T^{(0,1)}(M)$ in action:

```
sage: a = M.linear_form('a')
sage: a[:] = -2, 4, 1 ; a.display(e)
a = -2 e^0 + 4 e^1 + e^2
sage: a.parent() is L1
True
sage: ta = T01(a) ; ta
Type-(0,1) tensor a on the Rank-3 free module M over the Integer Ring
```

```
sage: ta.display(e)
a = -2 e^0 + 4 e^1 + e^2
```

The coercion map $T^{(0,1)}(M) \to \Lambda^1(M^*)$ in action:

```
sage: ta.parent() is T01
True
sage: lta = L1(ta) ; lta
Linear form a on the Rank-3 free module M over the Integer Ring
sage: lta.display(e)
a = -2 e^0 + 4 e^1 + e^2
sage: lta == a
True
```

There is a canonical identification between tensors of type (1,1) and endomorphisms of module M. Accordingly, coercion maps have been implemented between $T^{(1,1)}(M)$ and $\mathrm{End}(M)$ (the module of all endomorphisms of M, see *FreeModuleHomset*):

```
sage: T11 = M.tensor_module(1,1) ; T11
Free module of type-(1,1) tensors on the Rank-3 free module M over the
 Integer Ring
sage: End(M)
Set of Morphisms from Rank-3 free module M over the Integer Ring
 to Rank-3 free module M over the Integer Ring
 in Category of finite dimensional modules over Integer Ring
sage: T11.has_coerce_map_from(End(M))
True
sage: End(M).has_coerce_map_from(T11)
True
```

The coercion map $\mathrm{End}(M) \to T^{(1,1)}(M)$ in action:

```
sage: phi = End(M).an_element() ; phi
Generic endomorphism of Rank-3 free module M over the Integer Ring
sage: phi.matrix(e)
[1 1 1]
[1 1 1]
[1 1 1]
sage: tphi = T11(phi) ; tphi # image of phi by the coercion map
Type-(1,1) tensor on the Rank-3 free module M over the Integer Ring
sage: tphi[:]
[1 1 1]
[1 1 1]
[1 1 1]
sage: t = M.tensor((1,1))
sage: t[0,0], t[1,1], t[2,2] = -1,-2,-3
sage: t[:]
[-1  0  0]
[ 0 -2  0]
[ 0  0 -3]
sage: s = t + phi ; s  # phi is coerced to a type-(1,1) tensor prior to the␣
↪addition
Type-(1,1) tensor on the Rank-3 free module M over the Integer Ring
sage: s[:]
[ 0  1  1]
[ 1 -1  1]
[ 1  1 -2]
```

The coercion map $T^{(1,1)}(M) \to \mathrm{End}(M)$ in action:

```
sage: phi1 = End(M)(tphi) ; phi1
Generic endomorphism of Rank-3 free module M over the Integer Ring
sage: phi1 == phi
True
sage: s = phi + t ; s  # t is coerced to an endomorphism prior to the addition
Generic endomorphism of Rank-3 free module M over the Integer Ring
sage: s.matrix(e)
[ 0  1  1]
[ 1 -1  1]
[ 1  1 -2]
```

There is a coercion $\mathrm{GL}(M) \to T^{(1,1)}(M)$, i.e. from automorphisms of M to type-(1,1) tensors on M:

```
sage: GL = M.general_linear_group() ; GL
General linear group of the Rank-3 free module M over the Integer Ring
sage: T11.has_coerce_map_from(GL)
True
```

The coercion map $\mathrm{GL}(M) \to T^{(1,1)}(M)$ in action:

```
sage: a = GL.an_element() ; a
Automorphism of the Rank-3 free module M over the Integer Ring
sage: a.matrix(e)
[ 1  0  0]
[ 0 -1  0]
[ 0  0  1]
sage: ta = T11(a) ; ta
Type-(1,1) tensor on the Rank-3 free module M over the Integer Ring
sage: ta.display(e)
e_0*e^0 - e_1*e^1 + e_2*e^2
sage: a.display(e)
e_0*e^0 - e_1*e^1 + e_2*e^2
```

Of course, there is no coercion in the reverse direction, since not every type-(1,1) tensor is invertible:

```
sage: GL.has_coerce_map_from(T11)
False
```

Element
alias of `FreeModuleTensor`

base_module ( )
Return the free module on which `self` is constructed.

OUTPUT:

- instance of *FiniteRankFreeModule* representing the free module on which the tensor module is defined.

EXAMPLE:

Base module of a type-(1,2) tensor module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: T = M.tensor_module(1,2)
sage: T.base_module()
Rank-3 free module M over the Integer Ring
```

```
sage: T.base_module() is M
True
```

tensor_type ( )

Return the tensor type of `self` .

OUTPUT:

- pair (k, l) such that `self` is the module tensor product $T^{(k,l)}(M)$

EXAMPLE:

```
sage: M = FiniteRankFreeModule(ZZ, 3)
sage: T = M.tensor_module(1,2)
sage: T.tensor_type()
(1, 2)
```

3.2 Tensors on free modules

The class *FreeModuleTensor* implements tensors on a free module M of finite rank over a commutative ring. A *tensor of type* (k, l) on M is a multilinear map:

$$\underbrace{M^* \times \cdots \times M^*}_{k \text{ times}} \times \underbrace{M \times \cdots \times M}_{l \text{ times}} \longrightarrow R$$

where R is the commutative ring over which the free module M is defined and $M^* = \mathrm{Hom}_R(M, R)$ is the dual of M. The integer $k + l$ is called the *tensor rank*. The set $T^{(k,l)}(M)$ of tensors of type (k, l) on M is a free module of finite rank over R, described by the class *TensorFreeModule* .

Various derived classes of *FreeModuleTensor* are devoted to specific tensors:

- *FiniteRankFreeModuleElement* for elements of M, considered as type-(1,0) tensors thanks to the canonical identification $M^{**} = M$ (which holds since M is a free module of finite rank);
- *FreeModuleAltForm* for fully antisymmetric type-$(0, l)$ tensors (alternating forms);
- *FreeModuleAutomorphism* for type-(1,1) tensors representing invertible endomorphisms.

Each of these classes is a Sage *element* class, the corresponding *parent* classes being:

- for *FreeModuleTensor* : *TensorFreeModule*
- for *FiniteRankFreeModuleElement* : *FiniteRankFreeModule*
- for *FreeModuleAltForm* : *ExtPowerFreeModule*
- for *FreeModuleAutomorphism* : *FreeModuleLinearGroup*

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version

REFERENCES:

- Chap. 21 of R. Godement: *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)
- Chap. 12 of J. M. Lee: *Introduction to Smooth Manifolds*, 2nd ed., Springer (New York) (2013) (only when the free module is a vector space)
- Chap. 2 of B. O'Neill: *Semi-Riemannian Geometry*, Academic Press (San Diego) (1983)

EXAMPLES:

A tensor of type $(1,1)$ on a rank-3 free module over $\mathbf{Z}$:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: t = M.tensor((1,1), name='t') ; t
Type-(1,1) tensor t on the Rank-3 free module M over the Integer Ring
sage: t.parent()
Free module of type-(1,1) tensors on the Rank-3 free module M
 over the Integer Ring
sage: t.parent() is M.tensor_module(1,1)
True
sage: t in M.tensor_module(1,1)
True
```

Setting some component of the tensor in a given basis:

```
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: t.set_comp(e)[0,0] = -3  # the component [0,0] w.r.t. basis e is set to -3
```

The unset components are assumed to be zero:

```
sage: t.comp(e)[:]  # list of all components w.r.t. basis e
[-3  0  0]
[ 0  0  0]
[ 0  0  0]
sage: t.display(e)  # displays the expansion of t on the basis e_i*e^j of T^(1,1)(M)
t = -3 e_0*e^0
```

The commands `t.set_comp(e)` and `t.comp(e)` can be abridged by providing the basis as the first argument in the square brackets:

```
sage: t[e,0,0] = -3
sage: t[e,:]
[-3  0  0]
[ 0  0  0]
[ 0  0  0]
```

Actually, since `e` is `M`'s default basis, the mention of `e` can be omitted:

```
sage: t[0,0] = -3
sage: t[:]
[-3  0  0]
[ 0  0  0]
[ 0  0  0]
```

For tensors of rank 2, the matrix of components w.r.t. a given basis is obtained via the function `matrix`:

```
sage: matrix(t.comp(e))
[-3  0  0]
[ 0  0  0]
[ 0  0  0]
sage: matrix(t.comp(e)).parent()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

Tensor components can be modified (reset) at any time:

```
sage: t[0,0] = 0
sage: t[:]
[0 0 0]
[0 0 0]
[0 0 0]
```

Checking that `t` is zero:

```
sage: t.is_zero()
True
sage: t == 0
True
sage: t == M.tensor_module(1,1).zero()  # the zero element of the module of all type-
↪(1,1) tensors on M
True
```

The components are managed by the class *Components*:

```
sage: type(t.comp(e))
<class 'sage.tensor.modules.comp.Components'>
```

Only non-zero components are actually stored, in the dictionary `_comp` of class *Components*, whose keys are the indices:

```
sage: t.comp(e)._comp
{}
sage: t.set_comp(e)[0,0] = -3 ; t.set_comp(e)[1,2] = 2
sage: t.comp(e)._comp  # random output order (dictionary)
{(0, 0): -3, (1, 2): 2}
sage: t.display(e)
t = -3 e_0*e^0 + 2 e_1*e^2
```

Further tests of the comparison operator:

```
sage: t.is_zero()
False
sage: t == 0
False
sage: t == M.tensor_module(1,1).zero()
False
sage: t1 = t.copy()
sage: t1 == t
True
sage: t1[2,0] = 4
sage: t1 == t
False
```

As a multilinear map $M^* \times M \to \mathbf{Z}$, the type-$(1,1)$ tensor `t` acts on pairs formed by a linear form and a module element:

```
sage: a = M.linear_form(name='a') ; a[:] = (2, 1, -3) ; a
Linear form a on the Rank-3 free module M over the Integer Ring
sage: b = M([1,-6,2], name='b') ; b
Element b of the Rank-3 free module M over the Integer Ring
sage: t(a,b)
-2
```

class sage.tensor.modules.free_module_tensor. **FiniteRankFreeModuleElement** (*fmodule*, *name=None*, *latex_name=None*)

Bases: *sage.tensor.modules.free_module_tensor.FreeModuleTensor*

Element of a free module of finite rank over a commutative ring.

This is a Sage *element* class, the corresponding *parent* class being *FiniteRankFreeModule* .

The class *FiniteRankFreeModuleElement* inherits from *FreeModuleTensor* because the elements of a free module M of finite rank over a commutative ring R are identified with tensors of type $(1,0)$ on M via the canonical map

$$\begin{array}{rccccc} \Phi: & M & \longrightarrow & M^{**} & & \\ & v & \longmapsto & \bar{v}: & M^* & \longrightarrow & R \\ & & & & a & \longmapsto & a(v) \end{array}$$

Note that for free modules of finite rank, this map is actually an isomorphism, enabling the canonical identification: $M^{**} = M$.

INPUT:

- `fmodule` – free module M of finite rank over a commutative ring R, as an instance of *FiniteRankFreeModule*
- `name` – (default: `None`) name given to the element
- `latex_name` – (default: `None`) LaTeX symbol to denote the element; if none is provided, the LaTeX symbol is set to `name`

EXAMPLES:

Let us consider a rank-3 free module M over $\mathbf{Z}$:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
```

There are three ways to construct an element of the free module M: the first one (recommended) is using the free module:

```
sage: v = M([2,0,-1], basis=e, name='v') ; v
Element v of the Rank-3 free module M over the Integer Ring
sage: v.display()  # expansion on the default basis (e)
v = 2 e_0 - e_2
sage: v.parent() is M
True
```

The second way is to construct a tensor of type $(1,0)$ on M (cf. the canonical identification $M^{**} = M$ recalled above):

```
sage: v2 = M.tensor((1,0), name='v')
sage: v2[0], v2[2] = 2, -1 ; v2
Element v of the Rank-3 free module M over the Integer Ring
sage: v2.display()
v = 2 e_0 - e_2
sage: v2 == v
True
```

Finally, the third way is via some linear combination of the basis elements:

```
sage: v3 = 2*e[0] - e[2]
sage: v3.set_name('v') ; v3 # in this case, the name has to be set separately
Element v of the Rank-3 free module M over the Integer Ring
sage: v3.display()
v = 2 e_0 - e_2
sage: v3 == v
True
```

The canonical identification $M^{**} = M$ is implemented by letting the module elements act on linear forms, providing the same result as the reverse operation (cf. the map Φ defined above):

```
sage: a = M.linear_form(name='a')
sage: a[:] = (2, 1, -3) ; a
Linear form a on the Rank-3 free module M over the Integer Ring
sage: v(a)
7
sage: a(v)
7
sage: a(v) == v(a)
True
```

ARITHMETIC EXAMPLES

Addition:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: a = M([0,1,3], name='a') ; a
Element a of the Rank-3 free module M over the Integer Ring
sage: a.display()
a = e_1 + 3 e_2
sage: b = M([2,-2,1], name='b') ; b
Element b of the Rank-3 free module M over the Integer Ring
sage: b.display()
b = 2 e_0 - 2 e_1 + e_2
sage: s = a + b ; s
Element a+b of the Rank-3 free module M over the Integer Ring
sage: s.display()
a+b = 2 e_0 - e_1 + 4 e_2
sage: all(s[i] == a[i] + b[i] for i in M.irange())
True
```

Subtraction:

```
sage: s = a - b ; s
Element a-b of the Rank-3 free module M over the Integer Ring
sage: s.display()
a-b = -2 e_0 + 3 e_1 + 2 e_2
sage: all(s[i] == a[i] - b[i] for i in M.irange())
True
```

Multiplication by a scalar:

```
sage: s = 2*a ; s
Element of the Rank-3 free module M over the Integer Ring
```

```
sage: s.display()
2 e_1 + 6 e_2
sage: a.display()
a = e_1 + 3 e_2
```

Tensor product:

```
sage: s = a*b ; s
Type-(2,0) tensor a*b on the Rank-3 free module M over the Integer Ring
sage: s.symmetries()
no symmetry;  no antisymmetry
sage: s[:]
[ 0  0  0]
[ 2 -2  1]
[ 6 -6  3]
sage: s = a*s ; s
Type-(3,0) tensor a*a*b on the Rank-3 free module M over the Integer Ring
sage: s[:]
[[[0, 0, 0], [0, 0, 0], [0, 0, 0]],
 [[0, 0, 0], [2, -2, 1], [6, -6, 3]],
 [[0, 0, 0], [6, -6, 3], [18, -18, 9]]]
```

class sage.tensor.modules.free_module_tensor. **FreeModuleTensor** (*fmodule*, *tensor_type*, *name=None*, *latex_name=None*, *sym=None*, *antisym=None*, *parent=None*)

Bases: sage.structure.element.ModuleElement

Tensor over a free module of finite rank over a commutative ring.

This is a Sage *element* class, the corresponding *parent* class being *TensorFreeModule*.

INPUT:

- fmodule – free module M of finite rank over a commutative ring R, as an instance of *FiniteRankFreeModule*
- tensor_type – pair (k,l) with k being the contravariant rank and l the covariant rank
- name – (default: None) name given to the tensor
- latex_name – (default: None) LaTeX symbol to denote the tensor; if none is provided, the LaTeX symbol is set to name
- sym – (default: None) a symmetry or a list of symmetries among the tensor arguments: each symmetry is described by a tuple containing the positions of the involved arguments, with the convention position=0 for the first argument. For instance:
  - sym = (0,1) for a symmetry between the 1st and 2nd arguments;
  - sym = [(0,2),(1,3,4)] for a symmetry between the 1st and 3rd arguments and a symmetry between the 2nd, 4th and 5th arguments.
- antisym – (default: None) antisymmetry or list of antisymmetries among the arguments, with the same convention as for sym
- parent – (default: None) some specific parent (e.g. exterior power for alternating forms); if None, fmodule.tensor_module(k,l) is used

EXAMPLES:

A tensor of type $(1,1)$ on a rank-3 free module over $\mathbf{Z}$:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: t = M.tensor((1,1), name='t') ; t
Type-(1,1) tensor t on the Rank-3 free module M over the Integer Ring
```

Tensors are *Element* objects whose parents are tensor free modules:

```
sage: t.parent()
Free module of type-(1,1) tensors on the
 Rank-3 free module M over the Integer Ring
sage: t.parent() is M.tensor_module(1,1)
True
```

add_comp (*basis=None*)

Return the components of `self` w.r.t. a given module basis for assignment, keeping the components w.r.t. other bases.

To delete the components w.r.t. other bases, use the method *set_comp()* instead.

INPUT:

- `basis` – (default: `None`) basis in which the components are defined; if none is provided, the components are assumed to refer to the module's default basis

Warning: If the tensor has already components in other bases, it is the user's responsability to make sure that the components to be added are consistent with them.

OUTPUT:

- components in the given basis, as an instance of the class *Components*; if such components did not exist previously, they are created

EXAMPLES:

Setting components of a type-(1,1) tensor:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: t = M.tensor((1,1), name='t')
sage: t.add_comp()[0,1] = -3
sage: t.display()
t = -3 e_0*e^1
sage: t.add_comp()[1,2] = 2
sage: t.display()
t = -3 e_0*e^1 + 2 e_1*e^2
sage: t.add_comp(e)
2-indices components w.r.t. Basis (e_0,e_1,e_2) on the
 Rank-3 free module M over the Integer Ring
```

Adding components in a new basis:

```
sage: f =  M.basis('f')
sage: t.add_comp(f)[0,1] = 4
```

The components w.r.t. basis e have been kept:

```
sage: t._components.keys() # # random output (dictionary keys)
[Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring,
 Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring]
sage: t.display(f)
t = 4 f_0*f^1
sage: t.display(e)
t = -3 e_0*e^1 + 2 e_1*e^2
```

antisymmetrize (**pos*, ***kwargs*)

Antisymmetrization over some arguments.

INPUT:

- `pos` – list of argument positions involved in the antisymmetrization (with the convention `position=0` for the first argument); if none, the antisymmetrization is performed over all the arguments
- `basis` – (default: `None`) module basis with respect to which the component computation is to be performed; if none, the module's default basis is used if the tensor field has already components in it; otherwise another basis w.r.t. which the tensor has components will be picked

OUTPUT:

- the antisymmetrized tensor (instance of *FreeModuleTensor*)

EXAMPLES:

Antisymmetrization of a tensor of type $(2,0)$:

```
sage: M = FiniteRankFreeModule(QQ, 3, name='M')
sage: e = M.basis('e')
sage: t = M.tensor((2,0))
sage: t[:] = [[1,-2,3], [4,5,6], [7,8,-9]]
sage: s = t.antisymmetrize() ; s
Type-(2,0) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s.symmetries()
no symmetry;  antisymmetry: (0, 1)
sage: t[:], s[:]
(
[ 1 -2  3]  [ 0 -3 -2]
[ 4  5  6]  [ 3  0 -1]
[ 7  8 -9], [ 2  1  0]
)
sage: all(s[i,j] == 1/2*(t[i,j]-t[j,i])    # Check:
....:     for i in range(3) for j in range(3))
True
sage: s.antisymmetrize() == s  # another test
True
sage: t.antisymmetrize() == t.antisymmetrize(0,1)
True
```

Antisymmetrization of a tensor of type $(0,3)$ over the first two arguments:

```
sage: t = M.tensor((0,3))
sage: t[:] = [[[1,2,3], [-4,5,6], [7,8,-9]],
....:         [[10,-11,12], [13,14,-15], [16,17,18]],
....:         [[19,-20,-21], [-22,23,24], [25,26,-27]]]
sage: s = t.antisymmetrize(0,1) ; s  # (0,1) = the first two arguments
Type-(0,3) tensor on the 3-dimensional vector space M over the
```

```
 Rational Field
sage: s.symmetries()
no symmetry;  antisymmetry: (0, 1)
sage: s[:]
[[[0, 0, 0], [-7, 8, -3], [-6, 14, 6]],
 [[7, -8, 3], [0, 0, 0], [19, -3, -3]],
 [[6, -14, -6], [-19, 3, 3], [0, 0, 0]]]
sage: all(s[i,j,k] == 1/2*(t[i,j,k]-t[j,i,k])    # Check:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s.antisymmetrize(0,1) == s  # another test
True
sage: s.symmetrize(0,1) == 0  # of course
True
```

Instead of invoking the method *antisymmetrize()* , one can use the index notation with square brackets denoting the antisymmetrization; it suffices to pass the indices as a string inside square brackets:

```
sage: s1 = t['_[ij]k'] ; s1
Type-(0,3) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s1.symmetries()
no symmetry;  antisymmetry: (0, 1)
sage: s1 == s
True
```

The LaTeX notation is recognized:

```
sage: t['_{[ij]k}'] == s
True
```

Note that in the index notation, the name of the indices is irrelevant; they can even be replaced by dots:

```
sage: t['_[..].'] == s
True
```

Antisymmetrization of a tensor of type (0,3) over the first and last arguments:

```
sage: s = t.antisymmetrize(0,2) ; s  # (0,2) = first and last arguments
Type-(0,3) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s.symmetries()
no symmetry;  antisymmetry: (0, 2)
sage: s[:]
[[[0, -4, -8], [0, -4, 14], [0, -4, -17]],
 [[4, 0, 16], [4, 0, -19], [4, 0, -4]],
 [[8, -16, 0], [-14, 19, 0], [17, 4, 0]]]
sage: all(s[i,j,k] == 1/2*(t[i,j,k]-t[k,j,i])    # Check:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s.antisymmetrize(0,2) == s  # another test
True
sage: s.symmetrize(0,2) == 0  # of course
True
sage: s.symmetrize(0,1) == 0  # no reason for this to hold
False
```

Antisymmetrization of a tensor of type $(0, 3)$ over the last two arguments:

```
sage: s = t.antisymmetrize(1,2) ; s  # (1,2) = the last two arguments
Type-(0,3) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s.symmetries()
no symmetry;  antisymmetry: (1, 2)
sage: s[:]
[[[0, 3, -2], [-3, 0, -1], [2, 1, 0]],
 [[0, -12, -2], [12, 0, -16], [2, 16, 0]],
 [[0, 1, -23], [-1, 0, -1], [23, 1, 0]]]
sage: all(s[i,j,k] == 1/2*(t[i,j,k]-t[i,k,j])    # Check:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s.antisymmetrize(1,2) == s  # another test
True
sage: s.symmetrize(1,2) == 0  # of course
True
```

The index notation can be used instead of the explicit call to *antisymmetrize()* :

```
sage: t['_i[jk]'] == t.antisymmetrize(1,2)
True
```

Full antisymmetrization of a tensor of type (0,3):

```
sage: s = t.antisymmetrize() ; s
Alternating form of degree 3 on the 3-dimensional vector space M
 over the Rational Field
sage: s.symmetries()
no symmetry;  antisymmetry: (0, 1, 2)
sage: s[:]
[[[0, 0, 0], [0, 0, 2/3], [0, -2/3, 0]],
 [[0, 0, -2/3], [0, 0, 0], [2/3, 0, 0]],
 [[0, 2/3, 0], [-2/3, 0, 0], [0, 0, 0]]]
sage: all(s[i,j,k] == 1/6*(t[i,j,k]-t[i,k,j]+t[j,k,i]-t[j,i,k]
....:                      +t[k,i,j]-t[k,j,i])
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s.antisymmetrize() == s  # another test
True
sage: s.symmetrize(0,1) == 0  # of course
True
sage: s.symmetrize(0,2) == 0  # of course
True
sage: s.symmetrize(1,2) == 0  # of course
True
sage: t.antisymmetrize() == t.antisymmetrize(0,1,2)
True
```

The index notation can be used instead of the explicit call to *antisymmetrize()* :

```
sage: t['_[ijk]'] == t.antisymmetrize()
True
sage: t['_[abc]'] == t.antisymmetrize()
True
sage: t['_[...]'] == t.antisymmetrize()
True
sage: t['_{[ijk]}'] == t.antisymmetrize() # LaTeX notation
True
```

Antisymmetrization can be performed only on arguments on the same type:

```
sage: t = M.tensor((1,2))
sage: t[:] = [[[1,2,3], [-4,5,6], [7,8,-9]],
....:         [[10,-11,12], [13,14,-15], [16,17,18]],
....:         [[19,-20,-21], [-22,23,24], [25,26,-27]]]
sage: s = t.antisymmetrize(0,1)
Traceback (most recent call last):
...
TypeError: 0 is a contravariant position, while 1 is a covariant position;
antisymmetrization is meaningfull only on tensor arguments of the same type
sage: s = t.antisymmetrize(1,2) # OK: both 1 and 2 are covariant positions
```

The order of positions does not matter:

```
sage: t.antisymmetrize(2,1) == t.antisymmetrize(1,2)
True
```

Again, the index notation can be used:

```
sage: t['^i_[jk]'] == t.antisymmetrize(1,2)
True
sage: t['^i_{[jk]}'] == t.antisymmetrize(1,2)  # LaTeX notation
True
```

The character '^' can be skipped:

```
sage: t['i_[jk]'] == t.antisymmetrize(1,2)
True
```

base_module ( )

Return the module on which `self` is defined.

OUTPUT:

- instance of *FiniteRankFreeModule* representing the free module on which the tensor is defined.

EXAMPLES:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: M.an_element().base_module()
Rank-3 free module M over the Integer Ring
sage: t = M.tensor((2,1))
sage: t.base_module()
Rank-3 free module M over the Integer Ring
sage: t.base_module() is M
True
```

common_basis (*other*)

Find a common basis for the components of `self` and `other`.

In case of multiple common bases, the free module's default basis is privileged. If the current components of `self` and `other` are all relative to different bases, a common basis is searched by performing a component transformation, via the transformations listed in `self._fmodule._basis_changes`, still privileging transformations to the free module's default basis.

INPUT:

- `other` – a tensor (instance of *FreeModuleTensor*)

OUTPUT:

- instance of *FreeModuleBasis* representing the common basis; if no common basis is found, None is returned

EXAMPLES:

Common basis for the components of two module elements:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: u = M([2,1,-5])
sage: f = M.basis('f')
sage: M._basis_changes.clear() # to ensure that bases e and f are unrelated␣
↪at this stage
sage: v = M([0,4,2], basis=f)
sage: u.common_basis(v)
```

The above result is None since u and v have been defined on different bases and no connection between these bases have been set:

```
sage: u._components.keys()
[Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring]
sage: v._components.keys()
[Basis (f_1,f_2,f_3) on the Rank-3 free module M over the Integer Ring]
```

Linking bases e and f changes the result:

```
sage: a = M.automorphism()
sage: a[:] = [[0,0,1], [1,0,0], [0,-1,0]]
sage: M.set_change_of_basis(e, f, a)
sage: u.common_basis(v)
Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring
```

Indeed, v is now known in basis e:

```
sage: v._components.keys() # random output (dictionary keys)
[Basis (f_1,f_2,f_3) on the Rank-3 free module M over the Integer Ring,
 Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring]
```

comp (*basis=None*, *from_basis=None*)

Return the components of self w.r.t to a given module basis.

If the components are not known already, they are computed by the tensor change-of-basis formula from components in another basis.

INPUT:

- basis – (default: None) basis in which the components are required; if none is provided, the components are assumed to refer to the module's default basis
- from_basis – (default: None) basis from which the required components are computed, via the tensor change-of-basis formula, if they are not known already in the basis basis ; if none, a basis from which both the components and a change-of-basis to basis are known is selected.

OUTPUT:

- components in the basis basis , as an instance of the class *Components*

EXAMPLES:

Components of a tensor of type-$(1,1)$:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: t = M.tensor((1,1), name='t')
sage: t[1,2] = -3 ; t[3,3] = 2
sage: t.components()
2-indices components w.r.t. Basis (e_1,e_2,e_3)
 on the Rank-3 free module M over the Integer Ring
sage: t.components() is t.components(e)  # since e is M's default basis
True
sage: t.components()[:]
[ 0 -3  0]
[ 0  0  0]
[ 0  0  2]
```

A shortcut is `t.comp()` :

```
sage: t.comp() is t.components()
True
```

A direct access to the components w.r.t. the module's default basis is provided by the square brackets applied to the tensor itself:

```
sage: t[1,2] is t.comp(e)[1,2]
True
sage: t[:]
[ 0 -3  0]
[ 0  0  0]
[ 0  0  2]
```

Components computed via a change-of-basis formula:

```
sage: a = M.automorphism()
sage: a[:] = [[0,0,1], [1,0,0], [0,-1,0]]
sage: f = e.new_basis(a, 'f')
sage: t.comp(f)
2-indices components w.r.t. Basis (f_1,f_2,f_3)
 on the Rank-3 free module M over the Integer Ring
sage: t.comp(f)[:]
[ 0  0  0]
[ 0  2  0]
[-3  0  0]
```

components (*basis=None*, *from_basis=None*)

Return the components of `self` w.r.t to a given module basis.

If the components are not known already, they are computed by the tensor change-of-basis formula from components in another basis.

INPUT:

- `basis` – (default: `None`) basis in which the components are required; if none is provided, the components are assumed to refer to the module's default basis
- `from_basis` – (default: `None`) basis from which the required components are computed, via the tensor change-of-basis formula, if they are not known already in the basis `basis` ; if none, a basis from which both the components and a change-of-basis to `basis` are known is selected.

OUTPUT:

- components in the basis `basis` , as an instance of the class *Components*

EXAMPLES:

Components of a tensor of type-$(1,1)$:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: t = M.tensor((1,1), name='t')
sage: t[1,2] = -3 ; t[3,3] = 2
sage: t.components()
2-indices components w.r.t. Basis (e_1,e_2,e_3)
 on the Rank-3 free module M over the Integer Ring
sage: t.components() is t.components(e)  # since e is M's default basis
True
sage: t.components()[:]
[ 0 -3  0]
[ 0  0  0]
[ 0  0  2]
```

A shortcut is `t.comp()` :

```
sage: t.comp() is t.components()
True
```

A direct access to the components w.r.t. the module's default basis is provided by the square brackets applied to the tensor itself:

```
sage: t[1,2] is t.comp(e)[1,2]
True
sage: t[:]
[ 0 -3  0]
[ 0  0  0]
[ 0  0  2]
```

Components computed via a change-of-basis formula:

```
sage: a = M.automorphism()
sage: a[:] = [[0,0,1], [1,0,0], [0,-1,0]]
sage: f = e.new_basis(a, 'f')
sage: t.comp(f)
2-indices components w.r.t. Basis (f_1,f_2,f_3)
 on the Rank-3 free module M over the Integer Ring
sage: t.comp(f)[:]
[ 0  0  0]
[ 0  2  0]
[-3  0  0]
```

contract (**args*)

Contraction on one or more indices with another tensor.

INPUT:

- `pos1` – positions of the indices in `self` involved in the contraction; `pos1` must be a sequence of integers, with 0 standing for the first index position, 1 for the second one, etc; if `pos1` is not provided, a single contraction on the last index position of `self` is assumed
- `other` – the tensor to contract with
- `pos2` – positions of the indices in `other` involved in the contraction, with the same conventions as for `pos1` ; if `pos2` is not provided, a single contraction on the first index position of `other` is assumed

OUTPUT:

- tensor resulting from the contraction at the positions `pos1` and `pos2` of `self` with `other`

EXAMPLES:

Contraction of a tensor of type $(0,1)$ with a tensor of type $(1,0)$:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: a = M.linear_form()  # tensor of type (0,1) is a linear form
sage: a[:] = [-3,2,1]
sage: b = M([2,5,-2])  # tensor of type (1,0) is a module element
sage: s = a.contract(b) ; s
2
sage: s in M.base_ring()
True
sage: s == a[0]*b[0] + a[1]*b[1] + a[2]*b[2]  # check of the computation
True
```

The positions of the contraction indices can be set explicitely:

```
sage: s == a.contract(0, b, 0)
True
sage: s == a.contract(0, b)
True
sage: s == a.contract(b, 0)
True
```

Instead of the explicit call to the method *contract()* , the index notation can be used to specify the contraction, via Einstein convention (summation on repeated indices); it suffices to pass the indices as a string inside square brackets:

```
sage: s1 = a['_i']*b['^i'] ; s1
2
sage: s1 == s
True
```

In the present case, performing the contraction is identical to applying the linear form to the module element:

```
sage: a.contract(b) == a(b)
True
```

or to applying the module element, considered as a tensor of type (1,0), to the linear form:

```
sage: a.contract(b) == b(a)
True
```

We have also:

```
sage: a.contract(b) == b.contract(a)
True
```

Contraction of a tensor of type $(1,1)$ with a tensor of type $(1,0)$:

```
sage: a = M.tensor((1,1))
sage: a[:] = [[-1,2,3],[4,-5,6],[7,8,9]]
sage: s = a.contract(b) ; s
```

```
Element of the Rank-3 free module M over the Integer Ring
sage: s.display()
2 e_0 - 29 e_1 + 36 e_2
```

Since the index positions have not been specified, the contraction takes place on the last position of a (i.e. no. 1) and the first position of `b` (i.e. no. 0):

```
sage: a.contract(b) == a.contract(1, b, 0)
True
sage: a.contract(b) == b.contract(0, a, 1)
True
sage: a.contract(b) == b.contract(a, 1)
True
```

Using the index notation with Einstein convention:

```
sage: a['^i_j']*b['^j'] == a.contract(b)
True
```

The index `i` can be replaced by a dot:

```
sage: a['^._j']*b['^j'] == a.contract(b)
True
```

and the symbol `^` may be omitted, the distinction between contravariant and covariant indices being the position with respect to the symbol `_` :

```
sage: a['._j']*b['j'] == a.contract(b)
True
```

Contraction is possible only between a contravariant index and a covariant one:

```
sage: a.contract(0, b)
Traceback (most recent call last):
...
TypeError: contraction on two contravariant indices not permitted
```

Contraction of a tensor of type $(2,1)$ with a tensor of type $(0,2)$:

```
sage: a = a*b ; a
Type-(2,1) tensor on the Rank-3 free module M over the Integer Ring
sage: b = M.tensor((0,2))
sage: b[:] = [[-2,3,1], [0,-2,3], [4,-7,6]]
sage: s = a.contract(1, b, 1) ; s
Type-(1,2) tensor on the Rank-3 free module M over the Integer Ring
sage: s[:]
[[[-9, 16, 39], [18, -32, -78], [27, -48, -117]],
 [[36, -64, -156], [-45, 80, 195], [54, -96, -234]],
 [[63, -112, -273], [72, -128, -312], [81, -144, -351]]]
```

Check of the computation:

```
sage: all(s[i,j,k] == a[i,0,j]*b[k,0]+a[i,1,j]*b[k,1]+a[i,2,j]*b[k,2]
....:     for i in range(3) for j in range(3) for k in range(3))
True
```

Using index notation:

```
sage: a['il_j']*b['_kl'] == a.contract(1, b, 1)
True
```

LaTeX notation are allowed:

```
sage: a['^{il}_j']*b['_{kl}'] == a.contract(1, b, 1)
True
```

Indices not involved in the contraction may be replaced by dots:

```
sage: a['.l_.']*b['_.l'] == a.contract(1, b, 1)
True
```

The two tensors do not have to be defined on the same basis for the contraction to take place, reflecting the fact that the contraction is basis-independent:

```
sage: A = M.automorphism()
sage: A[:] =  [[0,0,1], [1,0,0], [0,-1,0]]
sage: h = e.new_basis(A, 'h')
sage: b.comp(h)[:]  # forces the computation of b's components w.r.t. basis h
[-2 -3  0]
[ 7  6 -4]
[ 3 -1 -2]
sage: b.del_other_comp(h)  # deletes components w.r.t. basis e
sage: b._components.keys()  # indeed:
[Basis (h_0,h_1,h_2) on the Rank-3 free module M over the Integer Ring]
sage: a._components.keys()  # while a is known only in basis e:
[Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring]
sage: s1 = a.contract(1, b, 1) ; s1  # yet the computation is possible
Type-(1,2) tensor on the Rank-3 free module M over the Integer Ring
sage: s1 == s  # ... and yields the same result as previously:
True
```

The contraction can be performed on more than a single index; for instance a 2-indices contraction of a type-$(2,1)$ tensor with a type-$(1,2)$ one is:

```
sage: a  # a is a tensor of type-(2,1)
Type-(2,1) tensor on the Rank-3 free module M over the Integer Ring
sage: b = M([1,-1,2])*b ; b # a tensor of type (1,2)
Type-(1,2) tensor on the Rank-3 free module M over the Integer Ring
sage: s = a.contract(1,2,b,1,0) ; s # the double contraction
Type-(1,1) tensor on the Rank-3 free module M over the Integer Ring
sage: s[:]
[ -36   30   15]
[-252  210  105]
[-204  170   85]
sage: s == a['^.k_l']*b['^l_k.']  # the same thing in index notation
True
```

copy ( )

Return an exact copy of `self` .

The name and the derived quantities are not copied.

EXAMPLES:

Copy of a tensor of type $(1,1)$:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: t = M.tensor((1,1), name='t')
sage: t[1,2] = -3 ; t[3,3] = 2
sage: t1 = t.copy()
sage: t1[:]
[ 0 -3  0]
[ 0  0  0]
[ 0  0  2]
sage: t1 == t
True
```

If the original tensor is modified, the copy is not:

```
sage: t[2,2] = 4
sage: t1[:]
[ 0 -3  0]
[ 0  0  0]
[ 0  0  2]
sage: t1 == t
False
```

del_other_comp (*basis=None*)

Delete all the components but those corresponding to `basis` .

INPUT:

- `basis` – (default: `None`) basis in which the components are kept; if none the module's default basis is assumed

EXAMPLE:

Deleting components of a module element:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: u = M([2,1,-5])
sage: f = M.basis('f')
sage: u.add_comp(f)[:] = [0,4,2]
sage: u._components.keys() # random output (dictionary keys)
[Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring,
 Basis (f_1,f_2,f_3) on the Rank-3 free module M over the Integer Ring]
sage: u.del_other_comp(f)
sage: u._components.keys()
[Basis (f_1,f_2,f_3) on the Rank-3 free module M over the Integer Ring]
```

Let us restore the components w.r.t. e and delete those w.r.t. f:

```
sage: u.add_comp(e)[:] = [2,1,-5]
sage: u._components.keys()  # random output (dictionary keys)
[Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring,
 Basis (f_1,f_2,f_3) on the Rank-3 free module M over the Integer Ring]
sage: u.del_other_comp()  # default argument: basis = e
sage: u._components.keys()
[Basis (e_1,e_2,e_3) on the Rank-3 free module M over the Integer Ring]
```

disp (*basis=None*, *format_spec=None*)

Display `self` in terms of its expansion w.r.t. a given module basis.

The expansion is actually performed onto tensor products of elements of the given basis and of elements of its dual basis (see examples below). The output is either text-formatted (console mode) or LaTeX-formatted (notebook mode).

INPUT:

- `basis` – (default: `None`) basis of the free module with respect to which the tensor is expanded; if none is provided, the module's default basis is assumed
- `format_spec` – (default: `None`) format specification passed to `self._fmodule._output_formatter` to format the output

EXAMPLES:

Display of a module element (type-$(1,0)$ tensor):

```
sage: M = FiniteRankFreeModule(QQ, 2, name='M', start_index=1)
sage: e = M.basis('e') ; e
Basis (e_1,e_2) on the 2-dimensional vector space M over the
 Rational Field
sage: v = M([1/3,-2], name='v')
sage: v.display(e)
v = 1/3 e_1 - 2 e_2
sage: v.display()  # a shortcut since e is M's default basis
v = 1/3 e_1 - 2 e_2
sage: latex(v.display())  # display in the notebook
v = \frac{1}{3} e_1 -2 e_2
```

A shortcut is `disp()` :

```
sage: v.disp()
v = 1/3 e_1 - 2 e_2
```

Display of a linear form (type-$(0,1)$ tensor):

```
sage: de = e.dual_basis() ; de
Dual basis (e^1,e^2) on the 2-dimensional vector space M over the
 Rational Field
sage: w = - 3/4 * de[1] + de[2] ; w
Linear form on the 2-dimensional vector space M over the Rational
 Field
sage: w.set_name('w', latex_name='\omega')
sage: w.display()
w = -3/4 e^1 + e^2
sage: latex(w.display())  # display in the notebook
\omega = -\frac{3}{4} e^1 +e^2
```

Display of a type-$(1,1)$ tensor:

```
sage: t = v*w ; t  # the type-(1,1) is formed as the tensor product of v by w
Type-(1,1) tensor v*w on the 2-dimensional vector space M over the
 Rational Field
sage: t.display()
v*w = -1/4 e_1*e^1 + 1/3 e_1*e^2 + 3/2 e_2*e^1 - 2 e_2*e^2
sage: latex(t.display())  # display in the notebook
v\otimes \omega = -\frac{1}{4} e_1\otimes e^1 +
 \frac{1}{3} e_1\otimes e^2 + \frac{3}{2} e_2\otimes e^1
 -2 e_2\otimes e^2
```

Display in a basis which is not the default one:

```
sage: a = M.automorphism(matrix=[[1,2],[3,4]], basis=e)
sage: f = e.new_basis(a, 'f')
sage: v.display(f) # the components w.r.t basis f are first computed via the␣
↪change-of-basis formula defined by a
v = -8/3 f_1 + 3/2 f_2
sage: w.display(f)
w = 9/4 f^1 + 5/2 f^2
sage: t.display(f)
v*w = -6 f_1*f^1 - 20/3 f_1*f^2 + 27/8 f_2*f^1 + 15/4 f_2*f^2
```

The output format can be set via the argument `output_formatter` passed at the module construction:

```
sage: N = FiniteRankFreeModule(QQ, 2, name='N', start_index=1,
....:                          output_formatter=Rational.numerical_approx)
sage: e = N.basis('e')
sage: v = N([1/3,-2], name='v')
sage: v.display()  # default format (53 bits of precision)
v = 0.333333333333333 e_1 - 2.00000000000000 e_2
sage: latex(v.display())
v = 0.333333333333333 e_1 -2.00000000000000 e_2
```

The output format is then controled by the argument `format_spec` of the method *display()* :

```
sage: v.display(format_spec=10)  # 10 bits of precision
v = 0.33 e_1 - 2.0 e_2
```

display (*basis=None*, *format_spec=None*)

Display `self` in terms of its expansion w.r.t. a given module basis.

The expansion is actually performed onto tensor products of elements of the given basis and of elements of its dual basis (see examples below). The output is either text-formatted (console mode) or LaTeX-formatted (notebook mode).

INPUT:

- `basis` – (default: `None`) basis of the free module with respect to which the tensor is expanded; if none is provided, the module's default basis is assumed
- `format_spec` – (default: `None`) format specification passed to `self._fmodule._output_formatter` to format the output

EXAMPLES:

Display of a module element (type-$(1,0)$ tensor):

```
sage: M = FiniteRankFreeModule(QQ, 2, name='M', start_index=1)
sage: e = M.basis('e') ; e
Basis (e_1,e_2) on the 2-dimensional vector space M over the
 Rational Field
sage: v = M([1/3,-2], name='v')
sage: v.display(e)
v = 1/3 e_1 - 2 e_2
sage: v.display()  # a shortcut since e is M's default basis
v = 1/3 e_1 - 2 e_2
sage: latex(v.display())  # display in the notebook
v = \frac{1}{3} e_1 -2 e_2
```

A shortcut is `disp()` :

```
sage: v.disp()
v = 1/3 e_1 - 2 e_2
```

Display of a linear form (type-$(0,1)$ tensor):

```
sage: de = e.dual_basis() ; de
Dual basis (e^1,e^2) on the 2-dimensional vector space M over the
 Rational Field
sage: w = - 3/4 * de[1] + de[2] ; w
Linear form on the 2-dimensional vector space M over the Rational
 Field
sage: w.set_name('w', latex_name='\omega')
sage: w.display()
w = -3/4 e^1 + e^2
sage: latex(w.display())  # display in the notebook
\omega = -\frac{3}{4} e^1 +e^2
```

Display of a type-$(1,1)$ tensor:

```
sage: t = v*w ; t  # the type-(1,1) is formed as the tensor product of v by w
Type-(1,1) tensor v*w on the 2-dimensional vector space M over the
 Rational Field
sage: t.display()
v*w = -1/4 e_1*e^1 + 1/3 e_1*e^2 + 3/2 e_2*e^1 - 2 e_2*e^2
sage: latex(t.display())  # display in the notebook
v\otimes \omega = -\frac{1}{4} e_1\otimes e^1 +
 \frac{1}{3} e_1\otimes e^2 + \frac{3}{2} e_2\otimes e^1
 -2 e_2\otimes e^2
```

Display in a basis which is not the default one:

```
sage: a = M.automorphism(matrix=[[1,2],[3,4]], basis=e)
sage: f = e.new_basis(a, 'f')
sage: v.display(f) # the components w.r.t basis f are first computed via the␣
 ↪change-of-basis formula defined by a
v = -8/3 f_1 + 3/2 f_2
sage: w.display(f)
w = 9/4 f^1 + 5/2 f^2
sage: t.display(f)
v*w = -6 f_1*f^1 - 20/3 f_1*f^2 + 27/8 f_2*f^1 + 15/4 f_2*f^2
```

The output format can be set via the argument `output_formatter` passed at the module construction:

```
sage: N = FiniteRankFreeModule(QQ, 2, name='N', start_index=1,
....:                          output_formatter=Rational.numerical_approx)
sage: e = N.basis('e')
sage: v = N([1/3,-2], name='v')
sage: v.display()  # default format (53 bits of precision)
v = 0.333333333333333 e_1 - 2.00000000000000 e_2
sage: latex(v.display())
v = 0.333333333333333 e_1 -2.00000000000000 e_2
```

The output format is then controled by the argument `format_spec` of the method *`display()`* :

```
sage: v.display(format_spec=10)  # 10 bits of precision
v = 0.33 e_1 - 2.0 e_2
```

pick_a_basis ( )

Return a basis in which the tensor components are defined.

The free module's default basis is privileged.

OUTPUT:

- instance of *FreeModuleBasis* representing the basis

EXAMPLES:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: t = M.tensor((2,0), name='t')
sage: e = M.basis('e')
sage: t[0,1] = 4  # component set in the default basis (e)
sage: t.pick_a_basis()
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: f = M.basis('f')
sage: t.add_comp(f)[2,1] = -4  # the components in basis e are not erased
sage: t.pick_a_basis()
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: t.set_comp(f)[2,1] = -4  # the components in basis e not erased
sage: t.pick_a_basis()
Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring
```

set_comp (*basis=None*)

Return the components of `self` w.r.t. a given module basis for assignment.

The components with respect to other bases are deleted, in order to avoid any inconsistency. To keep them, use the method *add_comp()* instead.

INPUT:

- `basis` – (default: `None`) basis in which the components are defined; if none is provided, the components are assumed to refer to the module's default basis

OUTPUT:

- components in the given basis, as an instance of the class *Components*; if such components did not exist previously, they are created.

EXAMPLES:

Setting components of a type-$(1,1)$ tensor:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: t = M.tensor((1,1), name='t')
sage: t.set_comp()[0,1] = -3
sage: t.display()
t = -3 e_0*e^1
sage: t.set_comp()[1,2] = 2
sage: t.display()
t = -3 e_0*e^1 + 2 e_1*e^2
sage: t.set_comp(e)
2-indices components w.r.t. Basis (e_0,e_1,e_2) on the
 Rank-3 free module M over the Integer Ring
```

Setting components in a new basis:

```
sage: f =  M.basis('f')
sage: t.set_comp(f)[0,1] = 4
```

```
sage: t._components.keys() # the components w.r.t. basis e have been deleted
[Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring]
sage: t.display(f)
t = 4 f_0*f^1
```

The components w.r.t. basis e can be deduced from those w.r.t. basis f, once a relation between the two bases has been set:

```
sage: a = M.automorphism()
sage: a[:] = [[0,0,1], [1,0,0], [0,-1,0]]
sage: M.set_change_of_basis(e, f, a)
sage: t.display(e)
t = -4 e_1*e^2
sage: t._components.keys()  # random output (dictionary keys)
[Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring,
 Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring]
```

set_name (*name=None*, *latex_name=None*)

Set (or change) the text name and LaTeX name of `self` .

INPUT:

- `name` – (default: `None`) string; name given to the tensor
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the tensor; if None while `name` is provided, the LaTeX symbol is set to `name`

EXAMPLES:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: t = M.tensor((2,1)) ; t
Type-(2,1) tensor on the Rank-3 free module M over the Integer Ring
sage: t.set_name('t') ; t
Type-(2,1) tensor t on the Rank-3 free module M over the Integer Ring
sage: latex(t)
t
sage: t.set_name(latex_name=r'\tau') ; t
Type-(2,1) tensor t on the Rank-3 free module M over the Integer Ring
sage: latex(t)
\tau
```

symmetries ( )

Print the list of symmetries and antisymmetries of `self` .

EXAMPLES:

Various symmetries / antisymmetries for a rank-4 tensor:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: t = M.tensor((4,0), name='T') # no symmetry declared
sage: t.symmetries()
no symmetry;  no antisymmetry
sage: t = M.tensor((4,0), name='T', sym=(0,1))
sage: t.symmetries()
symmetry: (0, 1);  no antisymmetry
sage: t = M.tensor((4,0), name='T', sym=[(0,1), (2,3)])
sage: t.symmetries()
symmetries: [(0, 1), (2, 3)];  no antisymmetry
sage: t = M.tensor((4,0), name='T', sym=(0,1), antisym=(2,3))
```

```
sage: t.symmetries()
symmetry: (0, 1);  antisymmetry: (2, 3)
```

symmetrize (**pos*, ***kwargs*)

Symmetrization over some arguments.

INPUT:

- `pos` – list of argument positions involved in the symmetrization (with the convention `position=0` for the first argument); if none, the symmetrization is performed over all the arguments
- `basis` – (default: `None`) module basis with respect to which the component computation is to be performed; if none, the module's default basis is used if the tensor field has already components in it; otherwise another basis w.r.t. which the tensor has components will be picked

OUTPUT:

- the symmetrized tensor (instance of *FreeModuleTensor*)

EXAMPLES:

Symmetrization of a tensor of type $(2,0)$:

```
sage: M = FiniteRankFreeModule(QQ, 3, name='M')
sage: e = M.basis('e')
sage: t = M.tensor((2,0))
sage: t[:] = [[2,1,-3],[0,-4,5],[-1,4,2]]
sage: s = t.symmetrize() ; s
Type-(2,0) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: t[:], s[:]
(
[ 2  1 -3]  [  2 1/2  -2]
[ 0 -4  5]  [1/2  -4 9/2]
[-1  4  2], [ -2 9/2   2]
)
sage: s.symmetries()
symmetry: (0, 1);  no antisymmetry
sage: all(s[i,j] == 1/2*(t[i,j]+t[j,i])    # check:
....:     for i in range(3) for j in range(3))
True
```

Instead of invoking the method *symmetrize()* , one may use the index notation with parentheses to denote the symmetrization; it suffices to pass the indices as a string inside square brackets:

```
sage: t['(ij)']
Type-(2,0) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: t['(ij)'].symmetries()
symmetry: (0, 1);  no antisymmetry
sage: t['(ij)'] == t.symmetrize()
True
```

The indices names are not significant; they can even be replaced by dots:

```
sage: t['(..)'] == t.symmetrize()
True
```

The LaTeX notation can be used as well:

```
sage: t['^{(ij)}'] == t.symmetrize()
True
```

Symmetrization of a tensor of type $(0,3)$ on the first two arguments:

```
sage: t = M.tensor((0,3))
sage: t[:] = [[[1,2,3], [-4,5,6], [7,8,-9]],
....:         [[10,-11,12], [13,14,-15], [16,17,18]],
....:         [[19,-20,-21], [-22,23,24], [25,26,-27]]]
sage: s = t.symmetrize(0,1) ; s  # (0,1) = the first two arguments
Type-(0,3) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s.symmetries()
symmetry: (0, 1);  no antisymmetry
sage: s[:]
[[[1, 2, 3], [3, -3, 9], [13, -6, -15]],
 [[3, -3, 9], [13, 14, -15], [-3, 20, 21]],
 [[13, -6, -15], [-3, 20, 21], [25, 26, -27]]]
sage: all(s[i,j,k] == 1/2*(t[i,j,k]+t[j,i,k])    # Check:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s.symmetrize(0,1) == s  # another test
True
```

Again the index notation can be used:

```
sage: t['_(ij)k'] == t.symmetrize(0,1)
True
sage: t['_(..).'] == t.symmetrize(0,1)  # no index name
True
sage: t['_{(ij)k}'] == t.symmetrize(0,1)  # LaTeX notation
True
sage: t['_{(..).}'] == t.symmetrize(0,1)  # this also allowed
True
```

Symmetrization of a tensor of type $(0,3)$ on the first and last arguments:

```
sage: s = t.symmetrize(0,2) ; s  # (0,2) = first and last arguments
Type-(0,3) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s.symmetries()
symmetry: (0, 2);  no antisymmetry
sage: s[:]
[[[1, 6, 11], [-4, 9, -8], [7, 12, 8]],
 [[6, -11, -4], [9, 14, 4], [12, 17, 22]],
 [[11, -4, -21], [-8, 4, 24], [8, 22, -27]]]
sage: all(s[i,j,k] == 1/2*(t[i,j,k]+t[k,j,i])
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s.symmetrize(0,2) == s  # another test
True
```

Symmetrization of a tensor of type $(0,3)$ on the last two arguments:

```
sage: s = t.symmetrize(1,2) ; s  # (1,2) = the last two arguments
Type-(0,3) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s.symmetries()
```

```
symmetry: (1, 2);  no antisymmetry
sage: s[:]
[[[1, -1, 5], [-1, 5, 7], [5, 7, -9]],
 [[10, 1, 14], [1, 14, 1], [14, 1, 18]],
 [[19, -21, 2], [-21, 23, 25], [2, 25, -27]]]
sage: all(s[i,j,k] == 1/2*(t[i,j,k]+t[i,k,j])    # Check:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s.symmetrize(1,2) == s  # another test
True
```

Use of the index notation:

```
sage: t['_i(jk)'] == t.symmetrize(1,2)
True
sage: t['_.(..)'] == t.symmetrize(1,2)
True
sage: t['_{i(jk)}'] == t.symmetrize(1,2)  # LaTeX notation
True
```

Full symmetrization of a tensor of type $(0,3)$:

```
sage: s = t.symmetrize() ; s
Type-(0,3) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s.symmetries()
symmetry: (0, 1, 2);  no antisymmetry
sage: s[:]
[[[1, 8/3, 29/3], [8/3, 7/3, 0], [29/3, 0, -5/3]],
 [[8/3, 7/3, 0], [7/3, 14, 25/3], [0, 25/3, 68/3]],
 [[29/3, 0, -5/3], [0, 25/3, 68/3], [-5/3, 68/3, -27]]]
sage: all(s[i,j,k] == 1/
↪6*(t[i,j,k]+t[i,k,j]+t[j,k,i]+t[j,i,k]+t[k,i,j]+t[k,j,i])  # Check:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s.symmetrize() == s  # another test
True
```

Index notation for the full symmetrization:

```
sage: t['_(ijk)'] == t.symmetrize()
True
sage: t['_{(ijk)}'] == t.symmetrize()  # LaTeX notation
True
```

Symmetrization can be performed only on arguments on the same type:

```
sage: t = M.tensor((1,2))
sage: t[:] = [[[1,2,3], [-4,5,6], [7,8,-9]],
....:         [[10,-11,12], [13,14,-15], [16,17,18]],
....:         [[19,-20,-21], [-22,23,24], [25,26,-27]]]
sage: s = t.symmetrize(0,1)
Traceback (most recent call last):
...
TypeError: 0 is a contravariant position, while 1 is a covariant position;
symmetrization is meaningfull only on tensor arguments of the same type
sage: s = t.symmetrize(1,2) # OK: both 1 and 2 are covariant positions
```

The order of positions does not matter:

```
sage: t.symmetrize(2,1) == t.symmetrize(1,2)
True
```

Use of the index notation:

```
sage: t['^i_(jk)'] == t.symmetrize(1,2)
True
sage: t['^._(..)'] ==  t.symmetrize(1,2)
True
```

The character ^ can be skipped, the character _ being sufficient to separate contravariant indices from covariant ones:

```
sage: t['i_(jk)'] == t.symmetrize(1,2)
True
```

The LaTeX notation can be employed:

```
sage: t['^{i}_{(jk)}'] == t.symmetrize(1,2)
True
```

tensor_rank ( )

Return the tensor rank of self .

OUTPUT:

- integer k+l , where k is the contravariant rank and l is the covariant rank

EXAMPLES:

```
sage: M = FiniteRankFreeModule(ZZ, 3)
sage: M.an_element().tensor_rank()
1
sage: t = M.tensor((2,1))
sage: t.tensor_rank()
3
```

tensor_type ( )

Return the tensor type of self .

OUTPUT:

- pair (k,l) , where k is the contravariant rank and l is the covariant rank

EXAMPLES:

```
sage: M = FiniteRankFreeModule(ZZ, 3)
sage: M.an_element().tensor_type()
(1, 0)
sage: t = M.tensor((2,1))
sage: t.tensor_type()
(2, 1)
```

trace (*pos1=0*, *pos2=1*)

Trace (contraction) on two slots of the tensor.

INPUT:

- `pos1` – (default: 0) position of the first index for the contraction, with the convention `pos1=0` for the first slot
- `pos2` – (default: 1) position of the second index for the contraction, with the same convention as for `pos1`; the variance type of `pos2` must be opposite to that of `pos1`

OUTPUT:

- tensor or scalar resulting from the `(pos1,pos2)` contraction

EXAMPLES:

Trace of a type-$(1,1)$ tensor:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e') ; e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: a = M.tensor((1,1), name='a') ; a
Type-(1,1) tensor a on the Rank-3 free module M over the Integer Ring
sage: a[:] = [[1,2,3], [4,5,6], [7,8,9]]
sage: a.trace()
15
sage: a.trace(0,1)  # equivalent to above (contraction of slot 0 with slot 1)
15
sage: a.trace(1,0)  # the order of the slots does not matter
15
```

Instead of the explicit call to the method `trace()`, one may use the index notation with Einstein convention (summation over repeated indices); it suffices to pass the indices as a string inside square brackets:

```
sage: a['^i_i']
15
```

The letter 'i' to denote the repeated index can be replaced by any other letter:

```
sage: a['^s_s']
15
```

Moreover, the symbol `^` can be omitted:

```
sage: a['i_i']
15
```

The contraction on two slots having the same tensor type cannot occur:

```
sage: b =  M.tensor((2,0), name='b') ; b
Type-(2,0) tensor b on the Rank-3 free module M over the Integer Ring
sage: b[:] = [[1,2,3], [4,5,6], [7,8,9]]
sage: b.trace(0,1)
Traceback (most recent call last):
...
IndexError: contraction on two contravariant indices is not allowed
```

The contraction either preserves or destroys the symmetries:

```
sage: b = M.alternating_form(2, 'b') ; b
Alternating form b of degree 2 on the Rank-3 free module M
 over the Integer Ring
sage: b[0,1], b[0,2], b[1,2] = 3, 2, 1
sage: t = a*b ; t
```

```
Type-(1,3) tensor a*b on the Rank-3 free module M
 over the Integer Ring
```

By construction, `t` is a tensor field antisymmetric w.r.t. its last two slots:

```
sage: t.symmetries()
no symmetry;  antisymmetry: (2, 3)
sage: s = t.trace(0,1) ; s   # contraction on the first two slots
Alternating form of degree 2 on the
 Rank-3 free module M over the Integer Ring
sage: s.symmetries()    # the antisymmetry is preserved
no symmetry;  antisymmetry: (0, 1)
sage: s[:]
[  0  45  30]
[-45   0  15]
[-30 -15   0]
sage: s == 15*b  # check
True
sage: s = t.trace(0,2) ; s   # contraction on the first and third slots
Type-(0,2) tensor on the Rank-3 free module M over the Integer Ring
sage: s.symmetries()  # the antisymmetry has been destroyed by the above␣
↪contraction:
no symmetry;  no antisymmetry
sage: s[:]  # indeed:
[-26  -4   6]
[-31  -2   9]
[-36   0  12]
sage: s[:] == matrix( [[sum(t[k,i,k,j] for k in M.irange())
....:           for j in M.irange()] for i in M.irange()] )  # check
True
```

Use of index notation instead of *trace()* :

```
sage: t['^k_kij'] == t.trace(0,1)
True
sage: t['^k_{kij}'] == t.trace(0,1) # LaTeX notation
True
sage: t['^k_ikj'] == t.trace(0,2)
True
sage: t['^k_ijk'] == t.trace(0,3)
True
```

Index symbols not involved in the contraction may be replaced by dots:

```
sage: t['^k_k..'] == t.trace(0,1)
True
sage: t['^k_.k.'] == t.trace(0,2)
True
sage: t['^k_..k'] == t.trace(0,3)
True
```

view (*basis=None*, *format_spec=None*)

Deprecated method.

Use method *display()* instead.

EXAMPLE:

```
sage: M = FiniteRankFreeModule(ZZ, 2, 'M')
sage: e = M.basis('e')
sage: v = M([2,-3], basis=e, name='v')
sage: v.view(e)
doctest:...: DeprecationWarning: Use function display() instead.
See http://trac.sagemath.org/15916 for details.
v = 2 e_0 - 3 e_1
sage: v.display(e)
v = 2 e_0 - 3 e_1
```

3.3 Index notation for tensors

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version

class sage.tensor.modules.tensor_with_indices. **TensorWithIndices** (*tensor*, *indices*)
Bases: sage.structure.sage_object.SageObject

Index notation for tensors.

This is a technical class to allow one to write some tensor operations (contractions and symmetrizations) in index notation.

INPUT:

- tensor – a tensor (or a tensor field)
- indices – string containing the indices, as single letters; the contravariant indices must be stated first and separated from the covariant indices by the character _

EXAMPLES:

Index representation of tensors on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(QQ, 3, name='M')
sage: e = M.basis('e')
sage: a = M.tensor((2,0), name='a')
sage: a[:] = [[1,2,3], [4,5,6], [7,8,9]]
sage: b = M.tensor((0,2), name='b')
sage: b[:] = [[-1,2,-3], [-4,5,6], [7,-8,9]]
sage: t = a*b ; t.set_name('t') ; t
Type-(2,2) tensor t on the 3-dimensional vector space M over the
 Rational Field
sage: from sage.tensor.modules.tensor_with_indices import TensorWithIndices
sage: T = TensorWithIndices(t, '^ij_kl') ; T
t^ij_kl
```

The *TensorWithIndices* object is returned by the square bracket operator acting on the tensor and fed with the string specifying the indices:

```
sage: a['^ij']
a^ij
sage: type(a['^ij'])
<class 'sage.tensor.modules.tensor_with_indices.TensorWithIndices'>
sage: b['_ef']
b_ef
```

```
sage: t['^ij_kl']
t^ij_kl
```

The symbol '^' may be omitted, since the distinction between covariant and contravariant indices is performed by the index position relative to the symbol '_':

```
sage: t['ij_kl']
t^ij_kl
```

Also, LaTeX notation may be used:

```
sage: t['^{ij}_{kl}']
t^ij_kl
```

If some operation is asked in the index notation, the resulting tensor is returned, not a *TensorWithIndices* object; for instance, for a symmetrization:

```
sage: s = t['^(ij)_kl'] ; s  # the symmetrization on i,j is indicated by␣
↪parentheses
Type-(2,2) tensor on the 3-dimensional vector space M over the
 Rational Field
sage: s.symmetries()
symmetry: (0, 1);  no antisymmetry
sage: s == t.symmetrize(0,1)
True
```

The letters denoting the indices can be chosen freely; since they carry no information, they can even be replaced by dots:

```
sage: t['^(..)_..'] == t.symmetrize(0,1)
True
```

Similarly, for an antisymmetrization:

```
sage: s = t['^ij_[kl]'] ; s # the symmetrization on k,l is indicated by square␣
↪brackets
Type-(2,2) tensor on the 3-dimensional vector space M over the Rational
 Field
sage: s.symmetries()
no symmetry;  antisymmetry: (2, 3)
sage: s == t.antisymmetrize(2,3)
True
```

Another example of an operation indicated by indices is a contraction:

```
sage: s = t['^ki_kj'] ; s  # contraction on the repeated index k
Type-(1,1) tensor on the 3-dimensional vector space M over the Rational
 Field
sage: s == t.trace(0,2)
True
```

Indices not involved in the contraction may be replaced by dots:

```
sage: s == t['^k._k.']
True
```

The contraction of two tensors is indicated by repeated indices and the $*$ operator:

```
sage: s = a['^ik'] * b['_kj'] ; s
Type-(1,1) tensor on the 3-dimensional vector space M over the Rational
 Field
sage: s == a.contract(1, b, 0)
True
sage: s = t['^.k_..'] * b['_.k'] ; s
Type-(1,3) tensor on the 3-dimensional vector space M over the Rational
 Field
sage: s == t.contract(1, b, 1)
True
sage: t['^{ik}_{jl}']*b['_{mk}'] == s # LaTeX notation
True
```

Contraction on two indices:

```
sage: s = a['^kl'] * b['_kl'] ; s
105
sage: s == a.contract(0,1, b, 0,1)
True
```

Some minimal arithmetics:

```
sage: 2*a['^ij']
X^ij
sage: (2*a['^ij'])._tensor == 2*a
True
sage: 2*t['ij_kl']
X^ij_kl
sage: +a['^ij']
+a^ij
sage: +t['ij_kl']
+t^ij_kl
sage: -a['^ij']
-a^ij
sage: -t['ij_kl']
-t^ij_kl
```

update ( )

Return the tensor contains in `self` if it differs from that used for creating `self` , otherwise return `self` .

EXAMPLES:

```
sage: from sage.tensor.modules.tensor_with_indices import TensorWithIndices
sage: M = FiniteRankFreeModule(QQ, 3, name='M')
sage: e = M.basis('e')
sage: a = M.tensor((1,1),  name='a')
sage: a[:] = [[1,-2,3], [-4,5,-6], [7,-8,9]]
sage: a_ind = TensorWithIndices(a, 'i_j') ; a_ind
a^i_j
sage: a_ind.update()
a^i_j
sage: a_ind.update() is a_ind
True
sage: a_ind = TensorWithIndices(a, 'k_k') ; a_ind
scalar
sage: a_ind.update()
15
```

CHAPTER

FOUR

ALTERNATING FORMS

4.1 Exterior powers of dual free modules

Given a free module M of finite rank over a commutative ring R and a positive integer p, the *p-th exterior power* of the dual of M is the set $\Lambda^p(M^*)$ of all alternating forms of degree p on M, i.e. of all multilinear maps

$$\underbrace{M \times \cdots \times M}_{p \text{ times}} \longrightarrow R$$

that vanish whenever any of two of their arguments are equal. Note that $\Lambda^1(M^*) = M^*$ (the dual of M).

$\Lambda^p(M^*)$ is a free module of rank $\binom{n}{p}$ over R, where n is the rank of M. Accordingly, exterior powers of free modules are implemented by a class, `ExtPowerFreeModule`, which inherits from the class `FiniteRankFreeModule`.

AUTHORS:

- Eric Gourgoulhon (2015): initial version

REFERENCES:

- K. Conrad: *Exterior powers*, http://www.math.uconn.edu/~kconrad/blurbs/
- Chap. 19 of S. Lang: *Algebra*, 3rd ed., Springer (New York) (2002)

class sage.tensor.modules.ext_pow_free_module. **ExtPowerFreeModule**(*fmodule*, *degree*, *name=None*, *latex_name=None*)

Bases: `sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule`

Class for the exterior powers of the dual of a free module of finite rank over a commutative ring.

Given a free module M of finite rank over a commutative ring R and a positive integer p, the *p-th exterior power* of the dual of M is the set $\Lambda^p(M^*)$ of all alternating forms of degree p on M, i.e. of all multilinear maps

$$\underbrace{M \times \cdots \times M}_{p \text{ times}} \longrightarrow R$$

that vanish whenever any of two of their arguments are equal. Note that $\Lambda^1(M^*) = M^*$ (the dual of M).

$\Lambda^p(M^*)$ is a free module of rank $\binom{n}{p}$ over R, where n is the rank of M. Accordingly, the class `ExtPowerFreeModule` inherits from the class `FiniteRankFreeModule`.

This is a Sage *parent* class, whose *element* class is `FreeModuleAltForm`.

INPUT:

- `fmodule` – free module M of finite rank, as an instance of *FiniteRankFreeModule*
- `degree` – positive integer; the degree p of the alternating forms
- `name` – (default: `None`) string; name given to $\Lambda^p(M^*)$
- `latex_name` – (default: `None`) string; LaTeX symbol to denote $\Lambda^p(M^*)$

EXAMPLES:

2nd exterior power of the dual of a free $\mathbf{Z}$-module of rank 3:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: from sage.tensor.modules.ext_pow_free_module import ExtPowerFreeModule
sage: A = ExtPowerFreeModule(M, 2) ; A
2nd exterior power of the dual of the Rank-3 free module M over the
 Integer Ring
```

Instead of importing ExtPowerFreeModule in the global name space, it is recommended to use the module's method *dual_exterior_power()* :

```
sage: A = M.dual_exterior_power(2) ; A
2nd exterior power of the dual of the Rank-3 free module M over the
 Integer Ring
sage: latex(A)
\Lambda^{2}\left(M^*\right)
```

`A` is a module (actually a free module) over $\mathbf{Z}$:

```
sage: A.category()
Category of finite dimensional modules over Integer Ring
sage: A in Modules(ZZ)
True
sage: A.rank()
3
sage: A.base_ring()
Integer Ring
sage: A.base_module()
Rank-3 free module M over the Integer Ring
```

`A` is a *parent* object, whose elements are alternating forms, represented by instances of the class *FreeModuleAltForm* :

```
sage: a = A.an_element() ; a
Alternating form of degree 2 on the Rank-3 free module M over the
 Integer Ring
sage: a.display() # expansion with respect to M's default basis (e)
e^0/\e^1
sage: from sage.tensor.modules.free_module_alt_form import FreeModuleAltForm
sage: isinstance(a, FreeModuleAltForm)
True
sage: a in A
True
sage: A.is_parent_of(a)
True
```

Elements can be constructed from `A` . In particular, 0 yields the zero element of `A` :

```
sage: A(0)
Alternating form zero of degree 2 on the Rank-3 free module M over the
 Integer Ring
sage: A(0) is A.zero()
True
```

while non-zero elements are constructed by providing their components in a given basis:

```
sage: e
Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer Ring
sage: comp = [[0,3,-1],[-3,0,4],[1,-4,0]]
sage: a = A(comp, basis=e, name='a') ; a
Alternating form a of degree 2 on the Rank-3 free module M over the
 Integer Ring
sage: a.display(e)
a = 3 e^0/\e^1 - e^0/\e^2 + 4 e^1/\e^2
```

An alternative is to construct the alternating form from an empty list of components and to set the nonzero components afterwards:

```
sage: a = A([], name='a')
sage: a.set_comp(e)[0,1] = 3
sage: a.set_comp(e)[0,2] = -1
sage: a.set_comp(e)[1,2] = 4
sage: a.display(e)
a = 3 e^0/\e^1 - e^0/\e^2 + 4 e^1/\e^2
```

The exterior powers are unique:

```
sage: A is M.dual_exterior_power(2)
True
```

The exterior power $\Lambda^1(M^*)$ is nothing but M^*:

```
sage: M.dual_exterior_power(1) is M.dual()
True
sage: M.dual()
Dual of the Rank-3 free module M over the Integer Ring
sage: latex(M.dual())
M^*
```

Since any tensor of type (0,1) is a linear form, there is a coercion map from the set $T^{(0,1)}(M)$ of such tensors to M^*:

```
sage: T01 = M.tensor_module(0,1) ; T01
Free module of type-(0,1) tensors on the Rank-3 free module M over the
 Integer Ring
sage: M.dual().has_coerce_map_from(T01)
True
```

There is also a coercion map in the reverse direction:

```
sage: T01.has_coerce_map_from(M.dual())
True
```

For a degree $p \geq 2$, the coercion holds only in the direction $\Lambda^p(M^*) \to T^{(0,p)}(M)$:

```
sage: T02 = M.tensor_module(0,2) ; T02
Free module of type-(0,2) tensors on the Rank-3 free module M over the
 Integer Ring
sage: T02.has_coerce_map_from(A)
True
sage: A.has_coerce_map_from(T02)
False
```

The coercion map $T^{(0,1)}(M) \to M^*$ in action:

```
sage: b = T01([-2,1,4], basis=e, name='b') ; b
Type-(0,1) tensor b on the Rank-3 free module M over the Integer Ring
sage: b.display(e)
b = -2 e^0 + e^1 + 4 e^2
sage: lb = M.dual()(b) ; lb
Linear form b on the Rank-3 free module M over the Integer Ring
sage: lb.display(e)
b = -2 e^0 + e^1 + 4 e^2
```

The coercion map $M^* \to T^{(0,1)}(M)$ in action:

```
sage: tlb = T01(lb) ; tlb
Type-(0,1) tensor b on the Rank-3 free module M over the Integer Ring
sage: tlb == b
True
```

The coercion map $\Lambda^2(M^*) \to T^{(0,2)}(M)$ in action:

```
sage: ta = T02(a) ; ta
Type-(0,2) tensor a on the Rank-3 free module M over the Integer Ring
sage: ta.display(e)
a = 3 e^0*e^1 - e^0*e^2 - 3 e^1*e^0 + 4 e^1*e^2 + e^2*e^0 - 4 e^2*e^1
sage: a.display(e)
a = 3 e^0/\e^1 - e^0/\e^2 + 4 e^1/\e^2
sage: ta.symmetries() # the antisymmetry is of course preserved
no symmetry;  antisymmetry: (0, 1)
```

Element
alias of `FreeModuleAltForm`

base_module ( )
Return the free module on which `self` is constructed.

OUTPUT:

- instance of *FiniteRankFreeModule* representing the free module on which the exterior power is defined.

EXAMPLE:

```
sage: M = FiniteRankFreeModule(ZZ, 5, name='M')
sage: A = M.dual_exterior_power(2)
sage: A.base_module()
Rank-5 free module M over the Integer Ring
sage: A.base_module() is M
True
```

degree ( )
Return the degree of `self` .

OUTPUT:

- integer p such that `self` is the exterior power $\Lambda^p(M^*)$

EXAMPLES:

```
sage: M = FiniteRankFreeModule(ZZ, 5, name='M')
sage: A = M.dual_exterior_power(2)
sage: A.degree()
2
sage: M.dual_exterior_power(4).degree()
4
```

4.2 Alternating forms on free modules

Given a free module M of finite rank over a commutative ring R and a positive integer p, an *alternating form of degree p* on M is a map

$$a : \underbrace{M \times \cdots \times M}_{p \text{ times}} \longrightarrow R$$

that (i) is multilinear and (ii) vanishes whenever any of two of its arguments are equal. An alternating form of degree p is a tensor on M of type $(0, p)$.

Alternating forms are implemented via the class *FreeModuleAltForm*, which is a subclass of the generic tensor class *FreeModuleTensor*.

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version

REFERENCES:

- Chap. 23 of R. Godement: *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)
- Chap. 15 of S. Lang: *Algebra*, 3rd ed., Springer (New York) (2002)

class sage.tensor.modules.free_module_alt_form. **FreeModuleAltForm** (*fmodule*, *degree*, *name=None*, *latex_name=None*)

Bases: *sage.tensor.modules.free_module_tensor.FreeModuleTensor*

Alternating form on a free module of finite rank over a commutative ring.

This is a Sage *element* class, the corresponding *parent* class being *ExtPowerFreeModule*.

INPUT:

- `fmodule` – free module M of finite rank over a commutative ring R, as an instance of *FiniteRankFreeModule*
- `degree` – positive integer; the degree p of the alternating form (i.e. its tensor rank)
- `name` – (default: `None`) string; name given to the alternating form
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the alternating form; if none is provided, `name` is used

EXAMPLES:

Alternating form of degree 2 on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: a = M.alternating_form(2, name='a') ; a
Alternating form a of degree 2 on the
 Rank-3 free module M over the Integer Ring
sage: type(a)
<class 'sage.tensor.modules.free_module_alt_form.ExtPowerFreeModule_with_category.
 ↪element_class'>
sage: a.parent()
2nd exterior power of the dual of the Rank-3 free module M over the Integer Ring
sage: a[1,2], a[2,3] = 4, -3
sage: a.display(e)
a = 4 e^1/\e^2 - 3 e^2/\e^3
```

The alternating form acting on the basis elements:

```
sage: a(e[1],e[2])
4
sage: a(e[1],e[3])
0
sage: a(e[2],e[3])
-3
sage: a(e[2],e[1])
-4
```

An alternating form of degree 1 is a linear form:

```
sage: b = M.linear_form('b') ; b
Linear form b on the Rank-3 free module M over the Integer Ring
sage: b[:] = [2,-1,3]  # components w.r.t. the module's default basis (e)
```

A linear form is a tensor of type (0,1):

```
sage: b.tensor_type()
(0, 1)
```

It is an element of the dual module:

```
sage: b.parent()
Dual of the Rank-3 free module M over the Integer Ring
sage: b.parent() is M.dual()
True
```

The members of a dual basis are linear forms:

```
sage: e.dual_basis()[1]
Linear form e^1 on the Rank-3 free module M over the Integer Ring
sage: e.dual_basis()[2]
Linear form e^2 on the Rank-3 free module M over the Integer Ring
sage: e.dual_basis()[3]
Linear form e^3 on the Rank-3 free module M over the Integer Ring
```

Any linear form is expanded onto them:

```
sage: b.display(e)
b = 2 e^1 - e^2 + 3 e^3
```

In the above example, an equivalent writing would have been `b.display()`, since the basis `e` is the module's default basis. A linear form maps module elements to ring elements:

```
sage: v = M([1,1,1])
sage: b(v)
4
sage: b(v) in M.base_ring()
True
```

Test of linearity:

```
sage: u = M([-5,-2,7])
sage: b(3*u - 4*v) == 3*b(u) - 4*b(v)
True
```

The standard tensor operations apply to alternating forms, like the extraction of components with respect to a given basis:

```
sage: a[e,1,2]
4
sage: a[1,2]  # since e is the module's default basis
4
sage: all( a[i,j] == - a[j,i] for i in {1,2,3} for j in {1,2,3} )
True
```

the tensor product:

```
sage: c = b*b ; c
Symmetric bilinear form  b*b on the Rank-3 free module M over the
 Integer Ring
sage: c.parent()
Free module of type-(0,2) tensors on the Rank-3 free module M over the
 Integer Ring
sage: c.display(e)
b*b = 4 e^1*e^1 - 2 e^1*e^2 + 6 e^1*e^3 - 2 e^2*e^1 + e^2*e^2
 - 3 e^2*e^3 + 6 e^3*e^1 - 3 e^3*e^2 + 9 e^3*e^3
```

the contractions:

```
sage: s = a.contract(v) ; s
Linear form on the Rank-3 free module M over the Integer Ring
sage: s.parent()
Dual of the Rank-3 free module M over the Integer Ring
sage: s.display(e)
4 e^1 - 7 e^2 + 3 e^3
```

or tensor arithmetics:

```
sage: s = 3*a + c ; s
Type-(0,2) tensor on the Rank-3 free module M over the Integer Ring
sage: s.parent()
Free module of type-(0,2) tensors on the Rank-3 free module M over the
 Integer Ring
sage: s.display(e)
```

```
4 e^1*e^1 + 10 e^1*e^2 + 6 e^1*e^3 - 14 e^2*e^1 + e^2*e^2
 - 12 e^2*e^3 + 6 e^3*e^1 + 6 e^3*e^2 + 9 e^3*e^3
```

Note that tensor arithmetics preserves the alternating character if both operands are alternating:

```
sage: s = a - 2*a ; s
Alternating form of degree 2 on the Rank-3 free module M over the
 Integer Ring
sage: s.parent() # note the difference with s = 3*a + c above
2nd exterior power of the dual of the Rank-3 free module M over the
 Integer Ring
sage: s == -a
True
```

An operation specific to alternating forms is of course the exterior product:

```
sage: s = a.wedge(b) ; s
Alternating form a/\b of degree 3 on the Rank-3 free module M over the
 Integer Ring
sage: s.parent()
3rd exterior power of the dual of the Rank-3 free module M over the
 Integer Ring
sage: s.display(e)
a/\b = 6 e^1/\e^2/\e^3
sage: s[1,2,3] == a[1,2]*b[3] + a[2,3]*b[1] + a[3,1]*b[2]
True
```

The exterior product is nilpotent on linear forms:

```
sage: s = b.wedge(b) ; s
Alternating form b/\b of degree 2 on the Rank-3 free module M over the
 Integer Ring
sage: s.display(e)
b/\b = 0
```

degree ( )

Return the degree of `self` .

EXAMPLE:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: a = M.alternating_form(2, name='a')
sage: a.degree()
2
```

disp (*basis=None*, *format_spec=None*)

Display the alternating form `self` in terms of its expansion w.r.t. a given module basis.

The expansion is actually performed onto exterior products of elements of the cobasis (dual basis) associated with `basis` (see examples below). The output is either text-formatted (console mode) or LaTeX-formatted (notebook mode).

INPUT:

- `basis` – (default: `None`) basis of the free module with respect to which the alternating form is expanded; if none is provided, the module's default basis is assumed
- `format_spec` – (default: `None`) format specification passed to `self._fmodule._output_formatter` to format the output

EXAMPLES:

Display of an alternating form of degree 1 (linear form) on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: e.dual_basis()
Dual basis (e^0,e^1,e^2) on the Rank-3 free module M over the Integer Ring
sage: a = M.linear_form('a', latex_name=r'\alpha')
sage: a[:] = [1,-3,4]
sage: a.display(e)
a = e^0 - 3 e^1 + 4 e^2
sage: a.display()  # a shortcut since e is M's default basis
a = e^0 - 3 e^1 + 4 e^2
sage: latex(a.display())  # display in the notebook
\alpha = e^0 -3 e^1 + 4 e^2
```

A shortcut is `disp()` :

```
sage: a.disp()
a = e^0 - 3 e^1 + 4 e^2
```

Display of an alternating form of degree 2 on a rank-3 free module:

```
sage: b = M.alternating_form(2, 'b', latex_name=r'\beta')
sage: b[0,1], b[0,2], b[1,2] = 3, 2, -1
sage: b.display()
b = 3 e^0/\e^1 + 2 e^0/\e^2 - e^1/\e^2
sage: latex(b.display())  # display in the notebook
\beta = 3 e^0\wedge e^1 + 2 e^0\wedge e^2 -e^1\wedge e^2
```

Display of an alternating form of degree 3 on a rank-3 free module:

```
sage: c = M.alternating_form(3, 'c')
sage: c[0,1,2] = 4
sage: c.display()
c = 4 e^0/\e^1/\e^2
sage: latex(c.display())
c = 4 e^0\wedge e^1\wedge e^2
```

Display of a vanishing alternating form:

```
sage: c[0,1,2] = 0  # the only independent component set to zero
sage: c.is_zero()
True
sage: c.display()
c = 0
sage: latex(c.display())
c = 0
sage: c[0,1,2] = 4  # value restored for what follows
```

Display in a basis which is not the default one:

```
sage: aut = M.automorphism(matrix=[[0,1,0], [0,0,-1], [1,0,0]],
....:                      basis=e)
sage: f = e.new_basis(aut, 'f')
sage: a.display(f)
a = 4 f^0 + f^1 + 3 f^2
sage: a.disp(f)     # shortcut notation
```

```
a = 4 f^0 + f^1 + 3 f^2
sage: b.display(f)
b = -2 f^0/\f^1 - f^0/\f^2 - 3 f^1/\f^2
sage: c.display(f)
c = -4 f^0/\f^1/\f^2
```

The output format can be set via the argument `output_formatter` passed at the module construction:

```
sage: N = FiniteRankFreeModule(QQ, 3, name='N', start_index=1,
....:                          output_formatter=Rational.numerical_approx)
sage: e = N.basis('e')
sage: b = N.alternating_form(2, 'b')
sage: b[1,2], b[1,3], b[2,3] = 1/3, 5/2, 4
sage: b.display()  # default format (53 bits of precision)
b = 0.333333333333333 e^1/\e^2 + 2.50000000000000 e^1/\e^3
 + 4.00000000000000 e^2/\e^3
```

The output format is then controled by the argument `format_spec` of the method *display()* :

```
sage: b.display(format_spec=10)  # 10 bits of precision
b = 0.33 e^1/\e^2 + 2.5 e^1/\e^3 + 4.0 e^2/\e^3
```

display (*basis=None*, *format_spec=None*)

Display the alternating form `self` in terms of its expansion w.r.t. a given module basis.

The expansion is actually performed onto exterior products of elements of the cobasis (dual basis) associated with `basis` (see examples below). The output is either text-formatted (console mode) or LaTeX-formatted (notebook mode).

INPUT:

- `basis` – (default: `None`) basis of the free module with respect to which the alternating form is expanded; if none is provided, the module's default basis is assumed
- `format_spec` – (default: `None`) format specification passed to `self._fmodule._output_formatter` to format the output

EXAMPLES:

Display of an alternating form of degree 1 (linear form) on a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: e.dual_basis()
Dual basis (e^0,e^1,e^2) on the Rank-3 free module M over the Integer Ring
sage: a = M.linear_form('a', latex_name=r'\alpha')
sage: a[:] = [1,-3,4]
sage: a.display(e)
a = e^0 - 3 e^1 + 4 e^2
sage: a.display()  # a shortcut since e is M's default basis
a = e^0 - 3 e^1 + 4 e^2
sage: latex(a.display())  # display in the notebook
\alpha = e^0 -3 e^1 + 4 e^2
```

A shortcut is `disp()` :

```
sage: a.disp()
a = e^0 - 3 e^1 + 4 e^2
```

Display of an alternating form of degree 2 on a rank-3 free module:

```
sage: b = M.alternating_form(2, 'b', latex_name=r'\beta')
sage: b[0,1], b[0,2], b[1,2] = 3, 2, -1
sage: b.display()
b = 3 e^0/\e^1 + 2 e^0/\e^2 - e^1/\e^2
sage: latex(b.display())  # display in the notebook
\beta = 3 e^0\wedge e^1 + 2 e^0\wedge e^2 -e^1\wedge e^2
```

Display of an alternating form of degree 3 on a rank-3 free module:

```
sage: c = M.alternating_form(3, 'c')
sage: c[0,1,2] = 4
sage: c.display()
c = 4 e^0/\e^1/\e^2
sage: latex(c.display())
c = 4 e^0\wedge e^1\wedge e^2
```

Display of a vanishing alternating form:

```
sage: c[0,1,2] = 0  # the only independent component set to zero
sage: c.is_zero()
True
sage: c.display()
c = 0
sage: latex(c.display())
c = 0
sage: c[0,1,2] = 4  # value restored for what follows
```

Display in a basis which is not the default one:

```
sage: aut = M.automorphism(matrix=[[0,1,0], [0,0,-1], [1,0,0]],
....:                      basis=e)
sage: f = e.new_basis(aut, 'f')
sage: a.display(f)
a = 4 f^0 + f^1 + 3 f^2
sage: a.disp(f)     # shortcut notation
a = 4 f^0 + f^1 + 3 f^2
sage: b.display(f)
b = -2 f^0/\f^1 - f^0/\f^2 - 3 f^1/\f^2
sage: c.display(f)
c = -4 f^0/\f^1/\f^2
```

The output format can be set via the argument `output_formatter` passed at the module construction:

```
sage: N = FiniteRankFreeModule(QQ, 3, name='N', start_index=1,
....:                  output_formatter=Rational.numerical_approx)
sage: e = N.basis('e')
sage: b = N.alternating_form(2, 'b')
sage: b[1,2], b[1,3], b[2,3] = 1/3, 5/2, 4
sage: b.display()  # default format (53 bits of precision)
b = 0.333333333333333 e^1/\e^2 + 2.50000000000000 e^1/\e^3
 + 4.00000000000000 e^2/\e^3
```

The output format is then controled by the argument `format_spec` of the method *`display()`* :

```
sage: b.display(format_spec=10)  # 10 bits of precision
b = 0.33 e^1/\e^2 + 2.5 e^1/\e^3 + 4.0 e^2/\e^3
```

wedge (*other*)

Exterior product of `self` with the alternating form `other`.

INPUT:

- `other` – an alternating form

OUTPUT:

- instance of *FreeModuleAltForm* representing the exterior product `self/\other`

EXAMPLES:

Exterior product of two linear forms:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: a = M.linear_form('A')
sage: a[:] = [1,-3,4]
sage: b = M.linear_form('B')
sage: b[:] = [2,-1,2]
sage: c = a.wedge(b) ; c
Alternating form A/\B of degree 2 on the Rank-3 free module M
 over the Integer Ring
sage: c.display()
A/\B = 5 e^0/\e^1 - 6 e^0/\e^2 - 2 e^1/\e^2
sage: latex(c)
A\wedge B
sage: latex(c.display())
A\wedge B = 5 e^0\wedge e^1 -6 e^0\wedge e^2 -2 e^1\wedge e^2
```

Test of the computation:

```
sage: a.wedge(b) == a*b - b*a
True
```

Exterior product of a linear form and an alternating form of degree 2:

```
sage: d = M.linear_form('D')
sage: d[:] = [-1,2,4]
sage: s = d.wedge(c) ; s
Alternating form D/\A/\B of degree 3 on the Rank-3 free module M
 over the Integer Ring
sage: s.display()
D/\A/\B = 34 e^0/\e^1/\e^2
```

Test of the computation:

```
sage: s[0,1,2] == d[0]*c[1,2] + d[1]*c[2,0] + d[2]*c[0,1]
True
```

Let us check that the exterior product is associative:

```
sage: d.wedge(a.wedge(b)) == (d.wedge(a)).wedge(b)
True
```

and that it is graded anticommutative:

```
sage: a.wedge(b) == - b.wedge(a)
True
```

```
sage: d.wedge(c) == c.wedge(d)
True
```

CHAPTER

FIVE

MORPHISMS

5.1 Sets of morphisms between free modules

The class *FreeModuleHomset* implements sets of homomorphisms between two free modules of finite rank over the same commutative ring.

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version

REFERENCES:

- Chaps. 13, 14 of R. Godement : *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)
- Chap. 3 of S. Lang : *Algebra*, 3rd ed., Springer (New York) (2002)

class sage.tensor.modules.free_module_homset. **FreeModuleHomset** (*fmodule1*, *fmodule2*, *name=None*, *latex_name=None*)

Bases: sage.categories.homset.Homset

Set of homomorphisms between free modules of finite rank over a commutative ring.

Given two free modules M and N of respective ranks m and n over a commutative ring R, the class *FreeModuleHomset* implements the set $\mathrm{Hom}(M, N)$ of homomorphisms $M \to N$. The set $\mathrm{Hom}(M, N)$ is actually a free module of rank mn over R, but this aspect is not taken into account here.

This is a Sage *parent* class, whose *element* class is *FiniteRankFreeModuleMorphism*.

INPUT:

- fmodule1 – free module M (domain of the homomorphisms), as an instance of *FiniteRankFreeModule*
- fmodule2 – free module N (codomain of the homomorphisms), as an instance of *FiniteRankFreeModule*
- name – (default: None) string; name given to the hom-set; if none is provided, Hom(M,N) will be used
- latex_name – (default: None) string; LaTeX symbol to denote the hom-set; if none is provided, $\mathrm{Hom}(M, N)$ will be used

EXAMPLES:

Set of homomorphisms between two free modules over **Z**:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: N = FiniteRankFreeModule(ZZ, 2, name='N')
sage: H = Hom(M,N) ; H
Set of Morphisms from Rank-3 free module M over the Integer Ring
 to Rank-2 free module N over the Integer Ring
 in Category of finite dimensional modules over Integer Ring
sage: type(H)
<class 'sage.tensor.modules.free_module_homset.FreeModuleHomset_with_category_
 ↪with_equality_by_id'>
sage: H.category()
Category of homsets of modules over Integer Ring
```

Hom-sets are cached:

```
sage: H is Hom(M,N)
True
```

The LaTeX formatting is:

```
sage: latex(H)
\mathrm{Hom}\left(M,N\right)
```

As usual, the construction of an element is performed by the `__call__` method; the argument can be the matrix representing the morphism in the default bases of the two modules:

```
sage: e = M.basis('e')
sage: f = N.basis('f')
sage: phi = H([[-1,2,0], [5,1,2]]) ; phi
Generic morphism:
  From: Rank-3 free module M over the Integer Ring
  To:   Rank-2 free module N over the Integer Ring
sage: phi.parent() is H
True
```

An example of construction from a matrix w.r.t. bases that are not the default ones:

```
sage: ep = M.basis('ep', latex_symbol=r"e'")
sage: fp = N.basis('fp', latex_symbol=r"f'")
sage: phi2 = H([[3,2,1], [1,2,3]], bases=(ep,fp)) ; phi2
Generic morphism:
  From: Rank-3 free module M over the Integer Ring
  To:   Rank-2 free module N over the Integer Ring
```

The zero element:

```
sage: z = H.zero() ; z
Generic morphism:
  From: Rank-3 free module M over the Integer Ring
  To:   Rank-2 free module N over the Integer Ring
sage: z.matrix(e,f)
[0 0 0]
[0 0 0]
```

The test suite for H is passed:

```
sage: TestSuite(H).run()
```

The set of homomorphisms $M \to M$, i.e. endomorphisms, is obtained by the function `End` :

```
sage: End(M)
Set of Morphisms from Rank-3 free module M over the Integer Ring
 to Rank-3 free module M over the Integer Ring
 in Category of finite dimensional modules over Integer Ring
```

`End(M)` is actually identical to `Hom(M,M)` :

```
sage: End(M) is Hom(M,M)
True
```

The unit of the endomorphism ring is the identity map:

```
sage: End(M).one()
Identity endomorphism of Rank-3 free module M over the Integer Ring
```

whose matrix in any basis is of course the identity matrix:

```
sage: End(M).one().matrix(e)
[1 0 0]
[0 1 0]
[0 0 1]
```

There is a canonical identification between endomorphisms of M and tensors of type (1,1) on M. Accordingly, coercion maps have been implemented between $\mathrm{End}(M)$ and $T^{(1,1)}(M)$ (the module of all type-(1,1) tensors on M, see *TensorFreeModule*):

```
sage: T11 = M.tensor_module(1,1) ; T11
Free module of type-(1,1) tensors on the Rank-3 free module M over
 the Integer Ring
sage: End(M).has_coerce_map_from(T11)
True
sage: T11.has_coerce_map_from(End(M))
True
```

See *TensorFreeModule* for examples of the above coercions.

There is a coercion $\mathrm{GL}(M) \to \mathrm{End}(M)$, since every automorphism is an endomorphism:

```
sage: GL = M.general_linear_group() ; GL
General linear group of the Rank-3 free module M over the Integer Ring
sage: End(M).has_coerce_map_from(GL)
True
```

Of course, there is no coercion in the reverse direction, since only bijective endomorphisms are automorphisms:

```
sage: GL.has_coerce_map_from(End(M))
False
```

The coercion $\mathrm{GL}(M) \to \mathrm{End}(M)$ in action:

```
sage: a = GL.an_element() ; a
Automorphism of the Rank-3 free module M over the Integer Ring
sage: a.matrix(e)
[ 1  0  0]
[ 0 -1  0]
[ 0  0  1]
sage: ea = End(M)(a) ; ea
Generic endomorphism of Rank-3 free module M over the Integer Ring
```

```
sage: ea.matrix(e)
[ 1  0  0]
[ 0 -1  0]
[ 0  0  1]
```

Element
alias of `FiniteRankFreeModuleMorphism`

one ( )
Return the identity element of `self` considered as a monoid (case of an endomorphism set).

This applies only when the codomain of `self` is equal to its domain, i.e. when `self` is of the type $\mathrm{Hom}(M, M)$.

OUTPUT:

- the identity element of $\mathrm{End}(M) = \mathrm{Hom}(M, M)$, as an instance of *FiniteRankFreeModuleMorphism*

EXAMPLE:

Identity element of the set of endomorphisms of a free module over $\mathbf{Z}$:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: H = End(M)
sage: H.one()
Identity endomorphism of Rank-3 free module M over the Integer Ring
sage: H.one().matrix(e)
[1 0 0]
[0 1 0]
[0 0 1]
sage: H.one().is_identity()
True
```

NB: mathematically, `H.one()` coincides with the identity map of the free module M. However the latter is considered here as an element of $\mathrm{GL}(M)$, the general linear group of M. Accordingly, one has to use the coercion map $\mathrm{GL}(M) \to \mathrm{End}(M)$ to recover `H.one()` from `M.identity_map()`:

```
sage: M.identity_map()
Identity map of the Rank-3 free module M over the Integer Ring
sage: M.identity_map().parent()
General linear group of the Rank-3 free module M over the Integer Ring
sage: H.one().parent()
Set of Morphisms from Rank-3 free module M over the Integer Ring
 to Rank-3 free module M over the Integer Ring
 in Category of finite dimensional modules over Integer Ring
sage: H.one() == H(M.identity_map())
True
```

Conversely, one can recover `M.identity_map()` from `H.one()` by means of a conversion $\mathrm{End}(M) \to \mathrm{GL}(M)$:

```
sage: GL = M.general_linear_group()
sage: M.identity_map() == GL(H.one())
True
```

5.2 Free module morphisms

The class `FiniteRankFreeModuleMorphism` implements homomorphisms between two free modules of finite rank over the same commutative ring.

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version

REFERENCES:

- Chaps. 13, 14 of R. Godement: *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)
- Chap. 3 of S. Lang: *Algebra*, 3rd ed., Springer (New York) (2002)

class sage.tensor.modules.free_module_morphism. **FiniteRankFreeModuleMorphism** (*parent*, *matrix_rep*, *bases=None*, *name=None*, *latex_name=None*, *is_identity=False*)

Bases: sage.categories.morphism.Morphism

Homomorphism between free modules of finite rank over a commutative ring.

An instance of this class is a homomorphism

$$\phi : M \longrightarrow N,$$

where M and N are two free modules of finite rank over the same commutative ring R.

This is a Sage *element* class, the corresponding *parent* class being `FreeModuleHomset` .

INPUT:

- `parent` – hom-set Hom(M,N) to which the homomorphism belongs
- `matrix_rep` – matrix representation of the homomorphism with respect to the bases `bases` ; this entry can actually be any material from which a matrix of size rank(N)*rank(M) of elements of R can be constructed; the *columns* of the matrix give the images of the basis of M (see the convention in the example below)
- `bases` – (default: `None`) pair (basis_M, basis_N) defining the matrix representation, basis_M being a basis of module M and basis_N a basis of module N ; if None the pair formed by the default bases of each module is assumed.
- `name` – (default: `None`) string; name given to the homomorphism
- `latex_name` – (default: `None`) string; LaTeX symbol to denote the homomorphism; if None, `name` will be used.
- `is_identity` – (default: `False`) determines whether the constructed object is the identity endomorphism; if set to `True` , then N must be M and the entry `matrix_rep` is not used.

EXAMPLES:

A homomorphism between two free modules over $\mathbf{Z}$ is contructed as an element of the corresponding hom-set, by means of the function `__call__` :

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: N = FiniteRankFreeModule(ZZ, 2, name='N')
sage: e = M.basis('e') ; f = N.basis('f')
sage: H = Hom(M,N) ; H
Set of Morphisms from Rank-3 free module M over the Integer Ring
 to Rank-2 free module N over the Integer Ring
 in Category of finite dimensional modules over Integer Ring
sage: phi = H([[2,-1,3], [1,0,-4]], name='phi', latex_name=r'\phi') ; phi
Generic morphism:
  From: Rank-3 free module M over the Integer Ring
  To:   Rank-2 free module N over the Integer Ring
```

Since no bases have been specified in the argument list, the provided matrix is relative to the default bases of modules M and N, so that the above is equivalent to:

```
sage: phi = H([[2,-1,3], [1,0,-4]], bases=(e,f), name='phi',
....:         latex_name=r'\phi') ; phi
Generic morphism:
  From: Rank-3 free module M over the Integer Ring
  To:   Rank-2 free module N over the Integer Ring
```

An alternative way to construct a homomorphism is to call the method *hom()* on the domain:

```
sage: phi = M.hom(N, [[2,-1,3], [1,0,-4]], bases=(e,f), name='phi',
....:             latex_name=r'\phi') ; phi
Generic morphism:
  From: Rank-3 free module M over the Integer Ring
  To:   Rank-2 free module N over the Integer Ring
```

The parent of a homomorphism is of course the corresponding hom-set:

```
sage: phi.parent() is H
True
sage: phi.parent() is Hom(M,N)
True
```

Due to Sage's category scheme, the actual class of the homomorphism `phi` is a derived class of *FiniteRankFreeModuleMorphism*:

```
sage: type(phi)
<class 'sage.tensor.modules.free_module_morphism.FreeModuleHomset_with_category_
 ↪with_equality_by_id.element_class'>
sage: isinstance(phi, sage.tensor.modules.free_module_morphism.
 ↪FiniteRankFreeModuleMorphism)
True
```

The domain and codomain of the homomorphism are returned respectively by the methods `domain()` and `codomain()`, which are implemented as Sage's constant functions:

```
sage: phi.domain()
Rank-3 free module M over the Integer Ring
sage: phi.codomain()
Rank-2 free module N over the Integer Ring
sage: type(phi.domain)
<type 'sage.misc.constant_function.ConstantFunction'>
```

The matrix of the homomorphism with respect to a pair of bases is returned by the method *matrix()*:

```
sage: phi.matrix(e,f)
[ 2 -1  3]
[ 1  0 -4]
```

The convention is that the columns of this matrix give the components of the images of the elements of basis `e` w.r.t basis `f` :

```
sage: phi(e[0]).display()
phi(e_0) = 2 f_0 + f_1
sage: phi(e[1]).display()
phi(e_1) = -f_0
sage: phi(e[2]).display()
phi(e_2) = 3 f_0 - 4 f_1
```

Test of the module homomorphism laws:

```
sage: phi(M.zero()) == N.zero()
True
sage: u = M([1,2,3], basis=e, name='u') ; u.display()
u = e_0 + 2 e_1 + 3 e_2
sage: v = M([-2,1,4], basis=e, name='v') ; v.display()
v = -2 e_0 + e_1 + 4 e_2
sage: phi(u).display()
phi(u) = 9 f_0 - 11 f_1
sage: phi(v).display()
phi(v) = 7 f_0 - 18 f_1
sage: phi(3*u + v).display()
34 f_0 - 51 f_1
sage: phi(3*u + v) == 3*phi(u) + phi(v)
True
```

The identity endomorphism:

```
sage: Id = End(M).one() ; Id
Identity endomorphism of Rank-3 free module M over the Integer Ring
sage: Id.parent()
Set of Morphisms from Rank-3 free module M over the Integer Ring
 to Rank-3 free module M over the Integer Ring
 in Category of finite dimensional modules over Integer Ring
sage: Id.parent() is End(M)
True
```

The matrix of Id with respect to the basis e is of course the identity matrix:

```
sage: Id.matrix(e)
[1 0 0]
[0 1 0]
[0 0 1]
```

The identity acting on a module element:

```
sage: Id(v) is v
True
```

is_identity ( )

Check whether `self` is the identity morphism.

EXAMPLES:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M')
sage: e = M.basis('e')
sage: phi = M.endomorphism([[1,0], [0,1]])
sage: phi.is_identity()
True
sage: (phi+phi).is_identity()
False
sage: End(M).zero().is_identity()
False
sage: a = M.automorphism() ; a[0,1], a[1,0] = 1, -1
sage: ep = e.new_basis(a, 'ep', latex_symbol="e'")
sage: phi = M.endomorphism([[1,0], [0,1]], basis=ep)
sage: phi.is_identity()
True
```

Example illustrating that the identity can be constructed from a matrix that is not the identity one, provided that it is relative to different bases:

```
sage: phi = M.hom(M, [[0,1], [-1,0]], bases=(ep,e))
sage: phi.is_identity()
True
```

Of course, if we ask for the matrix in a single basis, it is the identity matrix:

```
sage: phi.matrix(e)
[1 0]
[0 1]
sage: phi.matrix(ep)
[1 0]
[0 1]
```

is_injective ( )

Determine whether `self` is injective.

OUTPUT:

- `True` if `self` is an injective homomorphism and `False` otherwise

EXAMPLES:

Homomorphisms between two **Z**-modules:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: N = FiniteRankFreeModule(ZZ, 2, name='N')
sage: e = M.basis('e') ; f = N.basis('f')
sage: phi = M.hom(N, [[-1,2,0], [5,1,2]])
sage: phi.matrix(e,f)
[-1  2  0]
[ 5  1  2]
sage: phi.is_injective()
False
```

Indeed, phi has a non trivial kernel:

```
sage: phi(4*e[0] + 2*e[1] - 11*e[2]).display()
0
```

An injective homomorphism:

```
sage: psi = N.hom(M, [[1,-1], [0,3], [4,-5]])
sage: psi.matrix(f,e)
[ 1 -1]
[ 0  3]
[ 4 -5]
sage: psi.is_injective()
True
```

Of course, the identity endomorphism is injective:

```
sage: End(M).one().is_injective()
True
sage: End(N).one().is_injective()
True
```

is_surjective ( )

Determine whether `self` is surjective.

OUTPUT:

- `True` if `self` is a surjective homomorphism and `False` otherwise

EXAMPLE:

This method has not been implemented yet:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: N = FiniteRankFreeModule(ZZ, 2, name='N')
sage: e = M.basis('e') ; f = N.basis('f')
sage: phi = M.hom(N, [[-1,2,0], [5,1,2]])
sage: phi.is_surjective()
Traceback (most recent call last):
...
NotImplementedError: FiniteRankFreeModuleMorphism.is_surjective()
 has not been implemented yet
```

except for the identity endomorphisme (!):

```
sage: End(M).one().is_surjective()
True
sage: End(N).one().is_surjective()
True
```

matrix (*basis1=None*, *basis2=None*)

Return the matrix of `self` w.r.t to a pair of bases.

If the matrix is not known already, it is computed from the matrix in another pair of bases by means of the change-of-basis formula.

INPUT:

- `basis1` – (default: `None`) basis of the domain of `self`; if none is provided, the domain's default basis is assumed
- `basis2` – (default: `None`) basis of the codomain of `self`; if none is provided, `basis2` is set to `basis1` if `self` is an endomorphism, otherwise, `basis2` is set to the codomain's default basis.

OUTPUT:

- the matrix representing representing the homomorphism `self` w.r.t to bases `basis1` and `basis2` ; more precisely, the columns of this matrix are formed by the components w.r.t. `basis2` of the images of the elements of `basis1` .

EXAMPLES:

Matrix of a homomorphism between two **Z**-modules:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: N = FiniteRankFreeModule(ZZ, 2, name='N')
sage: e = M.basis('e') ; f = N.basis('f')
sage: phi = M.hom(N, [[-1,2,0], [5,1,2]])
sage: phi.matrix()      # default bases
[-1  2  0]
[ 5  1  2]
sage: phi.matrix(e,f)  # bases explicited
[-1  2  0]
[ 5  1  2]
sage: type(phi.matrix())
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
```

Matrix in bases different from those in which the homomorphism has been defined:

```
sage: a = M.automorphism(matrix=[[-1,0,0],[0,1,2],[0,1,3]], basis=e)
sage: ep = e.new_basis(a, 'ep', latex_symbol="e'")
sage: b = N.automorphism(matrix=[[3,5],[4,7]], basis=f)
sage: fp = f.new_basis(b, 'fp', latex_symbol="f'")
sage: phi.matrix(ep, fp)
[ 32  -1 -12]
[-19   1   8]
```

Check of the change-of-basis formula:

```
sage: phi.matrix(ep, fp) == (b^(-1)).matrix(f) * phi.matrix(e,f) * a.matrix(e)
True
```

Single change of basis:

```
sage: phi.matrix(ep, f)
[ 1  2  4]
[-5  3  8]
sage: phi.matrix(ep,f) == phi.matrix(e,f) * a.matrix(e)
True
sage: phi.matrix(e, fp)
[-32   9 -10]
[ 19  -5   6]
sage: phi.matrix(e, fp) == (b^(-1)).matrix(f) * phi.matrix(e,f)
True
```

Matrix of an endomorphism:

```
sage: phi = M.endomorphism([[1,2,3], [4,5,6], [7,8,9]], basis=ep)
sage: phi.matrix(ep)
[1 2 3]
[4 5 6]
[7 8 9]
sage: phi.matrix(ep,ep)  # same as above
[1 2 3]
[4 5 6]
```

```
[7 8 9]
sage: phi.matrix()  # matrix w.r.t to the module's default basis
[  1  -3   1]
[-18  39 -18]
[-25  54 -25]
```

5.3 General linear group of a free module

The set $\mathrm{GL}(M)$ of automorphisms (i.e. invertible endomorphims) of a free module of finite rank M is a group under composition of automorphisms, named the *general linear group* of M. In other words, $\mathrm{GL}(M)$ is the group of units (i.e. invertible elements) of $\mathrm{End}(M)$, the endomorphism ring of M.

The group $\mathrm{GL}(M)$ is implemented via the class *FreeModuleLinearGroup*.

AUTHORS:

- Eric Gourgoulhon (2015): initial version

REFERENCES:

- Chap. 15 of R. Godement: *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)

class sage.tensor.modules.free_module_linear_group. **FreeModuleLinearGroup** (*fmodule*)
Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.Parent

General linear group of a free module of finite rank over a commutative ring.

Given a free module of finite rank M over a commutative ring R, the *general linear group* of M is the group $\mathrm{GL}(M)$ of automorphisms (i.e. invertible endomorphims) of M. It is the group of units (i.e. invertible elements) of $\mathrm{End}(M)$, the endomorphism ring of M.

This is a Sage *parent* class, whose *element* class is *FreeModuleAutomorphism*.

INPUT:

- fmodule – free module M of finite rank over a commutative ring R, as an instance of *FiniteRankFreeModule*

EXAMPLES:

General linear group of a free $\mathbf{Z}$-module of rank 3:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: from sage.tensor.modules.free_module_linear_group import␣
↪FreeModuleLinearGroup
sage: GL = FreeModuleLinearGroup(M) ; GL
General linear group of the Rank-3 free module M over the Integer Ring
```

Instead of importing FreeModuleLinearGroup in the global name space, it is recommended to use the module's method *general_linear_group()*:

```
sage: GL = M.general_linear_group() ; GL
General linear group of the Rank-3 free module M over the Integer Ring
sage: latex(GL)
\mathrm{GL}\left( M \right)
```

As most parents, the general linear group has a unique instance:

```
sage: GL is M.general_linear_group()
True
```

$\mathrm{GL}(M)$ is in the category of groups:

```
sage: GL.category()
Category of groups
sage: GL in Groups()
True
```

`GL` is a *parent* object, whose elements are automorphisms of M, represented by instances of the class *FreeModuleAutomorphism*:

```
sage: GL.Element
<class 'sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism'>
sage: a = GL.an_element() ; a
Automorphism of the Rank-3 free module M over the Integer Ring
sage: a.matrix(e)
[ 1  0  0]
[ 0 -1  0]
[ 0  0  1]
sage: a in GL
True
sage: GL.is_parent_of(a)
True
```

As an endomorphism, `a` maps elements of M to elements of M:

```
sage: v = M.an_element() ; v
Element of the Rank-3 free module M over the Integer Ring
sage: v.display()
e_0 + e_1 + e_2
sage: a(v)
Element of the Rank-3 free module M over the Integer Ring
sage: a(v).display()
e_0 - e_1 + e_2
```

An automorphism can also be viewed as a tensor of type (1,1) on M:

```
sage: a.tensor_type()
(1, 1)
sage: a.display(e)
e_0*e^0 - e_1*e^1 + e_2*e^2
sage: type(a)
<class 'sage.tensor.modules.free_module_automorphism.FreeModuleLinearGroup_with_
↪category.element_class'>
```

As for any group, the identity element is obtained by the method *one()* :

```
sage: id = GL.one() ; id
Identity map of the Rank-3 free module M over the Integer Ring
sage: id*a == a
True
sage: a*id == a
True
sage: a*a^(-1) == id
True
```

```
sage: a^(-1)*a == id
True
```

The identity element is of course the identity map of the module M:

```
sage: id(v) == v
True
sage: id.matrix(e)
[1 0 0]
[0 1 0]
[0 0 1]
```

The module's changes of basis are stored as elements of the general linear group:

```
sage: f = M.basis('f', from_family=(-e[1], 4*e[0]+3*e[2], 7*e[0]+5*e[2]))
sage: f
Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer Ring
sage: M.change_of_basis(e,f)
Automorphism of the Rank-3 free module M over the Integer Ring
sage: M.change_of_basis(e,f) in GL
True
sage: M.change_of_basis(e,f).parent()
General linear group of the Rank-3 free module M over the Integer Ring
sage: M.change_of_basis(e,f).matrix(e)
[ 0  4  7]
[-1  0  0]
[ 0  3  5]
sage: M.change_of_basis(e,f) == M.change_of_basis(f,e).inverse()
True
```

Since every automorphism is an endomorphism, there is a coercion $\mathrm{GL}(M) \to \mathrm{End}(M)$ (the endomorphism ring of module M):

```
sage: End(M).has_coerce_map_from(GL)
True
```

(see *FreeModuleHomset* for details), but not in the reverse direction, since only bijective endomorphisms are automorphisms:

```
sage: GL.has_coerce_map_from(End(M))
False
```

A bijective endomorphism can be converted to an element of $\mathrm{GL}(M)$:

```
sage: h = M.endomorphism([[1,0,0], [0,-1,2], [0,1,-3]]) ; h
Generic endomorphism of Rank-3 free module M over the Integer Ring
sage: h.parent() is End(M)
True
sage: ah = GL(h) ; ah
Automorphism of the Rank-3 free module M over the Integer Ring
sage: ah.parent() is GL
True
```

As maps $M \to M$, `ah` and `h` are identical:

```
sage: v  # recall
Element of the Rank-3 free module M over the Integer Ring
sage: ah(v) == h(v)
```

```
True
sage: ah.matrix(e) == h.matrix(e)
True
```

Of course, non-invertible endomorphisms cannot be converted to elements of $\mathrm{GL}(M)$:

```
sage: GL(M.endomorphism([[0,0,0], [0,-1,2], [0,1,-3]]))
Traceback (most recent call last):
...
TypeError: the Generic endomorphism of Rank-3 free module M over the
 Integer Ring is not invertible
```

Similarly, there is a coercion $\mathrm{GL}(M) \to T^{(1,1)}(M)$ (module of type-(1,1) tensors):

```
sage: M.tensor_module(1,1).has_coerce_map_from(GL)
True
```

(see *TensorFreeModule* for details), but not in the reverse direction, since not every type-(1,1) tensor can be considered as an automorphism:

```
sage: GL.has_coerce_map_from(M.tensor_module(1,1))
False
```

Invertible type-(1,1) tensors can be converted to automorphisms:

```
sage: t = M.tensor((1,1), name='t')
sage: t[e,:] = [[-1,0,0], [0,1,2], [0,1,3]]
sage: at = GL(t) ; at
Automorphism t of the Rank-3 free module M over the Integer Ring
sage: at.matrix(e)
[-1  0  0]
[ 0  1  2]
[ 0  1  3]
sage: at.matrix(e) == t[e,:]
True
```

Non-invertible ones cannot:

```
sage: t0 = M.tensor((1,1), name='t_0')
sage: t0[e,0,0] = 1
sage: t0[e,:]  # the matrix is clearly not invertible
[1 0 0]
[0 0 0]
[0 0 0]
sage: GL(t0)
Traceback (most recent call last):
...
TypeError: the Type-(1,1) tensor t_0 on the Rank-3 free module M over
 the Integer Ring is not invertible
sage: t0[e,1,1], t0[e,2,2] = 2, 3
sage: t0[e,:]  # the matrix is not invertible in Mat_3(ZZ)
[1 0 0]
[0 2 0]
[0 0 3]
sage: GL(t0)
Traceback (most recent call last):
...
```

```
TypeError: the Type-(1,1) tensor t_0 on the Rank-3 free module M over
 the Integer Ring is not invertible
```

Element
alias of FreeModuleAutomorphism

base_module ( )
Return the free module of which self is the general linear group.

OUTPUT:

- instance of *FiniteRankFreeModule* representing the free module of which self is the general linear group

EXAMPLE:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M')
sage: GL = M.general_linear_group()
sage: GL.base_module()
Rank-2 free module M over the Integer Ring
sage: GL.base_module() is M
True
```

one ( )
Return the group identity element of self .

The group identity element is nothing but the module identity map.

OUTPUT:

- instance of *FreeModuleAutomorphism* representing the identity element.

EXAMPLES:

Identity element of the general linear group of a rank-2 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M', start_index=1)
sage: GL = M.general_linear_group()
sage: GL.one()
Identity map of the Rank-2 free module M over the Integer Ring
```

The identity element is cached:

```
sage: GL.one() is GL.one()
True
```

Check that the element returned is indeed the neutral element for the group law:

```
sage: e = M.basis('e')
sage: a = GL([[3,4],[5,7]], basis=e) ; a
Automorphism of the Rank-2 free module M over the Integer Ring
sage: a.matrix(e)
[3 4]
[5 7]
sage: GL.one() * a == a
True
sage: a * GL.one() == a
True
sage: a * a^(-1) == GL.one()
True
```

```
sage: a^(-1) * a == GL.one()
True
```

The unit element of $\mathrm{GL}(M)$ is the identity map of M:

```
sage: GL.one()(e[1])
Element e_1 of the Rank-2 free module M over the Integer Ring
sage: GL.one()(e[2])
Element e_2 of the Rank-2 free module M over the Integer Ring
```

Its matrix is the identity matrix in any basis:

```
sage: GL.one().matrix(e)
[1 0]
[0 1]
sage: f = M.basis('f', from_family=(e[1]+2*e[2], e[1]+3*e[2]))
sage: GL.one().matrix(f)
[1 0]
[0 1]
```

5.4 Free module automorphisms

Given a free module M of finite rank over a commutative ring R, an *automorphism* of M is a map

$$\phi : M \longrightarrow M$$

that is linear (i.e. is a module homomorphism) and bijective.

Automorphisms of a free module of finite rank are implemented via the class *FreeModuleAutomorphism*.

AUTHORS:

- Eric Gourgoulhon (2015): initial version

REFERENCES:

- Chaps. 15, 24 of R. Godement: *Algebra*, Hermann (Paris) / Houghton Mifflin (Boston) (1968)

class sage.tensor.modules.free_module_automorphism. **FreeModuleAutomorphism** (*fmodule*, *name=None*, *latex_name=None*, *is_identity=False*)

Bases: *sage.tensor.modules.free_module_tensor.FreeModuleTensor*, sage.structure.element.MultiplicativeGroupElement

Automorphism of a free module of finite rank over a commutative ring.

This is a Sage *element* class, the corresponding *parent* class being *FreeModuleLinearGroup*.

This class inherits from the classes *FreeModuleTensor* and MultiplicativeGroupElement.

INPUT:

- fmodule – free module M of finite rank over a commutative ring R, as an instance of *FiniteRankFreeModule*
- name – (default: None) name given to the automorphism

- `latex_name` – (default: `None`) LaTeX symbol to denote the automorphism; if none is provided, the LaTeX symbol is set to `name`
- `is_identity` – (default: `False`) determines whether the constructed object is the identity automorphism, i.e. the identity map of M considered as an automorphism (the identity element of the general linear group)

EXAMPLES:

Automorphism of a rank-2 free module over $\mathbf{Z}$:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M', start_index=1)
sage: a = M.automorphism(name='a', latex_name=r'\alpha') ; a
Automorphism a of the Rank-2 free module M over the Integer Ring
sage: a.parent()
General linear group of the Rank-2 free module M over the Integer Ring
sage: a.parent() is M.general_linear_group()
True
sage: latex(a)
\alpha
```

Setting the components of `a` w.r.t. a basis of module `M`:

```
sage: e = M.basis('e') ; e
Basis (e_1,e_2) on the Rank-2 free module M over the Integer Ring
sage: a[:] = [[1,2],[1,3]]
sage: a.matrix(e)
[1 2]
[1 3]
sage: a(e[1]).display()
a(e_1) = e_1 + e_2
sage: a(e[2]).display()
a(e_2) = 2 e_1 + 3 e_2
```

Actually, the components w.r.t. a given basis can be specified at the construction of the object:

```
sage: a = M.automorphism(matrix=[[1,2],[1,3]], basis=e, name='a',
....:                    latex_name=r'\alpha') ; a
Automorphism a of the Rank-2 free module M over the Integer Ring
sage: a.matrix(e)
[1 2]
[1 3]
```

Since e is the module's default basis, it can be omitted in the argument list:

```
sage: a == M.automorphism(matrix=[[1,2],[1,3]], name='a',
....:                     latex_name=r'\alpha')
True
```

The matrix of the automorphism can be obtained in any basis:

```
sage: f = M.basis('f', from_family=(3*e[1]+4*e[2], 5*e[1]+7*e[2])) ; f
Basis (f_1,f_2) on the Rank-2 free module M over the Integer Ring
sage: a.matrix(f)
[2 3]
[1 2]
```

Automorphisms are tensors of type $(1, 1)$:

```
sage: a.tensor_type()
(1, 1)
sage: a.tensor_rank()
2
```

In particular, they can be displayed as such:

```
sage: a.display(e)
a = e_1*e^1 + 2 e_1*e^2 + e_2*e^1 + 3 e_2*e^2
sage: a.display(f)
a = 2 f_1*f^1 + 3 f_1*f^2 + f_2*f^1 + 2 f_2*f^2
```

The automorphism acting on a module element:

```
sage: v = M([-2,3], name='v') ; v
Element v of the Rank-2 free module M over the Integer Ring
sage: a(v)
Element a(v) of the Rank-2 free module M over the Integer Ring
sage: a(v).display()
a(v) = 4 e_1 + 7 e_2
```

A second automorphism of the module `M` :

```
sage: b = M.automorphism([[0,1],[-1,0]], name='b') ; b
Automorphism b of the Rank-2 free module M over the Integer Ring
sage: b.matrix(e)
[ 0  1]
[-1  0]
sage: b(e[1]).display()
b(e_1) = -e_2
sage: b(e[2]).display()
b(e_2) = e_1
```

The composition of automorphisms is performed via the multiplication operator:

```
sage: s = a*b ; s
Automorphism of the Rank-2 free module M over the Integer Ring
sage: s(v) == a(b(v))
True
sage: s.matrix(f)
[ 11  19]
[ -7 -12]
sage: s.matrix(f) == a.matrix(f) * b.matrix(f)
True
```

It is not commutative:

```
sage: a*b != b*a
True
```

In other words, the parent of `a` and `b` , i.e. the group $\mathrm{GL}(M)$, is not abelian:

```
sage: M.general_linear_group() in CommutativeAdditiveGroups()
False
```

The neutral element for the composition law is the module identity map:

```
sage: id = M.identity_map() ; id
Identity map of the Rank-2 free module M over the Integer Ring
sage: id.parent()
General linear group of the Rank-2 free module M over the Integer Ring
sage: id(v) == v
True
sage: id.matrix(f)
[1 0]
[0 1]
sage: id*a == a
True
sage: a*id == a
True
```

The inverse of an automorphism is obtained via the method *inverse()* , or the operator ~, or the exponent -1:

```
sage: a.inverse()
Automorphism a^(-1) of the Rank-2 free module M over the Integer Ring
sage: a.inverse() is ~a
True
sage: a.inverse() is a^(-1)
True
sage: (a^(-1)).matrix(e)
[ 3 -2]
[-1  1]
sage: a*a^(-1) == id
True
sage: a^(-1)*a == id
True
sage: a^(-1)*s == b
True
sage: (a^(-1))(a(v)) == v
True
```

The module's changes of basis are stored as automorphisms:

```
sage: M.change_of_basis(e,f)
Automorphism of the Rank-2 free module M over the Integer Ring
sage: M.change_of_basis(e,f).parent()
General linear group of the Rank-2 free module M over the Integer Ring
sage: M.change_of_basis(e,f).matrix(e)
[3 5]
[4 7]
sage: M.change_of_basis(f,e) == M.change_of_basis(e,f).inverse()
True
```

The opposite of an automorphism is still an automorphism:

```
sage: -a
Automorphism -a of the Rank-2 free module M over the Integer Ring
sage: (-a).parent()
General linear group of the Rank-2 free module M over the Integer Ring
sage: (-a).matrix(e) == - (a.matrix(e))
True
```

Adding two automorphisms results in a generic type-(1,1) tensor:

```
sage: s = a + b ; s
Type-(1,1) tensor a+b on the Rank-2 free module M over the Integer Ring
sage: s.parent()
Free module of type-(1,1) tensors on the Rank-2 free module M over the
 Integer Ring
sage: a[:], b[:], s[:]
(
[1 2]  [ 0  1]  [1 3]
[1 3], [-1  0], [0 3]
)
```

To get the result as an endomorphism, one has to explicitely convert it via the parent of endormophisms, $\mathrm{End}(M)$:

```
sage: s = End(M)(a+b) ; s
Generic endomorphism of Rank-2 free module M over the Integer Ring
sage: s(v) == a(v) + b(v)
True
sage: s.matrix(e) == a.matrix(e) + b.matrix(e)
True
sage: s.matrix(f) == a.matrix(f) + b.matrix(f)
True
```

add_comp (*basis=None*)

Return the components of `self` w.r.t. a given module basis for assignment, keeping the components w.r.t. other bases.

To delete the components w.r.t. other bases, use the method *set_comp()* instead.

INPUT:

- `basis` – (default: `None`) basis in which the components are defined; if none is provided, the components are assumed to refer to the module's default basis

Warning: If the automorphism has already components in other bases, it is the user's responsability to make sure that the components to be added are consistent with them.

OUTPUT:

- components in the given basis, as an instance of the class *Components*; if such components did not exist previously, they are created

EXAMPLE:

Adding components to an automorphism of a rank-3 free **Z**-module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: a = M.automorphism(name='a')
sage: a[e,:] = [[1,0,0],[0,-1,2],[0,1,-3]]
sage: f = M.basis('f', from_family=(-e[0], 3*e[1]+4*e[2],
....:                               5*e[1]+7*e[2])) ; f
Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer
 Ring
sage: a.add_comp(f)[:] = [[1,0,0], [0, 80, 143], [0, -47, -84]]
```

The components in basis `e` have been kept:

```
sage: a._components # random (dictionary output)
{Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer
 Ring: 2-indices components w.r.t. Basis (e_0,e_1,e_2) on the
 Rank-3 free module M over the Integer Ring,
 Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer
 Ring: 2-indices components w.r.t. Basis (f_0,f_1,f_2) on the
 Rank-3 free module M over the Integer Ring}
```

For the identity map, it is not permitted to invoke *add_comp()* :

```
sage: id = M.identity_map()
sage: id.add_comp(e)
Traceback (most recent call last):
...
TypeError: the components of the identity map cannot be changed
```

Indeed, the components are automatically set by a call to *comp()* :

```
sage: id.comp(e)
Kronecker delta of size 3x3
sage: id.comp(f)
Kronecker delta of size 3x3
```

comp (*basis=None*, *from_basis=None*)

Return the components of `self` w.r.t to a given module basis.

If the components are not known already, they are computed by the tensor change-of-basis formula from components in another basis.

INPUT:

- `basis` – (default: `None`) basis in which the components are required; if none is provided, the components are assumed to refer to the module's default basis
- `from_basis` – (default: `None`) basis from which the required components are computed, via the tensor change-of-basis formula, if they are not known already in the basis `basis` ; if none, a basis from which both the components and a change-of-basis to `basis` are known is selected.

OUTPUT:

- components in the basis `basis` , as an instance of the class *Components* , or, for the identity automorphism, of the subclass *KroneckerDelta*

EXAMPLES:

Components of an automorphism on a rank-3 free $\mathbf{Z}$-module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: a = M.automorphism([[-1,0,0],[0,1,2],[0,1,3]], name='a')
sage: a.components(e)
2-indices components w.r.t. Basis (e_1,e_2,e_3) on the Rank-3 free
 module M over the Integer Ring
sage: a.components(e)[:]
[-1  0  0]
[ 0  1  2]
[ 0  1  3]
```

Since e is the module's default basis, it can be omitted:

```
sage: a.components() is a.components(e)
True
```

A shortcut is `a.comp()` :

```
sage: a.comp() is a.components()
True
sage: a.comp(e) is a.components()
True
```

Components in another basis:

```
sage: f1 = -e[2]
sage: f2 = 4*e[1] + 3*e[3]
sage: f3 = 7*e[1] + 5*e[3]
sage: f = M.basis('f', from_family=(f1,f2,f3))
sage: a.components(f)
2-indices components w.r.t. Basis (f_1,f_2,f_3) on the Rank-3 free
 module M over the Integer Ring
sage: a.components(f)[:]
[  1  -6 -10]
[ -7  83 140]
[  4 -48 -81]
```

Some check of the above matrix:

```
sage: a(f[1]).display(f)
a(f_1) = f_1 - 7 f_2 + 4 f_3
sage: a(f[2]).display(f)
a(f_2) = -6 f_1 + 83 f_2 - 48 f_3
sage: a(f[3]).display(f)
a(f_3) = -10 f_1 + 140 f_2 - 81 f_3
```

Components of the identity map:

```
sage: id = M.identity_map()
sage: id.components(e)
Kronecker delta of size 3x3
sage: id.components(e)[:]
[1 0 0]
[0 1 0]
[0 0 1]
sage: id.components(f)
Kronecker delta of size 3x3
sage: id.components(f)[:]
[1 0 0]
[0 1 0]
[0 0 1]
```

components (*basis=None*, *from_basis=None*)

Return the components of `self` w.r.t to a given module basis.

If the components are not known already, they are computed by the tensor change-of-basis formula from components in another basis.

INPUT:

- `basis` – (default: `None`) basis in which the components are required; if none is provided, the components are assumed to refer to the module's default basis

- `from_basis` – (default: `None`) basis from which the required components are computed, via the tensor change-of-basis formula, if they are not known already in the basis `basis`; if none, a basis from which both the components and a change-of-basis to `basis` are known is selected.

OUTPUT:

- components in the basis `basis`, as an instance of the class *Components*, or, for the identity automorphism, of the subclass *KroneckerDelta*

EXAMPLES:

Components of an automorphism on a rank-3 free **Z**-module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: a = M.automorphism([[-1,0,0],[0,1,2],[0,1,3]], name='a')
sage: a.components(e)
2-indices components w.r.t. Basis (e_1,e_2,e_3) on the Rank-3 free
 module M over the Integer Ring
sage: a.components(e)[:]
[-1  0  0]
[ 0  1  2]
[ 0  1  3]
```

Since e is the module's default basis, it can be omitted:

```
sage: a.components() is a.components(e)
True
```

A shortcut is `a.comp()`:

```
sage: a.comp() is a.components()
True
sage: a.comp(e) is a.components()
True
```

Components in another basis:

```
sage: f1 = -e[2]
sage: f2 = 4*e[1] + 3*e[3]
sage: f3 = 7*e[1] + 5*e[3]
sage: f = M.basis('f', from_family=(f1,f2,f3))
sage: a.components(f)
2-indices components w.r.t. Basis (f_1,f_2,f_3) on the Rank-3 free
 module M over the Integer Ring
sage: a.components(f)[:]
[  1  -6 -10]
[ -7  83 140]
[  4 -48 -81]
```

Some check of the above matrix:

```
sage: a(f[1]).display(f)
a(f_1) = f_1 - 7 f_2 + 4 f_3
sage: a(f[2]).display(f)
a(f_2) = -6 f_1 + 83 f_2 - 48 f_3
sage: a(f[3]).display(f)
a(f_3) = -10 f_1 + 140 f_2 - 81 f_3
```

Components of the identity map:

```
sage: id = M.identity_map()
sage: id.components(e)
Kronecker delta of size 3x3
sage: id.components(e)[:]
[1 0 0]
[0 1 0]
[0 0 1]
sage: id.components(f)
Kronecker delta of size 3x3
sage: id.components(f)[:]
[1 0 0]
[0 1 0]
[0 0 1]
```

det ( )

Return the determinant of `self` .

OUTPUT:

- element of the base ring of the module on which `self` is defined, equal to the determinant of `self` .

EXAMPLES:

Determinant of an automorphism on a **Z**-module of rank 2:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M')
sage: e = M.basis('e')
sage: a = M.automorphism([[4,7],[3,5]], name='a')
sage: a.matrix(e)
[4 7]
[3 5]
sage: a.det()
-1
sage: det(a)
-1
sage: ~a.det()  # determinant of the inverse automorphism
-1
sage: id = M.identity_map()
sage: id.det()
1
```

inverse ( )

Return the inverse automorphism.

OUTPUT:

- instance of *FreeModuleAutomorphism* representing the automorphism that is the inverse of `self` .

EXAMPLES:

Inverse of an automorphism of a rank-3 free module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: a = M.automorphism(name='a')
sage: a[e,:] = [[1,0,0],[0,-1,2],[0,1,-3]]
sage: a.inverse()
Automorphism a^(-1) of the Rank-3 free module M over the Integer
 Ring
sage: a.inverse().parent()
```

```
General linear group of the Rank-3 free module M over the Integer
 Ring
```

Check that `a.inverse()` is indeed the inverse automorphism:

```
sage: a.inverse() * a
Identity map of the Rank-3 free module M over the Integer Ring
sage: a * a.inverse()
Identity map of the Rank-3 free module M over the Integer Ring
sage: a.inverse().inverse() == a
True
```

Another check is:

```
sage: a.inverse().matrix(e)
[ 1  0  0]
[ 0 -3 -2]
[ 0 -1 -1]
sage: a.inverse().matrix(e) == (a.matrix(e))^(-1)
True
```

The inverse is cached (as long as `a` is not modified):

```
sage: a.inverse() is a.inverse()
True
```

If `a` is modified, the inverse is automatically recomputed:

```
sage: a[0,0] = -1
sage: a.matrix(e)
[-1  0  0]
[ 0 -1  2]
[ 0  1 -3]
sage: a.inverse().matrix(e)  # compare with above
[-1  0  0]
[ 0 -3 -2]
[ 0 -1 -1]
```

Shortcuts for *inverse()* are the operator `~` and the exponent `-1`:

```
sage: ~a is a.inverse()
True
sage: a^(-1) is a.inverse()
True
```

The inverse of the identity map is of course itself:

```
sage: id = M.identity_map()
sage: id.inverse() is id
True
```

and we have:

```
sage: a*a^(-1) == id
True
sage: a^(-1)*a == id
True
```

matrix (*basis1=None*, *basis2=None*)

Return the matrix of `self` w.r.t to a pair of bases.

If the matrix is not known already, it is computed from the matrix in another pair of bases by means of the change-of-basis formula.

INPUT:

- `basis1` – (default: `None`) basis of the free module on which `self` is defined; if none is provided, the module's default basis is assumed
- `basis2` – (default: `None`) basis of the free module on which `self` is defined; if none is provided, `basis2` is set to `basis1`

OUTPUT:

- the matrix representing representing the automorphism `self` w.r.t to bases `basis1` and `basis2`; more precisely, the columns of this matrix are formed by the components w.r.t. `basis2` of the images of the elements of `basis1`.

EXAMPLES:

Matrices of an automorphism of a rank-3 free $\mathbf{Z}$-module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M', start_index=1)
sage: e = M.basis('e')
sage: a = M.automorphism([[-1,0,0],[0,1,2],[0,1,3]], name='a')
sage: a.matrix(e)
[-1  0  0]
[ 0  1  2]
[ 0  1  3]
sage: a.matrix()
[-1  0  0]
[ 0  1  2]
[ 0  1  3]
sage: f = M.basis('f', from_family=(-e[2], 4*e[1]+3*e[3],  7*e[1]+5*e[3])) ; f
Basis (f_1,f_2,f_3) on the Rank-3 free module M over the Integer Ring
sage: a.matrix(f)
[  1  -6 -10]
[ -7  83 140]
[  4 -48 -81]
```

Check of the above matrix:

```
sage: a(f[1]).display(f)
a(f_1) = f_1 - 7 f_2 + 4 f_3
sage: a(f[2]).display(f)
a(f_2) = -6 f_1 + 83 f_2 - 48 f_3
sage: a(f[3]).display(f)
a(f_3) = -10 f_1 + 140 f_2 - 81 f_3
```

Check of the change-of-basis formula:

```
sage: P = M.change_of_basis(e,f).matrix(e)
sage: a.matrix(f) == P^(-1) * a.matrix(e) * P
True
```

Check that the matrix of the product of two automorphisms is the product of their matrices:

```
sage: b = M.change_of_basis(e,f) ; b
Automorphism of the Rank-3 free module M over the Integer Ring
```

```
sage: b.matrix(e)
[ 0  4  7]
[-1  0  0]
[ 0  3  5]
sage: (a*b).matrix(e) == a.matrix(e) * b.matrix(e)
True
```

Check that the matrix of the inverse automorphism is the inverse of the automorphism's matrix:

```
sage: (~a).matrix(e)
[-1  0  0]
[ 0  3 -2]
[ 0 -1  1]
sage: (~a).matrix(e) == ~(a.matrix(e))
True
```

Matrices of the identity map:

```
sage: id = M.identity_map()
sage: id.matrix(e)
[1 0 0]
[0 1 0]
[0 0 1]
sage: id.matrix(f)
[1 0 0]
[0 1 0]
[0 0 1]
```

set_comp (*basis=None*)

Return the components of `self` w.r.t. a given module basis for assignment.

The components with respect to other bases are deleted, in order to avoid any inconsistency. To keep them, use the method *add_comp()* instead.

INPUT:

- `basis` – (default: `None`) basis in which the components are defined; if none is provided, the components are assumed to refer to the module's default basis

OUTPUT:

- components in the given basis, as an instance of the class *Components*; if such components did not exist previously, they are created.

EXAMPLE:

Setting the components of an automorphism of a rank-3 free $\mathbf{Z}$-module:

```
sage: M = FiniteRankFreeModule(ZZ, 3, name='M')
sage: e = M.basis('e')
sage: a = M.automorphism(name='a')
sage: a.set_comp(e)
2-indices components w.r.t. Basis (e_0,e_1,e_2) on the Rank-3 free
 module M over the Integer Ring
sage: a.set_comp(e)[:] = [[1,0,0],[0,1,2],[0,1,3]]
sage: a.matrix(e)
[1 0 0]
[0 1 2]
[0 1 3]
```

Since `e` is the module's default basis, one has:

```
sage: a.set_comp() is a.set_comp(e)
True
```

The method *set_comp()* can be used to modify a single component:

```
sage: a.set_comp(e)[0,0] = -1
sage: a.matrix(e)
[-1  0  0]
[ 0  1  2]
[ 0  1  3]
```

A short cut to *set_comp()* is the bracket operator, with the basis as first argument:

```
sage: a[e,:] = [[1,0,0],[0,-1,2],[0,1,-3]]
sage: a.matrix(e)
[ 1  0  0]
[ 0 -1  2]
[ 0  1 -3]
sage: a[e,0,0] = -1
sage: a.matrix(e)
[-1  0  0]
[ 0 -1  2]
[ 0  1 -3]
```

The call to *set_comp()* erases the components previously defined in other bases; to keep them, use the method *add_comp()* instead:

```
sage: f = M.basis('f', from_family=(-e[0], 3*e[1]+4*e[2],
....:                               5*e[1]+7*e[2])) ; f
Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer
 Ring
sage: a._components
{Basis (e_0,e_1,e_2) on the Rank-3 free module M over the Integer
 Ring: 2-indices components w.r.t. Basis (e_0,e_1,e_2) on the
 Rank-3 free module M over the Integer Ring}
sage: a.set_comp(f)[:] = [[-1,0,0], [0,1,0], [0,0,-1]]
```

The components w.r.t. basis `e` have been erased:

```
sage: a._components
{Basis (f_0,f_1,f_2) on the Rank-3 free module M over the Integer
 Ring: 2-indices components w.r.t. Basis (f_0,f_1,f_2) on the
 Rank-3 free module M over the Integer Ring}
```

Of course, they can be computed from those in basis `f` by means of a change-of-basis formula, via the method *comp()* or *matrix()* :

```
sage: a.matrix(e)
[ -1   0   0]
[  0  41 -30]
[  0  56 -41]
```

For the identity map, it is not permitted to set components:

```
sage: id = M.identity_map()
sage: id.set_comp(e)
```

```
Traceback (most recent call last):
...
TypeError: the components of the identity map cannot be changed
```

Indeed, the components are automatically set by a call to *comp()* :

```
sage: id.comp(e)
Kronecker delta of size 3x3
sage: id.comp(f)
Kronecker delta of size 3x3
```

trace ( )

Return the trace of `self` .

OUTPUT:

- element of the base ring of the module on which `self` is defined, equal to the trace of `self` .

EXAMPLES:

Trace of an automorphism on a **Z**-module of rank 2:

```
sage: M = FiniteRankFreeModule(ZZ, 2, name='M')
sage: e = M.basis('e')
sage: a = M.automorphism([[4,7],[3,5]], name='a')
sage: a.matrix(e)
[4 7]
[3 5]
sage: a.trace()
9
sage: id = M.identity_map()
sage: id.trace()
2
```

CHAPTER

SIX

COMPONENTS AS INDEXED SETS OF RING ELEMENTS

The class *Components* is a technical class to take in charge the storage and manipulation of **indexed elements of a commutative ring** that represent the components of some "mathematical entity" with respect to some "frame". Examples of *entity/frame* are *vector/vector-space basis* or *vector field/vector frame on some manifold*. More generally, the components can be those of a tensor on a free module or those of a tensor field on a manifold. They can also be non-tensorial quantities, like connection coefficients or structure coefficients of a vector frame.

The individual components are assumed to belong to a given commutative ring and are labelled by *indices*, which are *tuples of integers*. The following operations are implemented on components with respect to a given frame:

- arithmetics (addition, subtraction, multiplication by a ring element)
- handling of symmetries or antisymmetries on the indices
- symmetrization and antisymmetrization
- tensor product
- contraction

Various subclasses of class *Components* are

- *CompWithSym* for components with symmetries or antisymmetries w.r.t. index permutations
  - *CompFullySym* for fully symmetric components w.r.t. index permutations
    - *KroneckerDelta* for the Kronecker delta symbol
  - *CompFullyAntiSym* for fully antisymmetric components w.r.t. index permutations

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version
- Joris Vankerschaver (2010): for the idea of storing only the non-zero components as dictionaries, whose keys are the component indices (see class `DifferentialForm`)
- Marco Mancini (2015) : parallelization of some computations

EXAMPLES:

Set of components with 2 indices on a 3-dimensional vector space, the frame being some basis of the vector space:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ,3)
sage: basis = V.basis() ; basis
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
```

```
sage: c = Components(QQ, basis, 2) ; c
2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
```

Actually, the frame can be any object that has some length, i.e. on which the function `len()` can be called:

```
sage: basis1 = V.gens() ; basis1
((1, 0, 0), (0, 1, 0), (0, 0, 1))
sage: c1 = Components(QQ, basis1, 2) ; c1
2-indices components w.r.t. ((1, 0, 0), (0, 1, 0), (0, 0, 1))
sage: basis2 = ['a', 'b' , 'c']
sage: c2 = Components(QQ, basis2, 2) ; c2
2-indices components w.r.t. ['a', 'b', 'c']
```

A just created set of components is initialized to zero:

```
sage: c.is_zero()
True
sage: c == 0
True
```

This can also be checked on the list of components, which is returned by the operator `[:]` :

```
sage: c[:]
[0 0 0]
[0 0 0]
[0 0 0]
```

Individual components are accessed by providing their indices inside square brackets:

```
sage: c[1,2] = -3
sage: c[:]
[ 0  0  0]
[ 0  0 -3]
[ 0  0  0]
sage: v = Components(QQ, basis, 1)
sage: v[:]
[0, 0, 0]
sage: v[0]
0
sage: v[:] = (-1,3,2)
sage: v[:]
[-1, 3, 2]
sage: v[0]
-1
```

Sets of components with 2 indices can be converted into a matrix:

```
sage: matrix(c)
[ 0  0  0]
[ 0  0 -3]
[ 0  0  0]
sage: matrix(c).parent()
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
```

By default, the indices range from 0 to $n-1$, where n is the length of the frame. This can be changed via the argument `start_index` in the *Components* constructor:

```
sage: v1 = Components(QQ, basis, 1, start_index=1)
sage: v1[:]
[0, 0, 0]
sage: v1[0]
Traceback (most recent call last):
...
IndexError: index out of range: 0 not in [1, 3]
sage: v1[1]
0
sage: v1[:] = v[:]  # list copy of all components
sage: v1[:]
[-1, 3, 2]
sage: v1[1], v1[2], v1[3]
(-1, 3, 2)
sage: v[0], v[1], v[2]
(-1, 3, 2)
```

If some formatter function or unbound method is provided via the argument `output_formatter` in the *Components* constructor, it is used to change the ouput of the access operator `[...]` :

```
sage: a = Components(QQ, basis, 2, output_formatter=Rational.numerical_approx)
sage: a[1,2] = 1/3
sage: a[1,2]
0.333333333333333
```

The format can be passed to the formatter as the last argument of the access operator `[...]` :

```
sage: a[1,2,10] # here the format is 10, for 10 bits of precision
0.33
sage: a[1,2,100]
0.33333333333333333333333333333
```

The raw (unformatted) components are then accessed by the double bracket operator:

```
sage: a[[1,2]]
1/3
```

For sets of components declared without any output formatter, there is no difference between `[...]` and `[[...]]` :

```
sage: c[1,2] = 1/3
sage: c[1,2], c[[1,2]]
(1/3, 1/3)
```

The formatter is also used for the complete list of components:

```
sage: a[:]
[0.000000000000000 0.000000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000 0.333333333333333]
[0.000000000000000 0.000000000000000 0.000000000000000]
sage: a[:,10] # with a format different from the default one (53 bits)
[0.00 0.00 0.00]
[0.00 0.00 0.33]
[0.00 0.00 0.00]
```

The complete list of components in raw form can be recovered by the double bracket operator, replacing `:` by `slice(None)` (since `a[[:]]` generates a Python syntax error):

```
sage: a[[slice(None)]]
[  0   0   0]
[  0   0 1/3]
[  0   0   0]
```

Another example of formatter: the Python built-in function `str()` to generate string outputs:

```
sage: b = Components(QQ, V.basis(), 1, output_formatter=str)
sage: b[:] = (1, 0, -4)
sage: b[:]
['1', '0', '-4']
```

For such a formatter, 2-indices components are no longer displayed as a matrix:

```
sage: b = Components(QQ, basis, 2, output_formatter=str)
sage: b[0,1] = 1/3
sage: b[:]
[['0', '1/3', '0'], ['0', '0', '0'], ['0', '0', '0']]
```

But unformatted outputs still are:

```
sage: b[[slice(None)]]
[  0 1/3   0]
[  0   0   0]
[  0   0   0]
```

Internally, the components are stored as a dictionary (`_comp`) whose keys are the indices; only the non-zero components are stored:

```
sage: a[:]
[0.000000000000000 0.000000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000 0.333333333333333]
[0.000000000000000 0.000000000000000 0.000000000000000]
sage: a._comp
{(1, 2): 1/3}
sage: v[:] = (-1, 0, 3)
sage: v._comp  # random output order of the component dictionary
{(0,): -1, (2,): 3}
```

In case of symmetries, only non-redundant components are stored:

```
sage: from sage.tensor.modules.comp import CompFullyAntiSym
sage: c = CompFullyAntiSym(QQ, basis, 2)
sage: c[0,1] = 3
sage: c[:]
[ 0  3  0]
[-3  0  0]
[ 0  0  0]
sage: c._comp
{(0, 1): 3}
```

class sage.tensor.modules.comp. **CompFullyAntiSym** (*ring*, *frame*, *nb_indices*, *start_index=0*, *output_formatter=None*)

Bases: *sage.tensor.modules.comp.CompWithSym*

Indexed set of ring elements forming some components with respect to a given "frame" that are fully antisymmetric with respect to any permutation of the indices.

The "frame" can be a basis of some vector space or a vector frame on some manifold (i.e. a field of bases). The stored quantities can be tensor components or non-tensorial quantities.

INPUT:

- `ring` – commutative ring in which each component takes its value
- `frame` – frame with respect to which the components are defined; whatever type `frame` is, it should have some method `__len__()` implemented, so that `len(frame)` returns the dimension, i.e. the size of a single index range
- `nb_indices` – number of indices labeling the components
- `start_index` – (default: 0) first value of a single index; accordingly a component index i must obey `start_index <= i <= start_index + dim -1`, where `dim = len(frame)`.
- `output_formatter` – (default: `None`) function or unbound method called to format the output of the component access operator `[...]` (method __getitem__); `output_formatter` must take 1 or 2 arguments: the 1st argument must be an instance of `ring` and the second one, if any, some format specification.

EXAMPLES:

Antisymmetric components with 2 indices on a 3-dimensional space:

```
sage: from sage.tensor.modules.comp import CompWithSym, CompFullyAntiSym
sage: V = VectorSpace(QQ, 3)
sage: c = CompFullyAntiSym(QQ, V.basis(), 2)
sage: c[0,1], c[0,2], c[1,2] = 3, 1/2, -1
sage: c[:]  # note that all components have been set according to the antisymmetry
[   0    3  1/2]
[  -3    0   -1]
[-1/2    1    0]
```

Internally, only non-redundant and non-zero components are stored:

```
sage: c._comp # random output order of the component dictionary
{(0, 1): 3, (0, 2): 1/2, (1, 2): -1}
```

Same thing, but with the starting index set to 1:

```
sage: c1 = CompFullyAntiSym(QQ, V.basis(), 2, start_index=1)
sage: c1[1,2], c1[1,3], c1[2,3] = 3, 1/2, -1
sage: c1[:]
[   0    3  1/2]
[  -3    0   -1]
[-1/2    1    0]
```

The values stored in `c` and `c1` are equal:

```
sage: c1[:] == c[:]
True
```

but not `c` and `c1`, since their starting indices differ:

```
sage: c1 == c
False
```

Fully antisymmetric components with 3 indices on a 3-dimensional space:

```
sage: a = CompFullyAntiSym(QQ, V.basis(), 3)
sage: a[0,1,2] = 3  # the only independent component in dimension 3
sage: a[:]
[[[0, 0, 0], [0, 0, 3], [0, -3, 0]],
 [[0, 0, -3], [0, 0, 0], [3, 0, 0]],
 [[0, 3, 0], [-3, 0, 0], [0, 0, 0]]]
```

Setting a nonzero value incompatible with the antisymmetry results in an error:

```
sage: a[0,1,0] = 4
Traceback (most recent call last):
...
ValueError: by antisymmetry, the component cannot have a nonzero value for the␣
↪indices (0, 1, 0)
sage: a[0,1,0] = 0   # OK
sage: a[2,0,1] = 3   # OK
```

The full antisymmetry is preserved by the arithmetics:

```
sage: b = CompFullyAntiSym(QQ, V.basis(), 3)
sage: b[0,1,2] = -4
sage: s = a + 2*b ; s
Fully antisymmetric 3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: a[:], b[:], s[:]
([[[0, 0, 0], [0, 0, 3], [0, -3, 0]],
  [[0, 0, -3], [0, 0, 0], [3, 0, 0]],
  [[0, 3, 0], [-3, 0, 0], [0, 0, 0]]],
 [[[0, 0, 0], [0, 0, -4], [0, 4, 0]],
  [[0, 0, 4], [0, 0, 0], [-4, 0, 0]],
  [[0, -4, 0], [4, 0, 0], [0, 0, 0]]],
 [[[0, 0, 0], [0, 0, -5], [0, 5, 0]],
  [[0, 0, 5], [0, 0, 0], [-5, 0, 0]],
  [[0, -5, 0], [5, 0, 0], [0, 0, 0]]])
```

It is lost if the added object is not fully antisymmetric:

```
sage: b1 = CompWithSym(QQ, V.basis(), 3, antisym=(0,1))  # b1 has only␣
↪antisymmetry on index positions (0,1)
sage: b1[0,1,2] = -4
sage: s = a + 2*b1 ; s  # the result has the same symmetry as b1:
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 1)
sage: a[:], b1[:], s[:]
([[[0, 0, 0], [0, 0, 3], [0, -3, 0]],
  [[0, 0, -3], [0, 0, 0], [3, 0, 0]],
  [[0, 3, 0], [-3, 0, 0], [0, 0, 0]]],
 [[[0, 0, 0], [0, 0, -4], [0, 0, 0]],
  [[0, 0, 4], [0, 0, 0], [0, 0, 0]],
  [[0, 0, 0], [0, 0, 0], [0, 0, 0]]],
 [[[0, 0, 0], [0, 0, -5], [0, -3, 0]],
```

```
  [[0, 0, 5], [0, 0, 0], [3, 0, 0]],
  [[0, 3, 0], [-3, 0, 0], [0, 0, 0]]])
sage: s = 2*b1 + a ; s
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 1)
sage: 2*b1 + a == a + 2*b1
True
```

class sage.tensor.modules.comp. **CompFullySym** (*ring*, *frame*, *nb_indices*, *start_index=0*, *output_formatter=None*)

Bases: *sage.tensor.modules.comp.CompWithSym*

Indexed set of ring elements forming some components with respect to a given "frame" that are fully symmetric with respect to any permutation of the indices.

The "frame" can be a basis of some vector space or a vector frame on some manifold (i.e. a field of bases). The stored quantities can be tensor components or non-tensorial quantities.

INPUT:

- `ring` – commutative ring in which each component takes its value
- `frame` – frame with respect to which the components are defined; whatever type `frame` is, it should have some method `__len__()` implemented, so that `len(frame)` returns the dimension, i.e. the size of a single index range
- `nb_indices` – number of indices labeling the components
- `start_index` – (default: 0) first value of a single index; accordingly a component index i must obey `start_index <= i <= start_index + dim -1`, where `dim = len(frame)`.
- `output_formatter` – (default: `None`) function or unbound method called to format the output of the component access operator `[...]` (method __getitem__); `output_formatter` must take 1 or 2 arguments: the 1st argument must be an instance of `ring` and the second one, if any, some format specification.

EXAMPLES:

Symmetric components with 2 indices on a 3-dimensional space:

```
sage: from sage.tensor.modules.comp import CompFullySym, CompWithSym
sage: V = VectorSpace(QQ, 3)
sage: c = CompFullySym(QQ, V.basis(), 2)
sage: c[0,0], c[0,1], c[1,2] = 1, -2, 3
sage: c[:] # note that c[1,0] and c[2,1] have been updated automatically (by␣
↪symmetry)
[ 1 -2  0]
[-2  0  3]
[ 0  3  0]
```

Internally, only non-redundant and non-zero components are stored:

```
sage: c._comp  # random output order of the component dictionary
{(0, 0): 1, (0, 1): -2, (1, 2): 3}
```

Same thing, but with the starting index set to 1:

```
sage: c1 = CompFullySym(QQ, V.basis(), 2, start_index=1)
sage: c1[1,1], c1[1,2], c1[2,3] = 1, -2, 3
sage: c1[:]
[ 1 -2  0]
[-2  0  3]
[ 0  3  0]
```

The values stored in `c` and `c1` are equal:

```
sage: c1[:] == c[:]
True
```

but not `c` and `c1`, since their starting indices differ:

```
sage: c1 == c
False
```

Fully symmetric components with 3 indices on a 3-dimensional space:

```
sage: a = CompFullySym(QQ, V.basis(), 3)
sage: a[0,1,2] = 3
sage: a[:]
[[[0, 0, 0], [0, 0, 3], [0, 3, 0]],
 [[0, 0, 3], [0, 0, 0], [3, 0, 0]],
 [[0, 3, 0], [3, 0, 0], [0, 0, 0]]]
sage: a[0,1,0] = 4
sage: a[:]
[[[0, 4, 0], [4, 0, 3], [0, 3, 0]],
 [[4, 0, 3], [0, 0, 0], [3, 0, 0]],
 [[0, 3, 0], [3, 0, 0], [0, 0, 0]]]
```

The full symmetry is preserved by the arithmetics:

```
sage: b = CompFullySym(QQ, V.basis(), 3)
sage: b[0,0,0], b[0,1,0], b[1,0,2], b[1,2,2] = -2, 3, 1, -5
sage: s = a + 2*b ; s
Fully symmetric 3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: a[:], b[:], s[:]
([[[0, 4, 0], [4, 0, 3], [0, 3, 0]],
  [[4, 0, 3], [0, 0, 0], [3, 0, 0]],
  [[0, 3, 0], [3, 0, 0], [0, 0, 0]]],
 [[[-2, 3, 0], [3, 0, 1], [0, 1, 0]],
  [[3, 0, 1], [0, 0, 0], [1, 0, -5]],
  [[0, 1, 0], [1, 0, -5], [0, -5, 0]]],
 [[[-4, 10, 0], [10, 0, 5], [0, 5, 0]],
  [[10, 0, 5], [0, 0, 0], [5, 0, -10]],
  [[0, 5, 0], [5, 0, -10], [0, -10, 0]]])
```

It is lost if the added object is not fully symmetric:

```
sage: b1 = CompWithSym(QQ, V.basis(), 3, sym=(0,1))  # b1 has only symmetry on␣
↪index positions (0,1)
sage: b1[0,0,0], b1[0,1,0], b1[1,0,2], b1[1,2,2] = -2, 3, 1, -5
sage: s = a + 2*b1 ; s  # the result has the same symmetry as b1:
```

```
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1)
sage: a[:], b1[:], s[:]
([[[0, 4, 0], [4, 0, 3], [0, 3, 0]],
  [[4, 0, 3], [0, 0, 0], [3, 0, 0]],
  [[0, 3, 0], [3, 0, 0], [0, 0, 0]]],
 [[[-2, 0, 0], [3, 0, 1], [0, 0, 0]],
  [[3, 0, 1], [0, 0, 0], [0, 0, -5]],
  [[0, 0, 0], [0, 0, -5], [0, 0, 0]]],
 [[[-4, 4, 0], [10, 0, 5], [0, 3, 0]],
  [[10, 0, 5], [0, 0, 0], [3, 0, -10]],
  [[0, 3, 0], [3, 0, -10], [0, 0, 0]]])
sage: s = 2*b1 + a ; s
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1)
sage: 2*b1 + a == a + 2*b1
True
```

class sage.tensor.modules.comp. **CompWithSym** (*ring*, *frame*, *nb_indices*, *start_index=0*, *output_formatter=None*, *sym=None*, *antisym=None*)

Bases: *sage.tensor.modules.comp.Components*

Indexed set of ring elements forming some components with respect to a given "frame", with symmetries or antisymmetries regarding permutations of the indices.

The "frame" can be a basis of some vector space or a vector frame on some manifold (i.e. a field of bases). The stored quantities can be tensor components or non-tensorial quantities, such as connection coefficients or structure coefficients.

Subclasses of *CompWithSym* are

- *CompFullySym* for fully symmetric components.
- *CompFullyAntiSym* for fully antisymmetric components.

INPUT:

- `ring` – commutative ring in which each component takes its value
- `frame` – frame with respect to which the components are defined; whatever type `frame` is, it should have some method `__len__()` implemented, so that `len(frame)` returns the dimension, i.e. the size of a single index range
- `nb_indices` – number of indices labeling the components
- `start_index` – (default: 0) first value of a single index; accordingly a component index i must obey `start_index <= i <= start_index + dim -1` , where `dim = len(frame)` .
- `output_formatter` – (default: `None`) function or unbound method called to format the output of the component access operator `[...]` (method __getitem__); `output_formatter` must take 1 or 2 arguments: the 1st argument must be an instance of `ring` and the second one, if any, some format specification.

- `sym` – (default: `None`) a symmetry or a list of symmetries among the indices: each symmetry is described by a tuple containing the positions of the involved indices, with the convention `position=0` for the first slot; for instance:
  - `sym = (0,1)` for a symmetry between the 1st and 2nd indices
  - `sym = [(0,2),(1,3,4)]` for a symmetry between the 1st and 3rd indices and a symmetry between the 2nd, 4th and 5th indices.
- `antisym` – (default: `None`) antisymmetry or list of antisymmetries among the indices, with the same convention as for `sym`

EXAMPLES:

Symmetric components with 2 indices:

```
sage: from sage.tensor.modules.comp import Components, CompWithSym
sage: V = VectorSpace(QQ,3)
sage: c = CompWithSym(QQ, V.basis(), 2, sym=(0,1))  # for demonstration only: it␣
↪is preferable to use CompFullySym in this case
sage: c[0,1] = 3
sage: c[:]  # note that c[1,0] has been set automatically
[0 3 0]
[3 0 0]
[0 0 0]
```

Antisymmetric components with 2 indices:

```
sage: c = CompWithSym(QQ, V.basis(), 2, antisym=(0,1))  # for demonstration only:␣
↪it is preferable to use CompFullyAntiSym in this case
sage: c[0,1] = 3
sage: c[:]  # note that c[1,0] has been set automatically
[ 0  3  0]
[-3  0  0]
[ 0  0  0]
```

Internally, only non-redundant components are stored:

```
sage: c._comp
{(0, 1): 3}
```

Components with 6 indices, symmetric among 3 indices (at position $(0,1,5)$) and antisymmetric among 2 indices (at position $(2,4)$):

```
sage: c = CompWithSym(QQ, V.basis(), 6, sym=(0,1,5), antisym=(2,4))
sage: c[0,1,2,0,1,2] = 3
sage: c[1,0,2,0,1,2]  # symmetry between indices in position 0 and 1
3
sage: c[2,1,2,0,1,0]  # symmetry between indices in position 0 and 5
3
sage: c[0,2,2,0,1,1]  # symmetry between indices in position 1 and 5
3
sage: c[0,1,1,0,2,2]  # antisymmetry between indices in position 2 and 4
-3
```

Components with 4 indices, antisymmetric with respect to the first pair of indices as well as with the second pair of indices:

```
sage: c = CompWithSym(QQ, V.basis(), 4, antisym=[(0,1),(2,3)])
sage: c[0,1,0,1] = 3
```

```
sage: c[1,0,0,1]  # antisymmetry on the first pair of indices
-3
sage: c[0,1,1,0]  # antisymmetry on the second pair of indices
-3
sage: c[1,0,1,0]  # consequence of the above
3
```

ARITHMETIC EXAMPLES

Addition of a symmetric set of components with a non-symmetric one: the symmetry is lost:

```
sage: V = VectorSpace(QQ, 3)
sage: a = Components(QQ, V.basis(), 2)
sage: a[:] = [[1,-2,3], [4,5,-6], [-7,8,9]]
sage: b = CompWithSym(QQ, V.basis(), 2, sym=(0,1))  # for demonstration only: it␣
↪is preferable to declare b = CompFullySym(QQ, V.basis(), 2)
sage: b[0,0], b[0,1], b[0,2] = 1, 2, 3
sage: b[1,1], b[1,2] = 5, 7
sage: b[2,2] = 11
sage: s = a + b ; s
2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: a[:], b[:], s[:]
(
[ 1 -2  3]  [ 1  2  3]  [ 2  0  6]
[ 4  5 -6]  [ 2  5  7]  [ 6 10  1]
[-7  8  9], [ 3  7 11], [-4 15 20]
)
sage: a + b == b + a
True
```

Addition of two symmetric set of components: the symmetry is preserved:

```
sage: c = CompWithSym(QQ, V.basis(), 2, sym=(0,1)) # for demonstration only: it␣
↪is preferable to declare c = CompFullySym(QQ, V.basis(), 2)
sage: c[0,0], c[0,1], c[0,2] = -4, 7, -8
sage: c[1,1], c[1,2] = 2, -4
sage: c[2,2] = 2
sage: s = b + c ; s
2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1)
sage: b[:], c[:], s[:]
(
[ 1  2  3]  [-4  7 -8]  [-3  9 -5]
[ 2  5  7]  [ 7  2 -4]  [ 9  7  3]
[ 3  7 11], [-8 -4  2], [-5  3 13]
)
sage: b + c == c + b
True
```

Check of the addition with counterparts not declared symmetric:

```
sage: bn = Components(QQ, V.basis(), 2)
sage: bn[:] = b[:]
sage: bn == b
True
sage: cn = Components(QQ, V.basis(), 2)
sage: cn[:] = c[:]
sage: cn == c
True
sage: bn + cn == b + c
True
```

Addition of an antisymmetric set of components with a non-symmetric one: the antisymmetry is lost:

```
sage: d = CompWithSym(QQ, V.basis(), 2, antisym=(0,1))  # for demonstration only:␣
↪it is preferable to declare d = CompFullyAntiSym(QQ, V.basis(), 2)
sage: d[0,1], d[0,2], d[1,2] = 4, -1, 3
sage: s = a + d ; s
2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: a[:], d[:], s[:]
(
[ 1 -2  3]  [ 0  4 -1]  [ 1  2  2]
[ 4  5 -6]  [-4  0  3]  [ 0  5 -3]
[-7  8  9], [ 1 -3  0], [-6  5  9]
)
sage: d + a == a + d
True
```

Addition of two antisymmetric set of components: the antisymmetry is preserved:

```
sage: e = CompWithSym(QQ, V.basis(), 2, antisym=(0,1))  # for demonstration only:␣
↪it is preferable to declare e = CompFullyAntiSym(QQ, V.basis(), 2)
sage: e[0,1], e[0,2], e[1,2] = 2, 3, -1
sage: s = d + e ; s
2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 1)
sage: d[:], e[:], s[:]
(
[ 0  4 -1]  [ 0  2  3]  [ 0  6  2]
[-4  0  3]  [-2  0 -1]  [-6  0  2]
[ 1 -3  0], [-3  1  0], [-2 -2  0]
)
sage: e + d == d + e
True
```

antisymmetrize (**pos*)

Antisymmetrization over the given index positions.

INPUT:

- `pos` – list of index positions involved in the antisymmetrization (with the convention `position=0`

for the first slot); if none, the antisymmetrization is performed over all the indices

OUTPUT:

- an instance of *CompWithSym* describing the antisymmetrized components

EXAMPLES:

Antisymmetrization of 3-indices components on a 3-dimensional space:

```
sage: from sage.tensor.modules.comp import Components, CompWithSym, \
...    CompFullySym, CompFullyAntiSym
sage: V = VectorSpace(QQ, 3)
sage: a = Components(QQ, V.basis(), 1)
sage: a[:] = (-2,1,3)
sage: b = CompFullyAntiSym(QQ, V.basis(), 2)
sage: b[0,1], b[0,2], b[1,2] = (4,1,2)
sage: c = a*b ; c   # tensor product of a by b
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (1, 2)
sage: s = c.antisymmetrize() ; s
Fully antisymmetric 3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: c[:], s[:]
([[[0, -8, -2], [8, 0, -4], [2, 4, 0]],
  [[0, 4, 1], [-4, 0, 2], [-1, -2, 0]],
  [[0, 12, 3], [-12, 0, 6], [-3, -6, 0]]],
 [[[0, 0, 0], [0, 0, 7/3], [0, -7/3, 0]],
  [[0, 0, -7/3], [0, 0, 0], [7/3, 0, 0]],
  [[0, 7/3, 0], [-7/3, 0, 0], [0, 0, 0]]])
```

Check of the antisymmetrization:

```
sage: all(s[i,j,k] == (c[i,j,k]-c[i,k,j]+c[j,k,i]-c[j,i,k]+c[k,i,j]-c[k,j,i])/
↪6
....:     for i in range(3) for j in range(3) for k in range(3))
True
```

Antisymmetrization over already antisymmetric indices does not change anything:

```
sage: s1 = s.antisymmetrize(1,2) ; s1
Fully antisymmetric 3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s1 == s
True
sage: c1 = c.antisymmetrize(1,2) ; c1
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (1, 2)
```

```
sage: c1 == c
True
```

But in general, antisymmetrization may alter previous antisymmetries:

```
sage: c2 = c.antisymmetrize(0,1) ; c2  # the antisymmetry (2,3) is lost:
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 1)
sage: c2 == c
False
sage: c = s*a ; c
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 1, 2)
sage: s = c.antisymmetrize(1,3) ; s
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (1, 3),
   with antisymmetry on the index positions (0, 2)
sage: s._antisym  # the antisymmetry (0,1,2) has been reduced to (0,2), since␣
↪1 is involved in the new antisymmetry (1,3):
[(1, 3), (0, 2)]
```

Partial antisymmetrization of 4-indices components with a symmetry on the first two indices:

```
sage: a = CompFullySym(QQ, V.basis(), 2)
sage: a[:] = [[-2,1,3], [1,0,-5], [3,-5,4]]
sage: b = Components(QQ, V.basis(), 2)
sage: b[:] = [[1,2,3], [5,7,11], [13,17,19]]
sage: c = a*b ; c
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1)
sage: s = c.antisymmetrize(2,3) ; s
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1),
   with antisymmetry on the index positions (2, 3)
```

Some check of the antisymmetrization:

```
sage: all(s[2,2,i,j] == (c[2,2,i,j] - c[2,2,j,i])/2
....:     for i in range(3) for j in range(i,3))
True
```

The full antisymmetrization results in zero because of the symmetry on the first two indices:

```
sage: s = c.antisymmetrize() ; s
Fully antisymmetric 4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s == 0
True
```

Similarly, the partial antisymmetrization on the first two indices results in zero:

```
sage: s = c.antisymmetrize(0,1) ; s
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 1)
sage: s == 0
True
```

The partial antisymmetrization on the positions $(0, 2)$ destroys the symmetry on $(0, 1)$:

```
sage: s = c.antisymmetrize(0,2) ; s
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 2)
sage: s != 0
True
sage: s[0,1,2,1]
27/2
sage: s[1,0,2,1]  # the symmetry (0,1) is lost
-2
sage: s[2,1,0,1]  # the antisymmetry (0,2) holds
-27/2
```

non_redundant_index_generator ( )

Generator of indices, with only ordered indices in case of symmetries, so that only non-redundant indices are generated.

OUTPUT:

- an iterable index

EXAMPLES:

Indices on a 2-dimensional space:

```
sage: from sage.tensor.modules.comp import Components, CompWithSym, \
...    CompFullySym, CompFullyAntiSym
sage: V = VectorSpace(QQ, 2)
sage: c = CompFullySym(QQ, V.basis(), 2)
sage: list(c.non_redundant_index_generator())
[(0, 0), (0, 1), (1, 1)]
sage: c = CompFullySym(QQ, V.basis(), 2, start_index=1)
sage: list(c.non_redundant_index_generator())
[(1, 1), (1, 2), (2, 2)]
sage: c = CompFullyAntiSym(QQ, V.basis(), 2)
```

```
sage: list(c.non_redundant_index_generator())
[(0, 1)]
```

Indices on a 3-dimensional space:

```
sage: V = VectorSpace(QQ, 3)
sage: c = CompFullySym(QQ, V.basis(), 2)
sage: list(c.non_redundant_index_generator())
[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]
sage: c = CompFullySym(QQ, V.basis(), 2, start_index=1)
sage: list(c.non_redundant_index_generator())
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
sage: c = CompFullyAntiSym(QQ, V.basis(), 2)
sage: list(c.non_redundant_index_generator())
[(0, 1), (0, 2), (1, 2)]
sage: c = CompWithSym(QQ, V.basis(), 3, sym=(1,2))  # symmetry on the last␣
↪two indices
sage: list(c.non_redundant_index_generator())
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 1), (0, 1, 2),
 (0, 2, 2), (1, 0, 0), (1, 0, 1), (1, 0, 2), (1, 1, 1),
 (1, 1, 2), (1, 2, 2), (2, 0, 0), (2, 0, 1), (2, 0, 2),
 (2, 1, 1), (2, 1, 2), (2, 2, 2)]
sage: c = CompWithSym(QQ, V.basis(), 3, antisym=(1,2))  # antisymmetry on the␣
↪last two indices
sage: list(c.non_redundant_index_generator())
[(0, 0, 1), (0, 0, 2), (0, 1, 2), (1, 0, 1), (1, 0, 2), (1, 1, 2),
 (2, 0, 1), (2, 0, 2), (2, 1, 2)]
sage: c = CompFullySym(QQ, V.basis(), 3)
sage: list(c.non_redundant_index_generator())
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 1), (0, 1, 2), (0, 2, 2),
 (1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
sage: c = CompFullyAntiSym(QQ, V.basis(), 3)
sage: list(c.non_redundant_index_generator())
[(0, 1, 2)]
```

Indices on a 4-dimensional space:

```
sage: V = VectorSpace(QQ, 4)
sage: c = Components(QQ, V.basis(), 1)
sage: list(c.non_redundant_index_generator())
[(0,), (1,), (2,), (3,)]
sage: c = CompFullyAntiSym(QQ, V.basis(), 2)
sage: list(c.non_redundant_index_generator())
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: c = CompFullyAntiSym(QQ, V.basis(), 3)
sage: list(c.non_redundant_index_generator())
[(0, 1, 2), (0, 1, 3), (0, 2, 3), (1, 2, 3)]
sage: c = CompFullyAntiSym(QQ, V.basis(), 4)
sage: list(c.non_redundant_index_generator())
[(0, 1, 2, 3)]
sage: c = CompFullyAntiSym(QQ, V.basis(), 5)
sage: list(c.non_redundant_index_generator())  # nothing since c is␣
↪identically zero in this case (for 5 > 4)
[]
```

swap_adjacent_indices (*pos1*, *pos2*, *pos3*)

Swap two adjacent sets of indices.

This method is essentially required to reorder the covariant and contravariant indices in the computation of a tensor product.

The symmetries are preserved and the corresponding indices are adjusted consequently.

INPUT:

- `pos1` – position of the first index of set 1 (with the convention position=0 for the first slot)
- `pos2` – position of the first index of set 2 = 1 + position of the last index of set 1 (since the two sets are adjacent)
- `pos3` – 1 + position of the last index of set 2

OUTPUT:

- Components with index set 1 permuted with index set 2.

EXAMPLES:

Swap of the index in position 0 with the pair of indices in position (1,2) in a set of components antisymmetric with respect to the indices in position (1,2):

```
sage: from sage.tensor.modules.comp import CompWithSym
sage: V = VectorSpace(QQ, 3)
sage: c = CompWithSym(QQ, V.basis(), 3, antisym=(1,2))
sage: c[0,0,1], c[0,0,2], c[0,1,2] = (1,2,3)
sage: c[1,0,1], c[1,0,2], c[1,1,2] = (4,5,6)
sage: c[2,0,1], c[2,0,2], c[2,1,2] = (7,8,9)
sage: c[:]
[[[0, 1, 2], [-1, 0, 3], [-2, -3, 0]],
 [[0, 4, 5], [-4, 0, 6], [-5, -6, 0]],
 [[0, 7, 8], [-7, 0, 9], [-8, -9, 0]]]
sage: c1 = c.swap_adjacent_indices(0,1,3)
sage: c._antisym   # c is antisymmetric with respect to the last pair of␣
↪indices...
[(1, 2)]
sage: c1._antisym  #...while c1 is antisymmetric with respect to the first␣
↪pair of indices
[(0, 1)]
sage: c[0,1,2]
3
sage: c1[1,2,0]
3
sage: c1[2,1,0]
-3
```

symmetrize (**pos*)

Symmetrization over the given index positions.

INPUT:

- `pos` – list of index positions involved in the symmetrization (with the convention `position=0` for the first slot); if none, the symmetrization is performed over all the indices

OUTPUT:

- an instance of *CompWithSym* describing the symmetrized components

EXAMPLES:

Symmetrization of 3-indices components on a 3-dimensional space:

```
sage: from sage.tensor.modules.comp import Components, CompWithSym, \
....:   CompFullySym, CompFullyAntiSym
sage: V = VectorSpace(QQ, 3)
sage: c = Components(QQ, V.basis(), 3)
sage: c[:] = [[[1,2,3], [4,5,6], [7,8,9]], [[10,11,12], [13,14,15],␣
↪[16,17,18]], [[19,20,21], [22,23,24], [25,26,27]]]
sage: cs = c.symmetrize(0,1) ; cs
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1)
sage: s = cs.symmetrize() ; s
Fully symmetric 3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: cs[:], s[:]
([[[1, 2, 3], [7, 8, 9], [13, 14, 15]],
  [[7, 8, 9], [13, 14, 15], [19, 20, 21]],
  [[13, 14, 15], [19, 20, 21], [25, 26, 27]]],
 [[[1, 16/3, 29/3], [16/3, 29/3, 14], [29/3, 14, 55/3]],
  [[16/3, 29/3, 14], [29/3, 14, 55/3], [14, 55/3, 68/3]],
  [[29/3, 14, 55/3], [14, 55/3, 68/3], [55/3, 68/3, 27]]])
sage: s == c.symmetrize() # should be true
True
sage: s1 = cs.symmetrize(0,1) ; s1   # should return a copy of cs
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1)
sage: s1 == cs    # check that s1 is a copy of cs
True
```

Let us now start with a symmetry on the last two indices:

```
sage: cs1 = c.symmetrize(1,2) ; cs1
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (1, 2)
sage: s2 = cs1.symmetrize() ; s2
Fully symmetric 3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s2 == c.symmetrize()
True
```

Symmetrization alters pre-existing symmetries: let us symmetrize w.r.t. the index positions $(1, 2)$ a set of components that is symmetric w.r.t. the index positions $(0, 1)$:

```
sage: cs = c.symmetrize(0,1) ; cs
3-indices components w.r.t. [
```

```
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1)
sage: css = cs.symmetrize(1,2)
sage: css # the symmetry (0,1) has been lost:
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (1, 2)
sage: css[:]
[[[1, 9/2, 8], [9/2, 8, 23/2], [8, 23/2, 15]],
 [[7, 21/2, 14], [21/2, 14, 35/2], [14, 35/2, 21]],
 [[13, 33/2, 20], [33/2, 20, 47/2], [20, 47/2, 27]]]
sage: cs[:]
[[[1, 2, 3], [7, 8, 9], [13, 14, 15]],
 [[7, 8, 9], [13, 14, 15], [19, 20, 21]],
 [[13, 14, 15], [19, 20, 21], [25, 26, 27]]]
sage: css == c.symmetrize() # css differs from the full symmetrized version
False
sage: css.symmetrize() == c.symmetrize() # one has to symmetrize css over all␣
↪indices to recover it
True
```

Another example of symmetry alteration: symmetrization over $(0, 1)$ of a 4-indices set of components that is symmetric w.r.t. $(1, 2, 3)$:

```
sage: v = Components(QQ, V.basis(), 1)
sage: v[:] = (-2,1,4)
sage: a = v*s ; a
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (1, 2, 3)
sage: a1 = a.symmetrize(0,1) ; a1 # the symmetry (1,2,3) has been reduced to␣
↪(2,3):
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1), with symmetry on the index␣
↪positions (2, 3)
sage: a1._sym  # a1 has two distinct symmetries:
[(0, 1), (2, 3)]
sage: a[0,1,2,0] == a[0,0,2,1]  # a is symmetric w.r.t. positions 1 and 3
True
sage: a1[0,1,2,0] == a1[0,0,2,1] # a1 is not
False
sage: a1[0,1,2,0] == a1[1,0,2,0] # but it is symmetric w.r.t. position 0 and 1
True
sage: a[0,1,2,0] == a[1,0,2,0] # while a is not
False
```

Partial symmetrization of 4-indices components with an antisymmetry on the last two indices:

```
sage: a = Components(QQ, V.basis(), 2)
sage: a[:] = [[-1,2,3], [4,5,-6], [7,8,9]]
sage: b = CompFullyAntiSym(QQ, V.basis(), 2)
sage: b[0,1], b[0,2], b[1,2] = (2, 4, 8)
sage: c = a*b ; c
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (2, 3)
sage: s = c.symmetrize(0,1) ; s  # symmetrization on the first two indices
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1), with antisymmetry on the␣
↪index positions (2, 3)
sage: s[0,1,2,1] == (c[0,1,2,1] + c[1,0,2,1]) / 2 # check of the␣
↪symmetrization
True
sage: s = c.symmetrize() ; s  # symmetrization over all the indices
Fully symmetric 4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s == 0    # the full symmetrization results in zero due to the␣
↪antisymmetry on the last two indices
True
sage: s = c.symmetrize(2,3) ; s
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (2, 3)
sage: s == 0    # must be zero since the symmetrization has been performed on␣
↪the antisymmetric indices
True
sage: s = c.symmetrize(0,2) ; s
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 2)
sage: s != 0  # s is not zero, but the antisymmetry on (2,3) is lost because␣
↪the position 2 is involved in the new symmetry
True
```

Partial symmetrization of 4-indices components with an antisymmetry on the last three indices:

```
sage: a = Components(QQ, V.basis(), 1)
sage: a[:] = (1, -2, 3)
sage: b = CompFullyAntiSym(QQ, V.basis(), 3)
sage: b[0,1,2] = 4
sage: c = a*b ; c
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
```

```
(0, 0, 1)
], with antisymmetry on the index positions (1, 2, 3)
sage: s = c.symmetrize(0,1) ; s
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1),
   with antisymmetry on the index positions (2, 3)
```

Note that the antisymmetry on $(1, 2, 3)$ has been reduced to $(2, 3)$ only:

```
sage: s = c.symmetrize(1,2) ; s
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (1, 2)
sage: s == 0 # because (1,2) are involved in the original antisymmetry
True
```

trace (*pos1*, *pos2*)

Index contraction, taking care of the symmetries.

INPUT:

- `pos1` – position of the first index for the contraction (with the convention position=0 for the first slot)
- `pos2` – position of the second index for the contraction

OUTPUT:

- set of components resulting from the (pos1, pos2) contraction

EXAMPLES:

Self-contraction of symmetric 2-indices components:

```
sage: from sage.tensor.modules.comp import Components, CompWithSym, \
....:   CompFullySym, CompFullyAntiSym
sage: V = VectorSpace(QQ, 3)
sage: a = CompFullySym(QQ, V.basis(), 2)
sage: a[:] = [[1,2,3],[2,4,5],[3,5,6]]
sage: a.trace(0,1)
11
sage: a[0,0] + a[1,1] + a[2,2]
11
```

Self-contraction of antisymmetric 2-indices components:

```
sage: b = CompFullyAntiSym(QQ, V.basis(), 2)
sage: b[0,1], b[0,2], b[1,2] = (3, -2, 1)
sage: b.trace(0,1)  # must be zero by antisymmetry
0
```

Self-contraction of 3-indices components with one symmetry:

```
sage: v = Components(QQ, V.basis(), 1)
sage: v[:] = (-2, 4, -8)
```

```
sage: c = v*b ; c
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (1, 2)
sage: s = c.trace(0,1) ; s
1-index components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s[:]
[-28, 2, 8]
sage: [sum(v[k]*b[k,i] for k in range(3)) for i in range(3)] # check
[-28, 2, 8]
sage: s = c.trace(1,2) ; s
1-index components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s[:] # is zero by antisymmetry
[0, 0, 0]
sage: c = b*v ; c
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 1)
sage: s = c.trace(0,1)
sage: s[:]  # is zero by antisymmetry
[0, 0, 0]
sage: s = c.trace(1,2) ; s[:]
[28, -2, -8]
sage: [sum(b[i,k]*v[k] for k in range(3)) for i in range(3)]  # check
[28, -2, -8]
```

Self-contraction of 4-indices components with two symmetries:

```
sage: c = a*b ; c
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1), with antisymmetry on the␣
↪index positions (2, 3)
sage: s = c.trace(0,1) ; s  # the symmetry on (0,1) is lost:
Fully antisymmetric 2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s[:]
[  0  33 -22]
[-33   0  11]
[ 22 -11   0]
sage: [[sum(c[k,k,i,j] for k in range(3)) for j in range(3)] for i in␣
↪range(3)]  # check
```

```
[[0, 33, -22], [-33, 0, 11], [22, -11, 0]]
sage: s = c.trace(1,2) ; s  # both symmetries are lost by this contraction
2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s[:]
[ 0  0  0]
[-2  1  0]
[-3  3 -1]
sage: [[sum(c[i,k,k,j] for k in range(3)) for j in range(3)] for i in␣
↪range(3)]  # check
[[0, 0, 0], [-2, 1, 0], [-3, 3, -1]]
```

class sage.tensor.modules.comp. **Components** (*ring*, *frame*, *nb_indices*, *start_index=0*, *output_formatter=None*)

Bases: sage.structure.sage_object.SageObject

Indexed set of ring elements forming some components with respect to a given "frame".

The "frame" can be a basis of some vector space or a vector frame on some manifold (i.e. a field of bases). The stored quantities can be tensor components or non-tensorial quantities, such as connection coefficients or structure coefficients. The symmetries over some indices are dealt by subclasses of the class *Components*.

INPUT:

- `ring` – commutative ring in which each component takes its value
- `frame` – frame with respect to which the components are defined; whatever type `frame` is, it should have a method `__len__()` implemented, so that `len(frame)` returns the dimension, i.e. the size of a single index range
- `nb_indices` – number of integer indices labeling the components
- `start_index` – (default: 0) first value of a single index; accordingly a component index i must obey `start_index <= i <= start_index + dim -1`, where `dim = len(frame)`.
- `output_formatter` – (default: `None`) function or unbound method called to format the output of the component access operator `[...]` (method __getitem__); `output_formatter` must take 1 or 2 arguments: the 1st argument must be an element of `ring` and the second one, if any, some format specification.

EXAMPLES:

Set of components with 2 indices on a 3-dimensional vector space, the frame being some basis of the vector space:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ,3)
sage: basis = V.basis() ; basis
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: c = Components(QQ, basis, 2) ; c
2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
```

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

Actually, the frame can be any object that has some length, i.e. on which the function `len()` can be called:

```
sage: basis1 = V.gens() ; basis1
((1, 0, 0), (0, 1, 0), (0, 0, 1))
sage: c1 = Components(QQ, basis1, 2) ; c1
2-indices components w.r.t. ((1, 0, 0), (0, 1, 0), (0, 0, 1))
sage: basis2 = ['a', 'b' , 'c']
sage: c2 = Components(QQ, basis2, 2) ; c2
2-indices components w.r.t. ['a', 'b', 'c']
```

By default, the indices range from 0 to $n-1$, where n is the length of the frame. This can be changed via the argument `start_index` :

```
sage: c1 = Components(QQ, basis, 2, start_index=1)
sage: c1[0,1]
Traceback (most recent call last):
...
IndexError: index out of range: 0 not in [1, 3]
sage: c[0,1]  # for c, the index 0 is OK
0
sage: c[0,1] = -3
sage: c1[:] = c[:] # list copy of all components
sage: c1[1,2]  # (1,2) = (0,1) shifted by 1
-3
```

If some formatter function or unbound method is provided via the argument `output_formatter` , it is used to change the ouput of the access operator `[...]` :

```
sage: a = Components(QQ, basis, 2, output_formatter=Rational.numerical_approx)
sage: a[1,2] = 1/3
sage: a[1,2]
0.333333333333333
```

The format can be passed to the formatter as the last argument of the access operator `[...]` :

```
sage: a[1,2,10] # here the format is 10, for 10 bits of precision
0.33
sage: a[1,2,100]
0.33333333333333333333333333333
```

The raw (unformatted) components are then accessed by the double bracket operator:

```
sage: a[[1,2]]
1/3
```

For sets of components declared without any output formatter, there is no difference between `[...]` and `[[...]]` :

```
sage: c[1,2] = 1/3
sage: c[1,2], c[[1,2]]
(1/3, 1/3)
```

The formatter is also used for the complete list of components:

```
sage: a[:]
[0.000000000000000 0.000000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000 0.333333333333333]
[0.000000000000000 0.000000000000000 0.000000000000000]
sage: a[:,10] # with a format different from the default one (53 bits)
[0.00 0.00 0.00]
[0.00 0.00 0.33]
[0.00 0.00 0.00]
```

The complete list of components in raw form can be recovered by the double bracket operator, replacing `:` by `slice(None)` (since `a[[:]]` generates a Python syntax error):

```
sage: a[[slice(None)]]
[  0   0   0]
[  0   0 1/3]
[  0   0   0]
```

Another example of formatter: the Python built-in function `str()` to generate string outputs:

```
sage: b = Components(QQ, V.basis(), 1, output_formatter=str)
sage: b[:] = (1, 0, -4)
sage: b[:]
['1', '0', '-4']
```

For such a formatter, 2-indices components are no longer displayed as a matrix:

```
sage: b = Components(QQ, basis, 2, output_formatter=str)
sage: b[0,1] = 1/3
sage: b[:]
[['0', '1/3', '0'], ['0', '0', '0'], ['0', '0', '0']]
```

But unformatted outputs still are:

```
sage: b[[slice(None)]]
[  0 1/3   0]
[  0   0   0]
[  0   0   0]
```

Internally, the components are stored as a dictionary (`_comp`) whose keys are the indices; only the non-zero components are stored:

```
sage: a[:]
[0.000000000000000 0.000000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000 0.333333333333333]
[0.000000000000000 0.000000000000000 0.000000000000000]
sage: a._comp
{(1, 2): 1/3}
sage: v = Components(QQ, basis, 1)
sage: v[:] = (-1, 0, 3)
sage: v._comp  # random output order of the component dictionary
{(0,): -1, (2,): 3}
```

ARITHMETIC EXAMPLES:

Unary plus operator:

```
sage: a = Components(QQ, basis, 1)
sage: a[:] = (-1, 0, 3)
sage: s = +a ; s[:]
[-1, 0, 3]
sage: +a == a
True
```

Unary minus operator:

```
sage: s = -a ; s[:]
[1, 0, -3]
```

Addition:

```
sage: b = Components(QQ, basis, 1)
sage: b[:] = (2, 1, 4)
sage: s = a + b ; s[:]
[1, 1, 7]
sage: a + b == b + a
True
sage: a + (-a) == 0
True
```

Subtraction:

```
sage: s = a - b ; s[:]
[-3, -1, -1]
sage: s + b == a
True
sage: a - b == - (b - a)
True
```

Multiplication by a scalar:

```
sage: s = 2*a ; s[:]
[-2, 0, 6]
```

Division by a scalar:

```
sage: s = a/2 ; s[:]
[-1/2, 0, 3/2]
sage: 2*(a/2) == a
True
```

Tensor product (by means of the operator $*$):

```
sage: c = a*b ; c
2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: a[:], b[:]
([-1, 0, 3], [2, 1, 4])
sage: c[:]
[-2 -1 -4]
[ 0  0  0]
[ 6  3 12]
```

```
sage: d = c*a ; d
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: d[:]
[[[2, 0, -6], [1, 0, -3], [4, 0, -12]],
 [[0, 0, 0], [0, 0, 0], [0, 0, 0]],
 [[-6, 0, 18], [-3, 0, 9], [-12, 0, 36]]]
sage: d[0,1,2] == a[0]*b[1]*a[2]
True
```

antisymmetrize (**pos*)

Antisymmetrization over the given index positions

INPUT:

- pos – list of index positions involved in the antisymmetrization (with the convention position=0 for the first slot); if none, the antisymmetrization is performed over all the indices

OUTPUT:

- an instance of *CompWithSym* describing the antisymmetrized components.

EXAMPLES:

Antisymmetrization of 2-indices components:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ, 3)
sage: c = Components(QQ, V.basis(), 2)
sage: c[:] = [[1,2,3], [4,5,6], [7,8,9]]
sage: s = c.antisymmetrize() ; s
Fully antisymmetric 2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: c[:], s[:]
(
[1 2 3]  [ 0 -1 -2]
[4 5 6]  [ 1  0 -1]
[7 8 9], [ 2  1  0]
)
sage: c.antisymmetrize() == c.antisymmetrize(0,1)
True
```

Full antisymmetrization of 3-indices components:

```
sage: c = Components(QQ, V.basis(), 3)
sage: c[:] = [[[-1,-2,3], [4,-5,4], [-7,8,9]], [[10,10,12], [13,-14,15], [-
↪16,17,19]], [[-19,20,21], [1,2,3], [-25,26,27]]]
sage: s = c.antisymmetrize() ; s
Fully antisymmetric 3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: c[:], s[:]
```

```
([[[-1, -2, 3], [4, -5, 4], [-7, 8, 9]],
  [[10, 10, 12], [13, -14, 15], [-16, 17, 19]],
  [[-19, 20, 21], [1, 2, 3], [-25, 26, 27]]],
 [[[0, 0, 0], [0, 0, -13/6], [0, 13/6, 0]],
  [[0, 0, 13/6], [0, 0, 0], [-13/6, 0, 0]],
  [[0, -13/6, 0], [13/6, 0, 0], [0, 0, 0]]])
sage: all(s[i,j,k] == (c[i,j,k]-c[i,k,j]+c[j,k,i]-c[j,i,k]+c[k,i,j]-c[k,j,i])/
 ↪6  # Check of the result:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: c.symmetrize() == c.symmetrize(0,1,2)
True
```

Partial antisymmetrization of 3-indices components:

```
sage: s = c.antisymmetrize(0,1) ; s  # antisymmetrization on the first two␣
 ↪indices
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 1)
sage: c[:], s[:]
([[[-1, -2, 3], [4, -5, 4], [-7, 8, 9]],
  [[10, 10, 12], [13, -14, 15], [-16, 17, 19]],
  [[-19, 20, 21], [1, 2, 3], [-25, 26, 27]]],
 [[[0, 0, 0], [-3, -15/2, -4], [6, -6, -6]],
  [[3, 15/2, 4], [0, 0, 0], [-17/2, 15/2, 8]],
  [[-6, 6, 6], [17/2, -15/2, -8], [0, 0, 0]]])
sage: all(s[i,j,k] == (c[i,j,k]-c[j,i,k])/2  # Check of the result:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s = c.antisymmetrize(1,2) ; s  # antisymmetrization on the last two␣
 ↪indices
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (1, 2)
sage: c[:], s[:]
([[[-1, -2, 3], [4, -5, 4], [-7, 8, 9]],
  [[10, 10, 12], [13, -14, 15], [-16, 17, 19]],
  [[-19, 20, 21], [1, 2, 3], [-25, 26, 27]]],
 [[[0, -3, 5], [3, 0, -2], [-5, 2, 0]],
  [[0, -3/2, 14], [3/2, 0, -1], [-14, 1, 0]],
  [[0, 19/2, 23], [-19/2, 0, -23/2], [-23, 23/2, 0]]])
sage: all(s[i,j,k] == (c[i,j,k]-c[i,k,j])/2  # Check of the result:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s = c.antisymmetrize(0,2) ; s  # antisymmetrization on the first and␣
 ↪last indices
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with antisymmetry on the index positions (0, 2)
sage: c[:], s[:]
([[[-1, -2, 3], [4, -5, 4], [-7, 8, 9]],
```

```
  [[10, 10, 12], [13, -14, 15], [-16, 17, 19]],
  [[-19, 20, 21], [1, 2, 3], [-25, 26, 27]]],
 [[[0, -6, 11], [0, -9, 3/2], [0, 12, 17]],
  [[6, 0, -4], [9, 0, 13/2], [-12, 0, -7/2]],
  [[-11, 4, 0], [-3/2, -13/2, 0], [-17, 7/2, 0]]])
sage: all(s[i,j,k] == (c[i,j,k]-c[k,j,i])/2  # Check of the result:
....:     for i in range(3) for j in range(3) for k in range(3))
True
```

The order of index positions in the argument does not matter:

```
sage: c.antisymmetrize(1,0) == c.antisymmetrize(0,1)
True
sage: c.antisymmetrize(2,1) == c.antisymmetrize(1,2)
True
sage: c.antisymmetrize(2,0) == c.antisymmetrize(0,2)
True
```

contract (**args*)

Contraction on one or many indices with another instance of *Components* .

INPUT:

- `pos1` – positions of the indices in `self` involved in the contraction; `pos1` must be a sequence of integers, with 0 standing for the first index position, 1 for the second one, etc. If `pos1` is not provided, a single contraction on the last index position of `self` is assumed
- `other` – the set of components to contract with
- `pos2` – positions of the indices in `other` involved in the contraction, with the same conventions as for `pos1` . If `pos2` is not provided, a single contraction on the first index position of `other` is assumed

OUTPUT:

- set of components resulting from the contraction

EXAMPLES:

Contraction of a 1-index set of components with a 2-index one:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ, 3)
sage: a = Components(QQ, V.basis(), 1)
sage: a[:] = (-1, 2, 3)
sage: b = Components(QQ, V.basis(), 2)
sage: b[:] = [[1,2,3], [4,5,6], [7,8,9]]
sage: s0 = a.contract(0, b, 0) ; s0
1-index components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s0[:]
[28, 32, 36]
sage: s0[:] == [sum(a[j]*b[j,i] for j in range(3)) for i in range(3)]  # check
True
sage: s1 = a.contract(0, b, 1) ; s1[:]
[12, 24, 36]
```

```
sage: s1[:] == [sum(a[j]*b[i,j] for j in range(3)) for i in range(3)]  # check
True
```

Parallel computations (see `Parallelism`):

```
sage: Parallelism().set('tensor', nproc=2)
sage: Parallelism().get('tensor')
2
sage: s0_par = a.contract(0, b, 0) ; s0_par
1-index components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s0_par[:]
[28, 32, 36]
sage: s0_par == s0
True
sage: s1_par = a.contract(0, b, 1) ; s1_par[:]
[12, 24, 36]
sage: s1_par == s1
True
sage: Parallelism().set('tensor', nproc = 1)  # switch off parallelization
```

Contraction on 2 indices:

```
sage: c = a*b ; c
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s = c.contract(1,2, b, 0,1) ; s
1-index components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s[:]
[-285, 570, 855]
sage: [sum(sum(c[i,j,k]*b[j,k] for k in range(3)) # check
....:      for j in range(3)) for i in range(3)]
[-285, 570, 855]
```

Parallel computation:

```
sage: Parallelism().set('tensor', nproc=2)
sage: c_par = a*b ; c_par
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: c_par == c
True
sage: s_par = c_par.contract(1,2, b, 0,1) ; s_par
1-index components w.r.t. [
(1, 0, 0),
```

```
(0, 1, 0),
(0, 0, 1)
]
sage: s_par[:]
[-285, 570, 855]
sage: s_par == s
True
sage: Parallelism().set('tensor', nproc=1)  # switch off parallelization
```

Consistency check with *trace()* :

```
sage: b = a*a ; b   # the tensor product of a with itself
Fully symmetric 2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: b[:]
[ 1 -2 -3]
[-2  4  6]
[-3  6  9]
sage: b.trace(0,1)
14
sage: a.contract(0, a, 0) == b.trace(0,1)
True
```

copy ( )

Return an exact copy of `self` .

EXAMPLES:

Copy of a set of components with a single index:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ,3)
sage: a = Components(QQ, V.basis(), 1)
sage: a[:] = -2, 1, 5
sage: b = a.copy() ; b
1-index components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: b[:]
[-2, 1, 5]
sage: b == a
True
sage: b is a  # b is a distinct object
False
```

index_generator ( )

Generator of indices.

OUTPUT:

- an iterable index

EXAMPLES:

Indices on a 3-dimensional vector space:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ,3)
sage: c = Components(QQ, V.basis(), 1)
sage: list(c.index_generator())
[(0,), (1,), (2,)]
sage: c = Components(QQ, V.basis(), 1, start_index=1)
sage: list(c.index_generator())
[(1,), (2,), (3,)]
sage: c = Components(QQ, V.basis(), 2)
sage: list(c.index_generator())
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0),
 (2, 1), (2, 2)]
```

is_zero ( )

Return `True` if all the components are zero and `False` otherwise.

EXAMPLES:

A just-created set of components is initialized to zero:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ,3)
sage: c = Components(QQ, V.basis(), 1)
sage: c.is_zero()
True
sage: c[:]
[0, 0, 0]
sage: c[0] = 1 ; c[:]
[1, 0, 0]
sage: c.is_zero()
False
sage: c[0] = 0 ; c[:]
[0, 0, 0]
sage: c.is_zero()
True
```

It is equivalent to use the operator == to compare to zero:

```
sage: c == 0
True
sage: c != 0
False
```

Comparing to a nonzero number is meaningless:

```
sage: c == 1
Traceback (most recent call last):
...
TypeError: cannot compare a set of components to a number
```

non_redundant_index_generator ( )

Generator of non redundant indices.

In the absence of declared symmetries, all possible indices are generated. So this method is equivalent to *index_generator()* . Only versions for derived classes with symmetries or antisymmetries are not trivial.

OUTPUT:

- an iterable index

EXAMPLES:

Indices on a 3-dimensional vector space:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ,3)
sage: c = Components(QQ, V.basis(), 2)
sage: list(c.non_redundant_index_generator())
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0),
 (2, 1), (2, 2)]
sage: c = Components(QQ, V.basis(), 2, start_index=1)
sage: list(c.non_redundant_index_generator())
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1),
 (3, 2), (3, 3)]
```

swap_adjacent_indices (*pos1*, *pos2*, *pos3*)

Swap two adjacent sets of indices.

This method is essentially required to reorder the covariant and contravariant indices in the computation of a tensor product.

INPUT:

- `pos1` – position of the first index of set 1 (with the convention `position=0` for the first slot)
- `pos2` – position of the first index of set 2 equals 1 plus the position of the last index of set 1 (since the two sets are adjacent)
- `pos3` – 1 plus position of the last index of set 2

OUTPUT:

- Components with index set 1 permuted with index set 2.

EXAMPLES:

Swap of the two indices of a 2-indices set of components:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ, 3)
sage: c = Components(QQ, V.basis(), 2)
sage: c[:] = [[1,2,3], [4,5,6], [7,8,9]]
sage: c1 = c.swap_adjacent_indices(0,1,2)
sage: c[:], c1[:]
(
[1 2 3]  [1 4 7]
[4 5 6]  [2 5 8]
[7 8 9], [3 6 9]
)
```

Swap of two pairs of indices on a 4-indices set of components:

```
sage: d = c*c1 ; d
4-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: d1 = d.swap_adjacent_indices(0,2,4)
sage: d[0,1,1,2]
```

```
16
sage: d1[1,2,0,1]
16
sage: d1[0,1,1,2]
24
sage: d[1,2,0,1]
24
```

symmetrize (**pos*)

Symmetrization over the given index positions.

INPUT:

- `pos` – list of index positions involved in the symmetrization (with the convention position=0 for the first slot); if none, the symmetrization is performed over all the indices

OUTPUT:

- an instance of *CompWithSym* describing the symmetrized components

EXAMPLES:

Symmetrization of 2-indices components:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ, 3)
sage: c = Components(QQ, V.basis(), 2)
sage: c[:] = [[1,2,3], [4,5,6], [7,8,9]]
sage: s = c.symmetrize() ; s
Fully symmetric 2-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: c[:], s[:]
(
[1 2 3]  [1 3 5]
[4 5 6]  [3 5 7]
[7 8 9], [5 7 9]
)
sage: c.symmetrize() == c.symmetrize(0,1)
True
```

Full symmetrization of 3-indices components:

```
sage: c = Components(QQ, V.basis(), 3)
sage: c[:] = [[[1,2,3], [4,5,6], [7,8,9]], [[10,11,12], [13,14,15],␣
↪[16,17,18]], [[19,20,21], [22,23,24], [25,26,27]]]
sage: s = c.symmetrize() ; s
Fully symmetric 3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: c[:], s[:]
([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
  [[10, 11, 12], [13, 14, 15], [16, 17, 18]],
  [[19, 20, 21], [22, 23, 24], [25, 26, 27]]],
 [[[1, 16/3, 29/3], [16/3, 29/3, 14], [29/3, 14, 55/3]],
  [[16/3, 29/3, 14], [29/3, 14, 55/3], [14, 55/3, 68/3]],
```

```
  [[29/3, 14, 55/3], [14, 55/3, 68/3], [55/3, 68/3, 27]]])
sage: all(s[i,j,k] == (c[i,j,k]+c[i,k,j]+c[j,k,i]+c[j,i,k]+c[k,i,j]+c[k,j,i])/
↪6  # Check of the result:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: c.symmetrize() == c.symmetrize(0,1,2)
True
```

Partial symmetrization of 3-indices components:

```
sage: s = c.symmetrize(0,1) ; s   # symmetrization on the first two indices
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 1)
sage: c[:], s[:]
([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
  [[10, 11, 12], [13, 14, 15], [16, 17, 18]],
  [[19, 20, 21], [22, 23, 24], [25, 26, 27]]],
 [[[1, 2, 3], [7, 8, 9], [13, 14, 15]],
  [[7, 8, 9], [13, 14, 15], [19, 20, 21]],
  [[13, 14, 15], [19, 20, 21], [25, 26, 27]]])
sage: all(s[i,j,k] == (c[i,j,k]+c[j,i,k])/2   # Check of the result:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s = c.symmetrize(1,2) ; s   # symmetrization on the last two indices
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (1, 2)
sage: c[:], s[:]
([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
  [[10, 11, 12], [13, 14, 15], [16, 17, 18]],
  [[19, 20, 21], [22, 23, 24], [25, 26, 27]]],
 [[[1, 3, 5], [3, 5, 7], [5, 7, 9]],
  [[10, 12, 14], [12, 14, 16], [14, 16, 18]],
  [[19, 21, 23], [21, 23, 25], [23, 25, 27]]])
sage: all(s[i,j,k] == (c[i,j,k]+c[i,k,j])/2   # Check of the result:
....:     for i in range(3) for j in range(3) for k in range(3))
True
sage: s = c.symmetrize(0,2) ; s   # symmetrization on the first and last␣
↪indices
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
], with symmetry on the index positions (0, 2)
sage: c[:], s[:]
([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
  [[10, 11, 12], [13, 14, 15], [16, 17, 18]],
  [[19, 20, 21], [22, 23, 24], [25, 26, 27]]],
 [[[1, 6, 11], [4, 9, 14], [7, 12, 17]],
  [[6, 11, 16], [9, 14, 19], [12, 17, 22]],
  [[11, 16, 21], [14, 19, 24], [17, 22, 27]]])
sage: all(s[i,j,k] == (c[i,j,k]+c[k,j,i])/2   # Check of the result:
....:     for i in range(3) for j in range(3) for k in range(3))
```

```
True
```

trace (*pos1*, *pos2*)

Index contraction.

INPUT:

- `pos1` – position of the first index for the contraction (with the convention position=0 for the first slot)
- `pos2` – position of the second index for the contraction

OUTPUT:

- set of components resulting from the (pos1, pos2) contraction

EXAMPLES:

Self-contraction of a set of components with 2 indices:

```
sage: from sage.tensor.modules.comp import Components
sage: V = VectorSpace(QQ, 3)
sage: c = Components(QQ, V.basis(), 2)
sage: c[:] = [[1,2,3], [4,5,6], [7,8,9]]
sage: c.trace(0,1)
15
sage: c[0,0] + c[1,1] + c[2,2]  # check
15
```

Three self-contractions of a set of components with 3 indices:

```
sage: v = Components(QQ, V.basis(), 1)
sage: v[:] = (-1,2,3)
sage: a = c*v ; a
3-indices components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s = a.trace(0,1) ; s  # contraction on the first two indices
1-index components w.r.t. [
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: s[:]
[-15, 30, 45]
sage: [sum(a[j,j,i] for j in range(3)) for i in range(3)]  # check
[-15, 30, 45]
sage: s = a.trace(0,2) ; s[:]  # contraction on the first and last indices
[28, 32, 36]
sage: [sum(a[j,i,j] for j in range(3)) for i in range(3)]  # check
[28, 32, 36]
sage: s = a.trace(1,2) ; s[:] # contraction on the last two indices
[12, 24, 36]
sage: [sum(a[i,j,j] for j in range(3)) for i in range(3)]  # check
[12, 24, 36]
```

class sage.tensor.modules.comp. **KroneckerDelta** (*ring*, *frame*, *start_index=0*, *output_formatter=None*)

Bases: *sage.tensor.modules.comp.CompFullySym*

Kronecker delta δ_{ij}.

INPUT:

- `ring` – commutative ring in which each component takes its value
- `frame` – frame with respect to which the components are defined; whatever type `frame` is, it should have some method `__len__()` implemented, so that `len(frame)` returns the dimension, i.e. the size of a single index range
- `start_index` – (default: 0) first value of a single index; accordingly a component index i must obey `start_index <= i <= start_index + dim -1`, where `dim = len(frame)`.
- `output_formatter` – (default: `None`) function or unbound method called to format the output of the component access operator `[...]` (method `__getitem__`); `output_formatter` must take 1 or 2 arguments: the first argument must be an instance of `ring` and the second one, if any, some format specification

EXAMPLES:

The Kronecker delta on a 3-dimensional space:

```
sage: from sage.tensor.modules.comp import KroneckerDelta
sage: V = VectorSpace(QQ,3)
sage: d = KroneckerDelta(QQ, V.basis()) ; d
Kronecker delta of size 3x3
sage: d[:]
[1 0 0]
[0 1 0]
[0 0 1]
```

One can read, but not set, the components of a Kronecker delta:

```
sage: d[1,1]
1
sage: d[1,1] = 2
Traceback (most recent call last):
...
TypeError: the components of a Kronecker delta cannot be changed
```

Examples of use with output formatters:

```
sage: d = KroneckerDelta(QQ, V.basis(), output_formatter=Rational.numerical_
↪approx)
sage: d[:]  # default format (53 bits of precision)
[ 1.00000000000000 0.000000000000000 0.000000000000000]
[0.000000000000000  1.00000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000  1.00000000000000]
sage: d[:,10] # format = 10 bits of precision
[ 1.0 0.00 0.00]
[0.00  1.0 0.00]
[0.00 0.00  1.0]
sage: d = KroneckerDelta(QQ, V.basis(), output_formatter=str)
sage: d[:]
[['1', '0', '0'], ['0', '1', '0'], ['0', '0', '1']]
```

CHAPTER

SEVEN

FORMATTING UTILITIES

This module defines helper functions that are not class methods.

AUTHORS:

- Eric Gourgoulhon, Michal Bejger (2014-2015): initial version
- Joris Vankerschaver (2010): for the function *is_atomic()*

class sage.tensor.modules.format_utilities. **FormattedExpansion** (*txt=None*, *latex=None*)

Bases: sage.structure.sage_object.SageObject

Helper class for displaying tensor expansions.

EXAMPLES:

```
sage: from sage.tensor.modules.format_utilities import FormattedExpansion
sage: f = FormattedExpansion('v', r'\tilde v')
sage: f
v
sage: latex(f)
\tilde v
sage: f = FormattedExpansion('x/2', r'\frac{x}{2}')
sage: f
x/2
sage: latex(f)
\frac{x}{2}
```

sage.tensor.modules.format_utilities. **format_mul_latex** (*name1*, *operator*, *name2*)

Helper function for LaTeX names of results of multiplication or tensor product.

EXAMPLES:

```
sage: from sage.tensor.modules.format_utilities import format_mul_latex
sage: format_mul_latex('a', '*', 'b')
'a*b'
sage: format_mul_latex('a+b', '*', 'c')
'\\left(a+b\\right)*c'
sage: format_mul_latex('a', '*', 'b+c')
'a*\\left(b+c\\right)'
sage: format_mul_latex('a+b', '*', 'c+d')
'\\left(a+b\\right)*\\left(c+d\\right)'
sage: format_mul_latex(None, '*', 'b')
sage: format_mul_latex('a', '*', None)
```

sage.tensor.modules.format_utilities. **format_mul_txt** (*name1*, *operator*, *name2*)

Helper function for text-formatted names of results of multiplication or tensor product.

EXAMPLES:

```
sage: from sage.tensor.modules.format_utilities import format_mul_txt
sage: format_mul_txt('a', '*', 'b')
'a*b'
sage: format_mul_txt('a+b', '*', 'c')
'(a+b)*c'
sage: format_mul_txt('a', '*', 'b+c')
'a*(b+c)'
sage: format_mul_txt('a+b', '*', 'c+d')
'(a+b)*(c+d)'
sage: format_mul_txt(None, '*', 'b')
sage: format_mul_txt('a', '*', None)
```

sage.tensor.modules.format_utilities. **format_unop_latex** (*operator*, *name*)

Helper function for LaTeX names of results of unary operator.

EXAMPLES:

```
sage: from sage.tensor.modules.format_utilities import format_unop_latex
sage: format_unop_latex('-', 'a')
'-a'
sage: format_unop_latex('-', 'a+b')
'-\\left(a+b\\right)'
sage: format_unop_latex('-', '(a+b)')
'-(a+b)'
sage: format_unop_latex('-', None)
```

sage.tensor.modules.format_utilities. **format_unop_txt** (*operator*, *name*)

Helper function for text-formatted names of results of unary operator.

EXAMPLES:

```
sage: from sage.tensor.modules.format_utilities import format_unop_txt
sage: format_unop_txt('-', 'a')
'-a'
sage: format_unop_txt('-', 'a+b')
'-(a+b)'
sage: format_unop_txt('-', '(a+b)')
'-(a+b)'
sage: format_unop_txt('-', None)
```

sage.tensor.modules.format_utilities. **is_atomic** (*expression*)

Helper function to check whether some LaTeX expression is atomic.

Adapted from method `_is_atomic()` of class `DifferentialFormFormatter` written by Joris Vankerschaver (2010).

INPUT:

- `expression` – string representing the expression (e.g. LaTeX string)

OUTPUT:

- `True` if additive operations are enclosed in parentheses and `False` otherwise.

EXAMPLES:

```
sage: from sage.tensor.modules.format_utilities import is_atomic
sage: is_atomic("2*x")
True
```

```
sage: is_atomic("2+x")
False
sage: is_atomic("(2+x)")
True
```

sage.tensor.modules.format_utilities. **is_atomic_wedge_latex** (*expression*)

Helper function to check whether LaTeX-formatted expression is atomic in terms of wedge products.

Adapted from method `_is_atomic()` of class `DifferentialFormFormatter` written by Joris Vankerschaver (2010).

INPUT:

- `expression` – string representing the LaTeX expression

OUTPUT:

- `True` if wedge products are enclosed in parentheses and `False` otherwise.

EXAMPLES:

```
sage: from sage.tensor.modules.format_utilities import is_atomic_wedge_latex
sage: is_atomic_wedge_latex(r"a")
True
sage: is_atomic_wedge_latex(r"a\wedge b")
False
sage: is_atomic_wedge_latex(r"(a\wedge b)")
True
sage: is_atomic_wedge_latex(r"(a\wedge b)\wedge c")
False
sage: is_atomic_wedge_latex(r"((a\wedge b)\wedge c)")
True
sage: is_atomic_wedge_latex(r"(a\wedge b\wedge c)")
True
sage: is_atomic_wedge_latex(r"\omega\wedge\theta")
False
sage: is_atomic_wedge_latex(r"(\omega\wedge\theta)")
True
sage: is_atomic_wedge_latex(r"\omega\wedge(\theta+a)")
False
```

sage.tensor.modules.format_utilities. **is_atomic_wedge_txt** (*expression*)

Helper function to check whether some text-formatted expression is atomic in terms of wedge products.

Adapted from method `_is_atomic()` of class `DifferentialFormFormatter` written by Joris Vankerschaver (2010).

INPUT:

- `expression` – string representing the text-formatted expression

OUTPUT:

- `True` if wedge products are enclosed in parentheses and `False` otherwise.

EXAMPLES:

```
sage: from sage.tensor.modules.format_utilities import is_atomic_wedge_txt
sage: is_atomic_wedge_txt("a")
True
sage: is_atomic_wedge_txt(r"a/\b")
False
```

```
sage: is_atomic_wedge_txt(r"(a/\b)")
True
sage: is_atomic_wedge_txt(r"(a/\b)/\c")
False
sage: is_atomic_wedge_txt(r"(a/\b/\c)")
True
```

CHAPTER

EIGHT

INDICES AND TABLES

- Index
- Module Index
- Search Page

t

sage.tensor.modules.comp, 121
sage.tensor.modules.ext_pow_free_module, 77
sage.tensor.modules.finite_rank_free_module, 3
sage.tensor.modules.format_utilities, 159
sage.tensor.modules.free_module_alt_form, 81
sage.tensor.modules.free_module_automorphism, 106
sage.tensor.modules.free_module_basis, 35
sage.tensor.modules.free_module_homset, 91
sage.tensor.modules.free_module_linear_group, 101
sage.tensor.modules.free_module_morphism, 95
sage.tensor.modules.free_module_tensor, 45
sage.tensor.modules.tensor_free_module, 39
sage.tensor.modules.tensor_with_indices, 74

A

add_comp() (sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism method), 110
add_comp() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 51
alternating_form() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 14
antisymmetrize() (sage.tensor.modules.comp.Components method), 147
antisymmetrize() (sage.tensor.modules.comp.CompWithSym method), 132
antisymmetrize() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 52
automorphism() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 15

B

base_module() (sage.tensor.modules.ext_pow_free_module.ExtPowerFreeModule method), 80
base_module() (sage.tensor.modules.free_module_linear_group.FreeModuleLinearGroup method), 105
base_module() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 55
base_module() (sage.tensor.modules.tensor_free_module.TensorFreeModule method), 44
bases() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 16
basis() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 17
Basis_abstract (class in sage.tensor.modules.free_module_basis), 35

C

change_of_basis() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 18
common_basis() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 55
comp() (sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism method), 111
comp() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 56
CompFullyAntiSym (class in sage.tensor.modules.comp), 124
CompFullySym (class in sage.tensor.modules.comp), 127
Components (class in sage.tensor.modules.comp), 143
components() (sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism method), 112
components() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 57
CompWithSym (class in sage.tensor.modules.comp), 129
contract() (sage.tensor.modules.comp.Components method), 149
contract() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 58
copy() (sage.tensor.modules.comp.Components method), 151
copy() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 61

D

default_basis() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 20
degree() (sage.tensor.modules.ext_pow_free_module.ExtPowerFreeModule method), 80

degree() (sage.tensor.modules.free_module_alt_form.FreeModuleAltForm method), 84
del_other_comp() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 62
det() (sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism method), 114
disp() (sage.tensor.modules.free_module_alt_form.FreeModuleAltForm method), 84
disp() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 62
display() (sage.tensor.modules.free_module_alt_form.FreeModuleAltForm method), 86
display() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 64
dual() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 20
dual_basis() (sage.tensor.modules.free_module_basis.FreeModuleBasis method), 36
dual_exterior_power() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 21

E

Element (sage.tensor.modules.ext_pow_free_module.ExtPowerFreeModule attribute), 80
Element (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule attribute), 14
Element (sage.tensor.modules.free_module_homset.FreeModuleHomset attribute), 94
Element (sage.tensor.modules.free_module_linear_group.FreeModuleLinearGroup attribute), 105
Element (sage.tensor.modules.tensor_free_module.TensorFreeModule attribute), 44
endomorphism() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 22
ExtPowerFreeModule (class in sage.tensor.modules.ext_pow_free_module), 77

F

FiniteRankFreeModule (class in sage.tensor.modules.finite_rank_free_module), 11
FiniteRankFreeModuleElement (class in sage.tensor.modules.free_module_tensor), 47
FiniteRankFreeModuleMorphism (class in sage.tensor.modules.free_module_morphism), 95
format_mul_latex() (in module sage.tensor.modules.format_utilities), 159
format_mul_txt() (in module sage.tensor.modules.format_utilities), 159
format_unop_latex() (in module sage.tensor.modules.format_utilities), 160
format_unop_txt() (in module sage.tensor.modules.format_utilities), 160
FormattedExpansion (class in sage.tensor.modules.format_utilities), 159
free_module() (sage.tensor.modules.free_module_basis.Basis_abstract method), 35
FreeModuleAltForm (class in sage.tensor.modules.free_module_alt_form), 81
FreeModuleAutomorphism (class in sage.tensor.modules.free_module_automorphism), 106
FreeModuleBasis (class in sage.tensor.modules.free_module_basis), 35
FreeModuleCoBasis (class in sage.tensor.modules.free_module_basis), 37
FreeModuleHomset (class in sage.tensor.modules.free_module_homset), 91
FreeModuleLinearGroup (class in sage.tensor.modules.free_module_linear_group), 101
FreeModuleTensor (class in sage.tensor.modules.free_module_tensor), 50

G

general_linear_group() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 23

H

hom() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 24

I

identity_map() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 25
index_generator() (sage.tensor.modules.comp.Components method), 151
inverse() (sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism method), 114
irange() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 26
is_atomic() (in module sage.tensor.modules.format_utilities), 160

is_atomic_wedge_latex() (in module sage.tensor.modules.format_utilities), 161
is_atomic_wedge_txt() (in module sage.tensor.modules.format_utilities), 161
is_identity() (sage.tensor.modules.free_module_morphism.FiniteRankFreeModuleMorphism method), 97
is_injective() (sage.tensor.modules.free_module_morphism.FiniteRankFreeModuleMorphism method), 98
is_surjective() (sage.tensor.modules.free_module_morphism.FiniteRankFreeModuleMorphism method), 99
is_zero() (sage.tensor.modules.comp.Components method), 152

K

KroneckerDelta (class in sage.tensor.modules.comp), 156

L

linear_form() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 26

M

matrix() (sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism method), 115
matrix() (sage.tensor.modules.free_module_morphism.FiniteRankFreeModuleMorphism method), 99

N

new_basis() (sage.tensor.modules.free_module_basis.FreeModuleBasis method), 37
non_redundant_index_generator() (sage.tensor.modules.comp.Components method), 152
non_redundant_index_generator() (sage.tensor.modules.comp.CompWithSym method), 135

O

one() (sage.tensor.modules.free_module_homset.FreeModuleHomset method), 94
one() (sage.tensor.modules.free_module_linear_group.FreeModuleLinearGroup method), 105

P

pick_a_basis() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 65
print_bases() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 27

R

rank() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 28

S

sage.tensor.modules.comp (module), 121
sage.tensor.modules.ext_pow_free_module (module), 77
sage.tensor.modules.finite_rank_free_module (module), 3
sage.tensor.modules.format_utilities (module), 159
sage.tensor.modules.free_module_alt_form (module), 81
sage.tensor.modules.free_module_automorphism (module), 106
sage.tensor.modules.free_module_basis (module), 35
sage.tensor.modules.free_module_homset (module), 91
sage.tensor.modules.free_module_linear_group (module), 101
sage.tensor.modules.free_module_morphism (module), 95
sage.tensor.modules.free_module_tensor (module), 45
sage.tensor.modules.tensor_free_module (module), 39
sage.tensor.modules.tensor_with_indices (module), 74
set_change_of_basis() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 28
set_comp() (sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism method), 117

set_comp() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 66
set_default_basis() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 29
set_name() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 67
swap_adjacent_indices() (sage.tensor.modules.comp.Components method), 153
swap_adjacent_indices() (sage.tensor.modules.comp.CompWithSym method), 136
sym_bilinear_form() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 29
symmetries() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 67
symmetrize() (sage.tensor.modules.comp.Components method), 154
symmetrize() (sage.tensor.modules.comp.CompWithSym method), 137
symmetrize() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 68

T

tensor() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 31
tensor_from_comp() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 31
tensor_module() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 32
tensor_rank() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 71
tensor_type() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 71
tensor_type() (sage.tensor.modules.tensor_free_module.TensorFreeModule method), 45
TensorFreeModule (class in sage.tensor.modules.tensor_free_module), 39
TensorWithIndices (class in sage.tensor.modules.tensor_with_indices), 74
trace() (sage.tensor.modules.comp.Components method), 156
trace() (sage.tensor.modules.comp.CompWithSym method), 141
trace() (sage.tensor.modules.free_module_automorphism.FreeModuleAutomorphism method), 119
trace() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 71

U

update() (sage.tensor.modules.tensor_with_indices.TensorWithIndices method), 76

V

view() (sage.tensor.modules.free_module_tensor.FreeModuleTensor method), 73

W

wedge() (sage.tensor.modules.free_module_alt_form.FreeModuleAltForm method), 87

Z

zero() (sage.tensor.modules.finite_rank_free_module.FiniteRankFreeModule method), 33