万兴PDF

(continued from previous page)

```
PowerBasis(x**4 + x**3 + x**2 + x + 1)
>>> B = A.submodule_from_matrix(2 * DomainMatrix.eye(4, ZZ), denom=3)
>>> print(B)
Submodule[[2, 0, 0, 0], [0, 2, 0, 0], [0, 0, 2, 0], [0, 0, 0, 2]]/3
>>> print(B.parent)
PowerBasis(x**4 + x**3 + x**2 + x + 1)
```

Thus, every module is either a *PowerBasis* (page 2725), or a *Submodule* (page 2726), some ancestor of which is a *PowerBasis* (page 2725). (If `S` is a *Submodule* (page 2726), then its ancestors are `S.parent`, `S.parent.parent`, and so on).

The *ModuleElement* (page 2729) class represents a linear combination of the generators of any module. Critically, the coefficients of this linear combination are not restricted to be integers, but may be any rational numbers. This is necessary so that any and all algebraic integers be representable, starting from the power basis in a primitive element θ for the number field in question. For example, in a quadratic field $\mathbb{Q}(\sqrt{d})$ where $d \equiv 1 \mod 4$, a denominator of 2 is needed.

A *ModuleElement* (page 2729) can be constructed from an integer column vector and a denominator:

```
>>> U = Poly(x**2 - 5)
>>> M = PowerBasis(U)
>>> e = M(DM([[1], [1]], ZZ), denom=2)
>>> print(e)
[1, 1]/2
>>> print(e.module)
PowerBasis(x**2 - 5)
```

The *PowerBasisElement* (page 2732) class is a subclass of *ModuleElement* (page 2729) that represents elements of a *PowerBasis* (page 2725), and adds functionality pertinent to elements represented directly over powers of the primitive element θ.

Arithmetic with module elements

While a *ModuleElement* (page 2729) represents a linear combination over the generators of a particular module, recall that every module is either a *PowerBasis* (page 2725) or a descendant (along a chain of *Submodule* (page 2726) objects) thereof, so that in fact every *ModuleElement* (page 2729) represents an algebraic number in some field $\mathbb{Q}(\theta)$, where θ is the defining element of some *PowerBasis* (page 2725). It thus makes sense to talk about the number field to which a given *ModuleElement* (page 2729) belongs.

This means that any two *ModuleElement* (page 2729) instances can be added, subtracted, multiplied, or divided, provided they belong to the same number field. Similarly, since $\mathbb{Q}$ is a subfield of every number field, any *ModuleElement* (page 2729) may be added, multiplied, etc. by any rational number.

```
>>> from sympy import QQ
>>> from sympy.polys.numberfields.modules import to_col
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> C = A.submodule_from_matrix(3 * DomainMatrix.eye(4, ZZ))
```

(continues on next page)

(continued from previous page)

```
>>> e = A(to_col([0, 2, 0, 0]), denom=3)
>>> f = A(to_col([0, 0, 0, 7]), denom=5)
>>> g = C(to_col([1, 1, 1, 1]))
>>> e + f
[0, 10, 0, 21]/15
>>> e - f
[0, 10, 0, -21]/15
>>> e - g
[-9, -7, -9, -9]/3
>>> e + QQ(7, 10)
[21, 20, 0, 0]/30
>>> e * f
[-14, -14, -14, -14]/15
>>> e ** 2
[0, 0, 4, 0]/9
>>> f // g
[7, 7, 7, 7]/15
>>> f * QQ(2, 3)
[0, 0, 0, 14]/15
```

However, care must be taken with arithmetic operations on *ModuleElement* (page 2729), because the module C to which the result will belong will be the nearest common ancestor (NCA) of the modules A, B to which the two operands belong, and C may be different from either or both of A and B.

```
>>> A = PowerBasis(T)
>>> B = A.submodule_from_matrix(2 * DomainMatrix.eye(4, ZZ))
>>> C = A.submodule_from_matrix(3 * DomainMatrix.eye(4, ZZ))
>>> print((B(0) * C(0)).module == A)
True
```

Before the arithmetic operation is performed, copies of the two operands are automatically converted into elements of the NCA (the operands themselves are not modified). This upward conversion along an ancestor chain is easy: it just requires the successive multiplication by the defining matrix of each *Submodule* (page 2726).

Conversely, downward conversion, i.e. representing a given *ModuleElement* (page 2729) in a submodule, is also supported – namely by the *represent()* (page 2729) method – but is not guaranteed to succeed in general, since the given element may not belong to the submodule. The main circumstance in which this issue tends to arise is with multiplication, since modules, while closed under addition, need not be closed under multiplication.

Multiplication

Generally speaking, a module need not be closed under multiplication, i.e. need not form a ring. However, many of the modules we work with in the context of number fields are in fact rings, and our classes do support multiplication.

Specifically, any *Module* (page 2719) can attempt to compute its own multiplication table, but this does not happen unless an attempt is made to multiply two *ModuleElement* (page 2729) instances belonging to it.

万兴PDF


```
>>> A = PowerBasis(T)
>>> print(A._mult_tab is None)
True
>>> a = A(0)*A(1)
>>> print(A._mult_tab is None)
False
```

Every *PowerBasis* (page 2725) is, by its nature, closed under multiplication, so instances of *PowerBasis* (page 2725) can always successfully compute their multiplication table.

When a *Submodule* (page 2726) attempts to compute its multiplication table, it converts each of its own generators into elements of its parent module, multiplies them there, in every possible pairing, and then tries to represent the results in itself, i.e. as $\mathbb{Z}$-linear combinations over its own generators. This will succeed if and only if the submodule is in fact closed under multiplication.

Module Homomorphisms

Many important number theoretic algorithms require the calculation of the kernel of one or more module homomorphisms. Accordingly we have several lightweight classes, *ModuleHomomorphism* (page 2733), *ModuleEndomorphism* (page 2734), *InnerEndomorphism* (page 2734), and *EndomorphismRing* (page 2734), which provide the minimal necessary machinery to support this.

Class Reference

class sympy.polys.numberfields.modules.**Module**

Generic finitely-generated module.

This is an abstract base class, and should not be instantiated directly. The two concrete subclasses are *PowerBasis* (page 2725) and *Submodule* (page 2726).

Every *Submodule* (page 2726) is derived from another module, referenced by its `parent` attribute. If `S` is a submodule, then we refer to `S.parent`, `S.parent.parent`, and so on, as the "ancestors" of `S`. Thus, every *Module* (page 2719) is either a *PowerBasis* (page 2725) or a *Submodule* (page 2726), some ancestor of which is a *PowerBasis* (page 2725).

__call__(*spec*, *denom=1*)

Generate a *ModuleElement* (page 2729) belonging to this module.

Parameters

spec : *DomainMatrix* (page 2671), int

Specifies the numerators of the coefficients of the *ModuleElement* (page 2729). Can be either a column vector over *ZZ* (page 2525), whose length must equal the number n of generators of this module, or else an integer `j`, $0 \leq j < n$, which is a shorthand for column j of I_n, the $n \times n$ identity matrix.

denom : int, optional (default=1)

Denominator for the coefficients of the *ModuleElement* (page 2729).

Returns

ModuleElement (page 2729)

The coefficients are the entries of the *spec* vector, divided by *denom*.

Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis, to_col
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> e = A(to_col([1, 2, 3, 4]), denom=3)
>>> print(e)
[1, 2, 3, 4]/3
>>> f = A(2)
>>> print(f)
[0, 0, 1, 0]
```

ancestors(*include_self=False*)

Return the list of ancestor modules of this module, from the foundational *PowerBasis* (page 2725) downward, optionally including `self`.

See also:

Module (page 2719)

basis_elements()

Get list of *ModuleElement* (page 2729) being the generators of this module.

element_from_rational(*a*)

Return a *ModuleElement* (page 2729) representing a rational number.

Parameters
a : int, *ZZ* (page 2525), *QQ* (page 2529)

Returns
ModuleElement (page 2729)

Explanation

The returned *ModuleElement* (page 2729) will belong to the first module on this module's ancestor chain (including this module itself) that starts with unity.

Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly, QQ
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> a = A.element_from_rational(QQ(2, 3))
>>> print(a)
[2, 0, 0, 0]/3
```

endomorphism_ring()

Form the *EndomorphismRing* (page 2734) for this module.

is_compat_col(*col*)

Say whether *col* is a suitable column vector for this module.

mult_tab()

Get the multiplication table for this module (if closed under mult).

Returns

dict of dict of lists

Raises

ClosureFailure

If the module is not closed under multiplication.

Explanation

Computes a dictionary `M` of dictionaries of lists, representing the upper triangular half of the multiplication table.

In other words, if `0 <= i <= j < self.n`, then `M[i][j]` is the list `c` of coefficients such that `g[i] * g[j] == sum(c[k]*g[k], k in range(self.n))`, where `g` is the list of generators of this module.

If `j < i` then `M[i][j]` is undefined.

Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> print(A.mult_tab())
{0: {0: [1, 0, 0, 0], 1: [0, 1, 0, 0], 2: [0, 0, 1, 0],     3: [0, 0,␣
↪0, 1]},
                  1: {1: [0, 0, 1, 0], 2: [0, 0, 0, 1],     3: [-1, -
↪1, -1, -1]},
                                       2: {2: [-1, -1, -1, -1], 3: [1, 0,␣
↪0, 0]},
                                                              3: {3: [0, 1,␣
↪0, 0]}}
```

property n

The number of generators of this module.

nearest_common_ancestor(*other*)

Locate the nearest common ancestor of this module and another.

Returns

Module (page 2719), None

See also:

Module (page 2719)

property number_field

Return the associated *AlgebraicField* (page 2539), if any.

Returns

AlgebraicField (page 2539), None

Explanation

A *PowerBasis* (page 2725) can be constructed on a *Poly* (page 2378) f or on an *AlgebraicField* (page 2539) K. In the latter case, the *PowerBasis* (page 2725) and all its descendant modules will return K as their .number_field property, while in the former case they will all return None.

one()

Return a *ModuleElement* (page 2729) representing unity, and belonging to the first ancestor of this module (including itself) that starts with unity.

property parent

The parent module, if any, for this module.

Returns

Module (page 2719), None

Explanation

For a *Submodule* (page 2726) this is its parent attribute; for a *PowerBasis* (page 2725) this is None.

See also:

Module (page 2719)

power_basis_ancestor()

Return the *PowerBasis* (page 2725) that is an ancestor of this module.

See also:

Module (page 2719)

represent(*elt*)

Represent a module element as an integer-linear combination over the generators of this module.

Parameters

elt : *ModuleElement* (page 2729)

The module element to be represented. Must belong to some ancestor module of this module (including this module itself).

Returns

DomainMatrix (page 2671) over *ZZ* (page 2525)

This will be a column vector, representing the coefficients of a linear combination of this module's generators, which equals the given element.

Raises

ClosureFailure

If the given element cannot be represented as a *ZZ* (page 2525)-linear combination over this module.

Explanation

In our system, to "represent" always means to write a *ModuleElement* (page 2729) as a *ZZ* (page 2525)-linear combination over the generators of the present *Module* (page 2719). Furthermore, the incoming *ModuleElement* (page 2729) must belong to an ancestor of the present *Module* (page 2719) (or to the present *Module* (page 2719) itself).

The most common application is to represent a *ModuleElement* (page 2729) in a *Submodule* (page 2726). For example, this is involved in computing multiplication tables.

On the other hand, representing in a *PowerBasis* (page 2725) is an odd case, and one which tends not to arise in practice, except for example when using a *ModuleEndomorphism* (page 2734) on a *PowerBasis* (page 2725).

In such a case, (1) the incoming *ModuleElement* (page 2729) must belong to the *PowerBasis* (page 2725) itself (since the latter has no proper ancestors) and (2) it is "representable" iff it belongs to $\mathbb{Z}[\theta]$ (although generally a *PowerBasisElement* (page 2732) may represent any element of $\mathbb{Q}(\theta)$, i.e. any algebraic number).

Examples

```
>>> from sympy import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis, to_col
>>> from sympy.abc import zeta
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> a = A(to_col([2, 4, 6, 8]))
```

The *ModuleElement* (page 2729) `a` has all even coefficients. If we represent `a` in the submodule `B = 2*A`, the coefficients in the column vector will be halved:

```
>>> B = A.submodule_from_gens([2*A(i) for i in range(4)])
>>> b = B.represent(a)
>>> print(b.transpose())
DomainMatrix([[1, 2, 3, 4]], (1, 4), ZZ)
```

However, the element of `B` so defined still represents the same algebraic number:

```
>>> print(a.poly(zeta).as_expr())
8*zeta**3 + 6*zeta**2 + 4*zeta + 2
>>> print(B(b).over_power_basis().poly(zeta).as_expr())
8*zeta**3 + 6*zeta**2 + 4*zeta + 2
```

See also:

Submodule.represent (page 2729), *PowerBasis.represent* (page 2725)

starts_with_unity()

Say whether the module's first generator equals unity.

submodule_from_gens(*gens*, *hnf=True*, *hnf_modulus=None*)

Form the submodule generated by a list of *ModuleElement* (page 2729) belonging to this module.

Parameters

gens : list of *ModuleElement* (page 2729) belonging to this module.

hnf : boolean, optional (default=True)

If True, we will reduce the matrix into Hermite Normal Form before forming the *Submodule* (page 2726).

hnf_modulus : int, None, optional (default=None)

Modulus for use in the HNF reduction algorithm. See *hermite_normal_form()* (page 2700).

Returns

Submodule (page 2726)

Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> gens = [A(0), 2*A(1), 3*A(2), 4*A(3)//5]
>>> B = A.submodule_from_gens(gens)
>>> print(B)
Submodule[[5, 0, 0, 0], [0, 10, 0, 0], [0, 0, 15, 0], [0, 0, 0, 4]]/5
```

See also:

submodule_from_matrix (page 2724)

submodule_from_matrix(*B*, *denom=1*)

Form the submodule generated by the elements of this module indicated by the columns of a matrix, with an optional denominator.

Parameters

B : *DomainMatrix* (page 2671) over *ZZ* (page 2525)

Each column gives the numerators of the coefficients of one generator of the submodule. Thus, the number of rows of *B* must equal the number of generators of the present module.

denom : int, optional (default=1)

Common denominator for all generators of the submodule.

Returns

Submodule (page 2726)

Raises

ValueError

If the given matrix *B* is not over *ZZ* (page 2525) or its number of rows does not equal the number of generators of the present module.

Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly, ZZ
>>> from sympy.polys.matrices import DM
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> B = A.submodule_from_matrix(DM([
...     [0, 10, 0, 0],
...     [0,  0, 7, 0],
... ], ZZ).transpose(), denom=15)
>>> print(B)
Submodule[[0, 10, 0, 0], [0, 0, 7, 0]]/15
```

See also:

submodule_from_gens (page 2723)

whole_submodule()

Return a submodule equal to this entire module.

Explanation

This is useful when you have a *PowerBasis* (page 2725) and want to turn it into a *Submodule* (page 2726) (in order to use methods belonging to the latter).

zero()

Return a *ModuleElement* (page 2729) representing zero.

class sympy.polys.numberfields.modules.**PowerBasis**(*T*)

The module generated by the powers of an algebraic integer.

__init__(*T*)

Parameters

T : *Poly* (page 2378), *AlgebraicField* (page 2539)

Either (1) the monic, irreducible, univariate polynomial over *ZZ* (page 2525), a root of which is the generator of the power basis, or (2) an *AlgebraicField* (page 2539) whose primitive element is the generator of the power basis.

element_from_ANP(*a*)

Convert an ANP into a PowerBasisElement.

element_from_alg_num(*a*)

Convert an AlgebraicNumber into a PowerBasisElement.

element_from_poly(*f*)

Produce an element of this module, representing *f* after reduction mod our defining minimal polynomial.

Parameters

f : *Poly* (page 2378) over *ZZ* (page 2525) in same var as our defining poly.

Returns

PowerBasisElement (page 2732)

represent(*elt*)

Represent a module element as an integer-linear combination over the generators of this module.

See also:

Module.represent (page 2722), *Submodule.represent* (page 2729)

class sympy.polys.numberfields.modules.**Submodule**(*parent*, *matrix*, *denom=1*, *mult_tab=None*)

A submodule of another module.

__init__(*parent*, *matrix*, *denom=1*, *mult_tab=None*)

Parameters

parent : *Module* (page 2719)

The module from which this one is derived.

matrix : *DomainMatrix* (page 2671) over *ZZ* (page 2525)

The matrix whose columns define this submodule's generators as linear combinations over the parent's generators.

denom : int, optional (default=1)

Denominator for the coefficients given by the matrix.

mult_tab : dict, None, optional

If already known, the multiplication table for this module may be supplied.

property QQ_matrix

DomainMatrix (page 2671) over *QQ* (page 2529), equal to `self.matrix / self.denom`, and guaranteed to be dense.

Returns

DomainMatrix (page 2671) over *QQ* (page 2529)

Explanation

Depending on how it is formed, a *DomainMatrix* (page 2671) may have an internal representation that is sparse or dense. We guarantee a dense representation here, so that tests for equivalence of submodules always come out as expected.

Examples

```
>>> from sympy.polys import Poly, cyclotomic_poly, ZZ
>>> from sympy.abc import x
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5, x))
>>> A = PowerBasis(T)
>>> B = A.submodule_from_matrix(3*DomainMatrix.eye(4, ZZ), denom=6)
>>> C = A.submodule_from_matrix(DomainMatrix.eye(4, ZZ), denom=2)
```

(continues on next page)

(continued from previous page)

```
>>> print(B.QQ_matrix == C.QQ_matrix)
True
```

add(*other*, *hnf=True*, *hnf_modulus=None*)

Add this *Submodule* (page 2726) to another.

Parameters

other : *Submodule* (page 2726)

hnf : boolean, optional (default=True)

If `True`, reduce the matrix of the combined module to its Hermite Normal Form.

hnf_modulus : *ZZ* (page 2525), None, optional

If a positive integer is provided, use this as modulus in the HNF reduction. See *hermite_normal_form()* (page 2700).

Returns

Submodule (page 2726)

Explanation

This represents the module generated by the union of the two modules' sets of generators.

basis_element_pullbacks()

Return list of this submodule's basis elements as elements of the submodule's parent module.

discard_before(*r*)

Produce a new module by discarding all generators before a given index *r*.

mul(*other*, *hnf=True*, *hnf_modulus=None*)

Multiply this *Submodule* (page 2726) by a rational number, a *ModuleElement* (page 2729), or another *Submodule* (page 2726).

Parameters

other : int, *ZZ* (page 2525), *QQ* (page 2529), *ModuleElement* (page 2729), *Submodule* (page 2726)

hnf : boolean, optional (default=True)

If `True`, reduce the matrix of the product module to its Hermite Normal Form.

hnf_modulus : *ZZ* (page 2525), None, optional

If a positive integer is provided, use this as modulus in the HNF reduction. See *hermite_normal_form()* (page 2700).

Returns

Submodule (page 2726)

Explanation

To multiply by a rational number or *ModuleElement* (page 2729) means to form the submodule whose generators are the products of this quantity with all the generators of the present submodule.

To multiply by another *Submodule* (page 2726) means to form the submodule whose generators are all the products of one generator from the one submodule, and one generator from the other.

reduce_element(*elt*)

If this submodule B has defining matrix W in square, maximal-rank Hermite normal form, then, given an element x of the parent module A, we produce an element $y \in A$ such that $x - y \in B$, and the ith coordinate of y satisfies $0 \leq y_i < w_{i,i}$. This representative y is unique, in the sense that every element of the coset $x + B$ reduces to it under this procedure.

Parameters

elt : *ModuleElement* (page 2729)

An element of this submodule's parent module.

Returns

elt : *ModuleElement* (page 2729)

An element of this submodule's parent module.

Raises

NotImplementedError

If the given *ModuleElement* (page 2729) does not belong to this submodule's parent module.

StructureError

If this submodule's defining matrix is not in square, maximal-rank Hermite normal form.

Explanation

In the special case where A is a power basis for a number field K, and B is a submodule representing an ideal I, this operation represents one of a few important ways of reducing an element of K modulo I to obtain a "small" representative. See [Cohen00] Section 1.4.3.

Examples

```
>>> from sympy import QQ, Poly, symbols
>>> t = symbols('t')
>>> k = QQ.alg_field_from_poly(Poly(t**3 + t**2 - 2*t + 8))
>>> Zk = k.maximal_order()
>>> A = Zk.parent
>>> B = (A(2) - 3*A(0))*Zk
>>> B.reduce_element(A(2))
[3, 0, 0]
```

References

[Cohen00]

reduced()

Produce a reduced version of this submodule.

Returns

Submodule (page 2726)

Explanation

In the reduced version, it is guaranteed that 1 is the only positive integer dividing both the submodule's denominator, and every entry in the submodule's matrix.

represent(*elt*)

Represent a module element as an integer-linear combination over the generators of this module.

See also:

Module.represent (page 2722), *PowerBasis.represent* (page 2725)

class sympy.polys.numberfields.modules.**ModuleElement**(*module*, *col*, *denom=1*)

Represents an element of a *Module* (page 2719).

NOTE: Should not be constructed directly. Use the *__call__()* (page 2719) method or the *make_mod_elt()* (page 2733) factory function instead.

__init__(*module*, *col*, *denom=1*)

Parameters

module : *Module* (page 2719)

The module to which this element belongs.

col : *DomainMatrix* (page 2671) over *ZZ* (page 2525)

Column vector giving the numerators of the coefficients of this element.

denom : int, optional (default=1)

Denominator for the coefficients of this element.

__add__(*other*)

A *ModuleElement* (page 2729) can be added to a rational number, or to another *ModuleElement* (page 2729).

Explanation

When the other summand is a rational number, it will be converted into a *ModuleElement* (page 2729) (belonging to the first ancestor of this module that starts with unity).

In all cases, the sum belongs to the nearest common ancestor (NCA) of the modules of the two summands. If the NCA does not exist, we return `NotImplemented`.

__mul__(*other*)

A *ModuleElement* (page 2729) can be multiplied by a rational number, or by another *ModuleElement* (page 2729).

Explanation

When the multiplier is a rational number, the product is computed by operating directly on the coefficients of this *ModuleElement* (page 2729).

When the multiplier is another *ModuleElement* (page 2729), the product will belong to the nearest common ancestor (NCA) of the modules of the two operands, and that NCA must have a multiplication table. If the NCA does not exist, we return `NotImplemented`. If the NCA does not have a mult. table, `ClosureFailure` will be raised.

__mod__(*m*)

Reduce this *ModuleElement* (page 2729) mod a *Submodule* (page 2726).

Parameters

m : int, *ZZ* (page 2525), *QQ* (page 2529), *Submodule* (page 2726)

If a *Submodule* (page 2726), reduce `self` relative to this. If an integer or rational, reduce relative to the *Submodule* (page 2726) that is our own module times this constant.

See also:

Submodule.reduce_element (page 2728)

property QQ_col

DomainMatrix (page 2671) over *QQ* (page 2529), equal to `self.col / self.denom`, and guaranteed to be dense.

See also:

Submodule.QQ_matrix (page 2726)

column(*domain=None*)

Get a copy of this element's column, optionally converting to a domain.

equiv(*other*)

A *ModuleElement* (page 2729) may test as equivalent to a rational number or another *ModuleElement* (page 2729), if they represent the same algebraic number.

Parameters

other : int, *ZZ* (page 2525), *QQ* (page 2529), *ModuleElement* (page 2729)

Returns

bool

Raises
UnificationFailed

If `self` and `other` do not share a common *PowerBasis* (page 2725) ancestor.

Explanation

This method is intended to check equivalence only in those cases in which it is easy to test; namely, when *other* is either a *ModuleElement* (page 2729) that can be unified with this one (i.e. one which shares a common *PowerBasis* (page 2725) ancestor), or else a rational number (which is easy because every *PowerBasis* (page 2725) represents every rational number).

classmethod from_int_list(*module*, *coeffs*, *denom=1*)

Make a *ModuleElement* (page 2729) from a list of ints (instead of a column vector).

is_compat(*other*)

Test whether other is another *ModuleElement* (page 2729) with same module.

property n

The length of this element's column.

over_power_basis()

Transform into a *PowerBasisElement* (page 2732) over our *PowerBasis* (page 2725) ancestor.

reduced()

Produce a reduced version of this ModuleElement, i.e. one in which the gcd of the denominator together with all numerator coefficients is 1.

reduced_mod_p(*p*)

Produce a version of this *ModuleElement* (page 2729) in which all numerator coefficients have been reduced mod *p*.

to_ancestor(*anc*)

Transform into a *ModuleElement* (page 2729) belonging to a given ancestor of this element's module.

Parameters
anc : *Module* (page 2719)

to_parent()

Transform into a *ModuleElement* (page 2729) belonging to the parent of this element's module.

unify(*other*)

Try to make a compatible pair of *ModuleElement* (page 2729), one equivalent to this one, and one equivalent to the other.

Returns
Pair `(e1, e2)`

Each `ei` is a *ModuleElement* (page 2729), they belong to the same *Module* (page 2719), `e1` is equivalent to `self`, and `e2` is equivalent to `other`.

Raises
UnificationFailed

If `self` and `other` have no common ancestor module.

Explanation

We search for the nearest common ancestor module for the pair of elements, and represent each one there.

class sympy.polys.numberfields.modules.**PowerBasisElement**(*module*, *col*, *denom=1*)

Subclass for *ModuleElement* (page 2729) instances whose module is a *PowerBasis* (page 2725).

property T

Access the defining polynomial of the *PowerBasis* (page 2725).

as_expr(*x=None*)

Create a Basic expression from `self`.

property generator

Return a *Symbol* (page 976) to be used when expressing this element as a polynomial.

If we have an associated *AlgebraicField* (page 2539) whose primitive element has an alias symbol, we use that. Otherwise we use the variable of the minimal polynomial defining the power basis to which we belong.

property is_rational

Say whether this element represents a rational number.

norm(*T=None*)

Compute the norm of this number.

numerator(*x=None*)

Obtain the numerator as a polynomial over *ZZ* (page 2525).

poly(*x=None*)

Obtain the number as a polynomial over *QQ* (page 2529).

to_ANP()

Convert to an equivalent *ANP* (page 2568).

to_alg_num()

Try to convert to an equivalent *AlgebraicNumber* (page 988).

Returns
AlgebraicNumber (page 988)

Raises
StructureError

If the *PowerBasis* (page 2725) to which this element belongs does not have an associated *AlgebraicField* (page 2539).

Explanation

In general, the conversion from an *AlgebraicNumber* (page 988) to a *PowerBasisElement* (page 2732) throws away information, because an *AlgebraicNumber* (page 988) specifies a complex embedding, while a *PowerBasisElement* (page 2732) does not. However, in some cases it is possible to convert a *PowerBasisElement* (page 2732) back into an *AlgebraicNumber* (page 988), namely when the associated *PowerBasis* (page 2725) has a reference to an *AlgebraicField* (page 2539).

sympy.polys.numberfields.modules.**make_mod_elt**(*module*, *col*, *denom=1*)

Factory function which builds a *ModuleElement* (page 2729), but ensures that it is a *PowerBasisElement* (page 2732) if the module is a *PowerBasis* (page 2725).

class sympy.polys.numberfields.modules.**ModuleHomomorphism**(*domain*, *codomain*, *mapping*)

A homomorphism from one module to another.

__init__(*domain*, *codomain*, *mapping*)

Parameters

domain : *Module* (page 2719)

The domain of the mapping.

codomain : *Module* (page 2719)

The codomain of the mapping.

mapping : callable

An arbitrary callable is accepted, but should be chosen so as to represent an actual module homomorphism. In particular, should accept elements of *domain* and return elements of *codomain*.

Examples

```
>>> from sympy import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis,␣
↪ModuleHomomorphism
>>> T = Poly(cyclotomic_poly(5))
>>> A = PowerBasis(T)
>>> B = A.submodule_from_gens([2*A(j) for j in range(4)])
>>> phi = ModuleHomomorphism(A, B, lambda x: 6*x)
>>> print(phi.matrix())
DomainMatrix([[3, 0, 0, 0], [0, 3, 0, 0], [0, 0, 3, 0], [0, 0, 0, 3]],
↪ (4, 4), ZZ)
```

kernel(*modulus=None*)

Compute a Submodule representing the kernel of this homomorphism.

Parameters

modulus : int, optional

A positive prime number p if the kernel should be computed mod p.

Returns

Submodule (page 2726)

This submodule's generators span the kernel of this homomorphism over *ZZ* (page 2525), or else over *GF(p)* (page 2522) if a modulus was given.

matrix(*modulus=None*)

Compute the matrix of this homomorphism.

Parameters
modulus : int, optional

A positive prime number p if the matrix should be reduced mod p.

Returns
DomainMatrix (page 2671)

The matrix is over *ZZ* (page 2525), or else over *GF(p)* (page 2522) if a modulus was given.

class sympy.polys.numberfields.modules.**ModuleEndomorphism**(*domain*, *mapping*)

A homomorphism from one module to itself.

__init__(*domain*, *mapping*)

Parameters
domain : *Module* (page 2719)

The common domain and codomain of the mapping.

mapping : callable

An arbitrary callable is accepted, but should be chosen so as to represent an actual module endomorphism. In particular, should accept and return elements of *domain*.

class sympy.polys.numberfields.modules.**InnerEndomorphism**(*domain*, *multiplier*)

An inner endomorphism on a module, i.e. the endomorphism corresponding to multiplication by a fixed element.

__init__(*domain*, *multiplier*)

Parameters
domain : *Module* (page 2719)

The domain and codomain of the endomorphism.

multiplier : *ModuleElement* (page 2729)

The element a defining the mapping as $x \mapsto ax$.

class sympy.polys.numberfields.modules.**EndomorphismRing**(*domain*)

The ring of endomorphisms on a module.

__init__(*domain*)

Parameters
domain : *Module* (page 2719)

The domain and codomain of the endomorphisms.

inner_endomorphism(*multiplier*)

Form an inner endomorphism belonging to this endomorphism ring.

Parameters
multiplier : *ModuleElement* (page 2729)

万兴PDF

Element a defining the inner endomorphism $x \mapsto ax$.

Returns

InnerEndomorphism (page 2734)

represent(*element*)

Represent an element of this endomorphism ring, as a single column vector.

Parameters

element : *ModuleEndomorphism* (page 2734) belonging to this ring.

Returns

DomainMatrix (page 2671)

Column vector equalling the vertical stacking of all the columns of the matrix that represents the given *element* as a mapping.

Explanation

Let M be a module, and E its ring of endomorphisms. Let N be another module, and consider a homomorphism $\varphi : N \to E$. In the event that φ is to be represented by a matrix A, each column of A must represent an element of E. This is possible when the elements of E are themselves representable as matrices, by stacking the columns of such a matrix into a single column.

This method supports calculating such matrices A, by representing an element of this endomorphism ring first as a matrix, and then stacking that matrix's columns into a single column.

Examples

Note that in these examples we print matrix transposes, to make their columns easier to inspect.

```
>>> from sympy import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> from sympy.polys.numberfields.modules import ModuleHomomorphism
>>> T = Poly(cyclotomic_poly(5))
>>> M = PowerBasis(T)
>>> E = M.endomorphism_ring()
```

Let ζ be a primitive 5th root of unity, a generator of our field, and consider the inner endomorphism τ on the ring of integers, induced by ζ:

```
>>> zeta = M(1)
>>> tau = E.inner_endomorphism(zeta)
>>> tau.matrix().transpose()
DomainMatrix(
    [[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1], [-1, -1, -1, -1]],
    (4, 4), ZZ)
```

The matrix representation of τ is as expected. The first column shows that multiplying by ζ carries 1 to ζ, the second column that it carries ζ to ζ^2, and so forth.

The `represent` method of the endomorphism ring `E` stacks these into a single column:

```
>>> E.represent(tau).transpose()
DomainMatrix(
    [[0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, -1, -1, -1, -1]],
    (1, 16), ZZ)
```

This is useful when we want to consider a homomorphism φ having `E` as codomain:

```
>>> phi = ModuleHomomorphism(M, E, lambda x: E.inner_endomorphism(x))
```

and we want to compute the matrix of such a homomorphism:

```
>>> phi.matrix().transpose()
DomainMatrix(
    [[1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
    [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, -1, -1, -1, -1],
    [0, 0, 1, 0, 0, 0, 0, 1, -1, -1, -1, -1, 1, 0, 0, 0],
    [0, 0, 0, 1, -1, -1, -1, -1, 1, 0, 0, 0, 0, 1, 0, 0]],
    (4, 16), ZZ)
```

Note that the stacked matrix of τ occurs as the second column in this example. This is because ζ is the second basis element of `M`, and $\varphi(\zeta) = \tau$.

`sympy.polys.numberfields.modules.`**`find_min_poly`**(*alpha*, *domain*, *x=None*, *powers=None*)

Find a polynomial of least degree (not necessarily irreducible) satisfied by an element of a finitely-generated ring with unity.

Parameters

alpha : *ModuleElement* (page 2729)

The element whose min poly is to be found, and whose module has multiplication and starts with unity.

domain : *Domain* (page 2504)

The desired domain of the polynomial.

x : *Symbol* (page 976), optional

The desired variable for the polynomial.

powers : list, optional

If desired, pass an empty list. The powers of *alpha* (as *ModuleElement* (page 2729) instances) from the zeroth up to the degree of the min poly will be recorded here, as we compute them.

Returns

Poly (page 2378), `None`

The minimal polynomial for alpha, or `None` if no polynomial could be found over the desired domain.

Raises

MissingUnityError

If the module to which alpha belongs does not start with unity.

ClosureFailure

If the module to which alpha belongs is not closed under multiplication.

Examples

For the nth cyclotomic field, n an odd prime, consider the quadratic equation whose roots are the two periods of length $(n-1)/2$. Article 356 of Gauss tells us that we should get $x^2+x-(n-1)/4$ or $x^2+x+(n+1)/4$ according to whether n is 1 or 3 mod 4, respectively.

```
>>> from sympy import Poly, cyclotomic_poly, primitive_root, QQ
>>> from sympy.abc import x
>>> from sympy.polys.numberfields.modules import PowerBasis, find_min_
↪poly
>>> n = 13
>>> g = primitive_root(n)
>>> C = PowerBasis(Poly(cyclotomic_poly(n, x)))
>>> ee = [g**(2*k+1) % n for k in range((n-1)//2)]
>>> eta = sum(C(e) for e in ee)
>>> print(find_min_poly(eta, QQ, x=x).as_expr())
x**2 + x - 3
>>> n = 19
>>> g = primitive_root(n)
>>> C = PowerBasis(Poly(cyclotomic_poly(n, x)))
>>> ee = [g**(2*k+2) % n for k in range((n-1)//2)]
>>> eta = sum(C(e) for e in ee)
>>> print(find_min_poly(eta, QQ, x=x).as_expr())
x**2 + x + 5
```

Utilities

sympy.polys.numberfields.utilities.**is_rat**(*c*)

Test whether an argument is of an acceptable type to be used as a rational number.

Explanation

Returns `True` on any argument of type `int`, *ZZ* (page 2525), or *QQ* (page 2529).

See also:

is_int (page 2737)

sympy.polys.numberfields.utilities.**is_int**(*c*)

Test whether an argument is of an acceptable type to be used as an integer.

Explanation

Returns `True` on any argument of type `int` or *ZZ* (page 2525).

See also:

is_rat (page 2737)

`sympy.polys.numberfields.utilities.`**`get_num_denom`**`(`c`)`

Given any argument on which *is_rat()* (page 2737) is `True`, return the numerator and denominator of this number.

See also:

is_rat (page 2737)

`sympy.polys.numberfields.utilities.`**`extract_fundamental_discriminant`**`(`a`)`

Extract a fundamental discriminant from an integer a.

Parameters
a: int, must be 0 or 1 mod 4

Returns
Pair `(D, F)` of dictionaries.

Raises
ValueError

If a is not 0 or 1 mod 4.

Explanation

Given any rational integer a that is 0 or 1 mod 4, write $a = df^2$, where d is either 1 or a fundamental discriminant, and return a pair of dictionaries `(D, F)` giving the prime factorizations of d and f respectively, in the same format returned by *factorint()* (page 1493).

A fundamental discriminant d is different from unity, and is either 1 mod 4 and squarefree, or is 0 mod 4 and such that $d/4$ is squarefree and 2 or 3 mod 4. This is the same as being the discriminant of some quadratic field.

Examples

```
>>> from sympy.polys.numberfields.utilities import extract_fundamental_
↪discriminant
>>> print(extract_fundamental_discriminant(-432))
({3: 1, -1: 1}, {2: 2, 3: 1})
```

For comparison:

```
>>> from sympy import factorint
>>> print(factorint(-432))
{2: 4, 3: 3, -1: 1}
```

References

[R716]

class sympy.polys.numberfields.utilities.**AlgIntPowers**(*T*, *modulus=None*)

Compute the powers of an algebraic integer.

Explanation

Given an algebraic integer θ by its monic irreducible polynomial T over *ZZ* (page 2525), this class computes representations of arbitrarily high powers of θ, as *ZZ* (page 2525)-linear combinations over $\{1, \theta, \ldots, \theta^{n-1}\}$, where $n = \deg(T)$.

The representations are computed using the linear recurrence relations for powers of θ, derived from the polynomial T. See [1], Sec. 4.2.2.

Optionally, the representations may be reduced with respect to a modulus.

Examples

```
>>> from sympy import Poly, cyclotomic_poly
>>> from sympy.polys.numberfields.utilities import AlgIntPowers
>>> T = Poly(cyclotomic_poly(5))
>>> zeta_pow = AlgIntPowers(T)
>>> print(zeta_pow[0])
[1, 0, 0, 0]
>>> print(zeta_pow[1])
[0, 1, 0, 0]
>>> print(zeta_pow[4])
[-1, -1, -1, -1]
>>> print(zeta_pow[24])
[-1, -1, -1, -1]
```

References

[R717]

__init__(*T*, *modulus=None*)

Parameters

T : *Poly* (page 2378)

The monic irreducible polynomial over *ZZ* (page 2525) defining the algebraic integer.

modulus : int, None, optional

If not None, all representations will be reduced w.r.t. this.

sympy.polys.numberfields.utilities.**coeff_search**(*m*, *R*)

Generate coefficients for searching through polynomials.

Parameters

m : int

Length of coeff list.

R : int

Initial max abs val for coeffs (will increase as search proceeds).

Returns

generator

Infinite generator of lists of coefficients.

Explanation

Lead coeff is always non-negative. Explore all combinations with coeffs bounded in absolute value before increasing the bound. Skip the all-zero list, and skip any repeats. See examples.

Examples

```
>>> from sympy.polys.numberfields.utilities import coeff_search
>>> cs = coeff_search(2, 1)
>>> C = [next(cs) for i in range(13)]
>>> print(C)
[[1, 1], [1, 0], [1, -1], [0, 1], [2, 2], [2, 1], [2, 0], [2, -1], [2, -
↪2],
 [1, 2], [1, -2], [0, 2], [3, 3]]
```

sympy.polys.numberfields.utilities.**supplement_a_subspace**(M)

Extend a basis for a subspace to a basis for the whole space.

Parameters

M : *DomainMatrix* (page 2671)

The columns give the basis for the subspace.

Returns

DomainMatrix (page 2671)

This matrix is invertible and its first r columns equal M.

Raises

DMRankError

If M was not of maximal rank.

Explanation

Given an $n \times r$ matrix M of rank r (so $r \leq n$), this function computes an invertible $n \times n$ matrix B such that the first r columns of B equal M.

This operation can be interpreted as a way of extending a basis for a subspace, to give a basis for the whole space.

To be precise, suppose you have an n-dimensional vector space V, with basis $\{v_1, v_2, \ldots, v_n\}$, and an r-dimensional subspace W of V, spanned by a basis $\{w_1, w_2, \ldots, w_r\}$, where the w_j are given as linear combinations of the v_i. If the columns of M represent

万兴PDF

the w_j as such linear combinations, then the columns of the matrix B computed by this function give a new basis $\{u_1, u_2, \ldots, u_n\}$ for V, again relative to the $\{v_i\}$ basis, and such that $u_j = w_j$ for $1 \leq j \leq r$.

Examples

Note: The function works in terms of columns, so in these examples we print matrix transposes in order to make the columns easier to inspect.

```
>>> from sympy.polys.matrices import DM
>>> from sympy import QQ, FF
>>> from sympy.polys.numberfields.utilities import supplement_a_subspace
>>> M = DM([[1, 7, 0], [2, 3, 4]], QQ).transpose()
>>> print(supplement_a_subspace(M).to_Matrix().transpose())
Matrix([[1, 7, 0], [2, 3, 4], [1, 0, 0]])
```

```
>>> M2 = M.convert_to(FF(7))
>>> print(M2.to_Matrix().transpose())
Matrix([[1, 0, 0], [2, 3, -3]])
>>> print(supplement_a_subspace(M2).to_Matrix().transpose())
Matrix([[1, 0, 0], [2, 3, -3], [0, 1, 0]])
```

References

万兴PDF

[R718]

sympy.polys.numberfields.utilities.**isolate**(*alg*, *eps=None*, *fast=False*)

Find a rational isolating interval for a real algebraic number.

Parameters

alg : str, int, *Expr* (page 947)

The algebraic number to be isolated. Must be a real number, to use this particular function. However, see also *Poly.intervals()* (page 2398), which isolates complex roots when you pass `all=True`.

eps : positive element of *QQ* (page 2529), None, optional (default=None)

Precision to be passed to *Poly.refine_root()* (page 2412)

fast : boolean, optional (default=False)

Say whether fast refinement procedure should be used. (Will be passed to *Poly.refine_root()* (page 2412).)

Returns

Pair of rational numbers defining an isolating interval for the given algebraic number.

Examples

```
>>> from sympy import isolate, sqrt, Rational
>>> print(isolate(sqrt(2)))
(1, 2)
>>> print(isolate(sqrt(2), eps=Rational(1, 100)))
(24/17, 17/12)
```

See also:

Poly.intervals (page 2398)

Category Theory

Introduction

The category theory module for SymPy will allow manipulating diagrams within a single category, including drawing them in TikZ and deciding whether they are commutative or not.

The general reference work this module tries to follow is

The latest version of this book should be available for free download from

katmat.math.uni-bremen.de/acc/acc.pdf

The module is still in its pre-embryonic stage.

Base Class Reference

This section lists the classes which implement some of the basic notions in category theory: objects, morphisms, categories, and diagrams.

class sympy.categories.**Object**(*name*, ***assumptions*)

The base class for any kind of object in an abstract category.

Explanation

While technically any instance of *Basic* (page 927) will do, this class is the recommended way to create abstract objects in abstract categories.

class sympy.categories.**Morphism**(*domain*, *codomain*)

The base class for any morphism in an abstract category.

Explanation

In abstract categories, a morphism is an arrow between two category objects. The object where the arrow starts is called the domain, while the object where the arrow ends is called the codomain.

Two morphisms between the same pair of objects are considered to be the same morphisms. To distinguish between morphisms between the same objects use *NamedMorphism* (page 2744).

It is prohibited to instantiate this class. Use one of the derived classes instead.

See also:

IdentityMorphism (page 2746), *NamedMorphism* (page 2744), *CompositeMorphism* (page 2744)

property codomain

Returns the codomain of the morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.codomain
Object("B")
```

compose(*other*)

Composes self with the supplied morphism.

The order of elements in the composition is the usual order, i.e., to construct $g \circ f$ use `g.compose(f)`.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> (g * f).domain
Object("A")
>>> (g * f).codomain
Object("C")
```

property domain

Returns the domain of the morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.domain
Object("A")
```

class sympy.categories.**NamedMorphism**(*domain*, *codomain*, *name*)

Represents a morphism which has a name.

Explanation

Names are used to distinguish between morphisms which have the same domain and codomain: two named morphisms are equal if they have the same domains, codomains, and names.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f
NamedMorphism(Object("A"), Object("B"), "f")
>>> f.name
'f'
```

See also:

Morphism (page 2742)

property name

Returns the name of the morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> f.name
'f'
```

class sympy.categories.**CompositeMorphism**(**components*)

Represents a morphism which is a composition of other morphisms.

万兴PDF

Explanation

Two composite morphisms are equal if the morphisms they were obtained from (components) are the same and were listed in the same order.

The arguments to the constructor for this class should be listed in diagram order: to obtain the composition $g \circ f$ from the instances of *Morphism* (page 2742) `g` and `f` use `CompositeMorphism(f, g)`.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, CompositeMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> g * f
CompositeMorphism((NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g")))
>>> CompositeMorphism(f, g) == g * f
True
```

property codomain

Returns the codomain of this composite morphism.

The codomain of the composite morphism is the codomain of its last component.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).codomain
Object("C")
```

property components

Returns the components of this composite morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).components
(NamedMorphism(Object("A"), Object("B"), "f"),
NamedMorphism(Object("B"), Object("C"), "g"))
```

property domain

Returns the domain of this composite morphism.

The domain of the composite morphism is the domain of its first component.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).domain
Object("A")
```

flatten(*new_name*)

Forgets the composite structure of this morphism.

Explanation

If new_name is not empty, returns a *NamedMorphism* (page 2744) with the supplied name, otherwise returns a *Morphism* (page 2742). In both cases the domain of the new morphism is the domain of this composite morphism and the codomain of the new morphism is the codomain of this composite morphism.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> (g * f).flatten("h")
NamedMorphism(Object("A"), Object("C"), "h")
```

万兴PDF

class sympy.categories.**IdentityMorphism**(*domain*)

Represents an identity morphism.

Explanation

An identity morphism is a morphism with equal domain and codomain, which acts as an identity with respect to composition.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, IdentityMorphism
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> f * id_A == f
True
>>> id_B * f == f
True
```

See also:

Morphism (page 2742)

class sympy.categories.**Category**(*name*, *objects=EmptySet*, *commutative_diagrams=EmptySet*)

An (abstract) category.

Explanation

A category [JoyOfCats] is a quadruple $\mathrm{K} = (O, \mathrm{hom}, id, \circ)$ consisting of

- a (set-theoretical) class O, whose members are called K-objects,
- for each pair (A, B) of K-objects, a set $\mathrm{hom}(A, B)$ whose members are called K-morphisms from A to B,
- for a each K-object A, a morphism $id : A \to A$, called the K-identity of A,
- a composition law $\circ$ associating with every K-morphisms $f : A \to B$ and $g : B \to C$ a K-morphism $g \circ f : A \to C$, called the composite of f and g.

Composition is associative, K-identities are identities with respect to composition, and the sets $\mathrm{hom}(A, B)$ are pairwise disjoint.

This class knows nothing about its objects and morphisms. Concrete cases of (abstract) categories should be implemented as classes derived from this one.

Certain instances of *Diagram* (page 2749) can be asserted to be commutative in a *Category* (page 2747) by supplying the argument `commutative_diagrams` in the constructor.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

See also:

Diagram (page 2749)

property commutative_diagrams

Returns the *FiniteSet* (page 1197) of diagrams which are known to be commutative in this category.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram,␣
↪Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> K = Category("K", commutative_diagrams=[d])
>>> K.commutative_diagrams == FiniteSet(d)
True
```

property name

Returns the name of this category.

Examples

```
>>> from sympy.categories import Category
>>> K = Category("K")
>>> K.name
'K'
```

property objects

Returns the class of objects of this category.

Examples

```
>>> from sympy.categories import Object, Category
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> K = Category("K", FiniteSet(A, B))
>>> K.objects
Class({Object("A"), Object("B")})
```

class sympy.categories.**Diagram**(**args*)

Represents a diagram in a certain category.

Explanation

Informally, a diagram is a collection of objects of a category and certain morphisms between them. A diagram is still a monoid with respect to morphism composition; i.e., identity morphisms, as well as all composites of morphisms included in the diagram belong to the diagram. For a more formal approach to this notion see [Pare1970].

The components of composite morphisms are also added to the diagram. No properties are assigned to such morphisms by default.

A commutative diagram is often accompanied by a statement of the following kind: "if such morphisms with such properties exist, then such morphisms which such properties exist and the diagram is commutative". To represent this, an instance of *Diagram* (page 2749) includes a collection of morphisms which are the premises and another collection of conclusions. `premises` and `conclusions` associate morphisms belonging to the corresponding categories with the *FiniteSet* (page 1197)'s of their properties.

The set of properties of a composite morphism is the intersection of the sets of properties of its components. The domain and codomain of a conclusion morphism should be among the domains and codomains of the morphisms listed as the premises of a diagram.

No checks are carried out of whether the supplied object and morphisms do belong to one and the same category.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import pprint, default_sort_key
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> premises_keys = sorted(d.premises.keys(), key=default_sort_key)
>>> pprint(premises_keys, use_unicode=False)
[g*f:A-->C, id:A-->A, id:B-->B, id:C-->C, f:A-->B, g:B-->C]
>>> pprint(d.premises, use_unicode=False)
{g*f:A-->C: EmptySet, id:A-->A: EmptySet, id:B-->B: EmptySet, id:C-->C:␣
```

(continues on next page)

(continued from previous page)

```
↪EmptyS
et, f:A-->B: EmptySet, g:B-->C: EmptySet}
>>> d = Diagram([f, g], {g * f: "unique"})
>>> pprint(d.conclusions,use_unicode=False)
{g*f:A-->C: {unique}}
```

References

[Pare1970] B. Pareigis: Categories and functors. Academic Press, 1970.

property conclusions

Returns the conclusions of this diagram.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> IdentityMorphism(A) in d.premises.keys()
True
>>> g * f in d.premises.keys()
True
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d.conclusions[g * f] == FiniteSet("unique")
True
```

hom(A, B)

Returns a 2-tuple of sets of morphisms between objects `A` and `B`: one set of morphisms listed as premises, and the other set of morphisms listed as conclusions.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import pretty
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> print(pretty(d.hom(A, C), use_unicode=False))
({g*f:A-->C}, {g*f:A-->C})
```

See also:

Object (page 2742), *Morphism* (page 2742)

is_subdiagram(*diagram*)

Checks whether `diagram` is a subdiagram of `self`. Diagram D' is a subdiagram of D if all premises (conclusions) of D' are contained in the premises (conclusions) of D. The morphisms contained both in D' and D should have the same properties for D' to be a subdiagram of D.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {g * f: "unique"})
>>> d1 = Diagram([f])
>>> d.is_subdiagram(d1)
True
>>> d1.is_subdiagram(d)
False
```

property objects

Returns the *FiniteSet* (page 1197) of objects that appear in this diagram.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g])
>>> d.objects
{Object("A"), Object("B"), Object("C")}
```

property premises

Returns the premises of this diagram.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import IdentityMorphism, Diagram
>>> from sympy import pretty
>>> A = Object("A")
>>> B = Object("B")
>>> f = NamedMorphism(A, B, "f")
>>> id_A = IdentityMorphism(A)
>>> id_B = IdentityMorphism(B)
>>> d = Diagram([f])
>>> print(pretty(d.premises, use_unicode=False))
{id:A-->A: EmptySet, id:B-->B: EmptySet, f:A-->B: EmptySet}
```

subdiagram_from_objects(*objects*)

If `objects` is a subset of the objects of `self`, returns a diagram which has as premises all those premises of `self` which have a domains and codomains in `objects`, likewise for conclusions. Properties are preserved.

Examples

```
>>> from sympy.categories import Object, NamedMorphism, Diagram
>>> from sympy import FiniteSet
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> d = Diagram([f, g], {f: "unique", g*f: "veryunique"})
>>> d1 = d.subdiagram_from_objects(FiniteSet(A, B))
>>> d1 == Diagram([f], {f: "unique"})
True
```

Diagram Drawing

This section lists the classes which allow automatic drawing of diagrams.

class sympy.categories.diagram_drawing.**DiagramGrid**(*diagram*, *groups=None*, ***hints*)

Constructs and holds the fitting of the diagram into a grid.

万兴PDF

Explanation

The mission of this class is to analyse the structure of the supplied diagram and to place its objects on a grid such that, when the objects and the morphisms are actually drawn, the diagram would be "readable", in the sense that there will not be many intersections of moprhisms. This class does not perform any actual drawing. It does strive nevertheless to offer sufficient metadata to draw a diagram.

Consider the following simple diagram.

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> from sympy import pprint
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
```

The simplest way to have a diagram laid out is the following:

```
>>> grid = DiagramGrid(diagram)
>>> (grid.width, grid.height)
(2, 2)
>>> pprint(grid)
A  B

   C
```

Sometimes one sees the diagram as consisting of logical groups. One can advise `DiagramGrid` as to such groups by employing the `groups` keyword argument.

Consider the following diagram:

```
>>> D = Object("D")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(D, A, "h")
>>> k = NamedMorphism(D, B, "k")
>>> diagram = Diagram([f, g, h, k])
```

Lay it out with generic layout:

```
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A  B  D

   C
```

Now, we can group the objects A and D to have them near one another:

```
>>> grid = DiagramGrid(diagram, groups=[[A, D], B, C])
>>> pprint(grid)
B     C
```

(continues on next page)

(continued from previous page)

```
A  D
```

Note how the positioning of the other objects changes.

Further indications can be supplied to the constructor of *DiagramGrid* (page 2752) using keyword arguments. The currently supported hints are explained in the following paragraphs.

DiagramGrid (page 2752) does not automatically guess which layout would suit the supplied diagram better. Consider, for example, the following linear diagram:

```
>>> E = Object("E")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> h = NamedMorphism(C, D, "h")
>>> i = NamedMorphism(D, E, "i")
>>> diagram = Diagram([f, g, h, i])
```

When laid out with the generic layout, it does not get to look linear:

```
>>> grid = DiagramGrid(diagram)
>>> pprint(grid)
A  B

   C  D

      E
```

To get it laid out in a line, use `layout="sequential"`:

```
>>> grid = DiagramGrid(diagram, layout="sequential")
>>> pprint(grid)
A  B  C  D  E
```

One may sometimes need to transpose the resulting layout. While this can always be done by hand, *DiagramGrid* (page 2752) provides a hint for that purpose:

```
>>> grid = DiagramGrid(diagram, layout="sequential", transpose=True)
>>> pprint(grid)
A

B

C

D

E
```

Separate hints can also be provided for each group. For an example, refer to `tests/test_drawing.py`, and see the different ways in which the five lemma [FiveLemma] can be laid out.

See also:

Diagram (page 2749)

References

[FiveLemma]

property height

Returns the number of rows in this diagram layout.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.height
2
```

property morphisms

Returns those morphisms (and their properties) which are sufficiently meaningful to be drawn.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.morphisms
{NamedMorphism(Object("A"), Object("B"), "f"): EmptySet,
NamedMorphism(Object("B"), Object("C"), "g"): EmptySet}
```

property width

Returns the number of columns in this diagram layout.

Examples

```
>>> from sympy.categories import Object, NamedMorphism
>>> from sympy.categories import Diagram, DiagramGrid
>>> A = Object("A")
>>> B = Object("B")
>>> C = Object("C")
>>> f = NamedMorphism(A, B, "f")
>>> g = NamedMorphism(B, C, "g")
>>> diagram = Diagram([f, g])
>>> grid = DiagramGrid(diagram)
>>> grid.width
2
```

class sympy.categories.diagram_drawing.**ArrowStringDescription**(*unit*, *curving*, *curving_amount*, *looping_start*, *looping_end*, *horizontal_direction*, *vertical_direction*, *label_position*, *label*)

Stores the information necessary for producing an Xy-pic description of an arrow.

The principal goal of this class is to abstract away the string representation of an arrow and to also provide the functionality to produce the actual Xy-pic string.

`unit` sets the unit which will be used to specify the amount of curving and other distances. `horizontal_direction` should be a string of "`r`" or "`l`" specifying the horizontal offset of the target cell of the arrow relatively to the current one. `vertical_direction` should specify the vertical offset using a series of either "`d`" or "`u`". `label_position` should be either "`^`", "`_`", or "`|`" to specify that the label should be positioned above the arrow, below the arrow or just over the arrow, in a break. Note that the notions "above" and "below" are relative to arrow direction. `label` stores the morphism label.

This works as follows (disregard the yet unexplained arguments):

```
>>> from sympy.categories.diagram_drawing import ArrowStringDescription
>>> astr = ArrowStringDescription(
... unit="mm", curving=None, curving_amount=None,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar[dr]_{f}
```

`curving` should be one of "`^`", "`_`" to specify in which direction the arrow is going to curve. `curving_amount` is a number describing how many `unit`'s the morphism is going to curve:

```
>>> astr = ArrowStringDescription(
... unit="mm", curving="^", curving_amount=12,
... looping_start=None, looping_end=None, horizontal_direction="d",
... vertical_direction="r", label_position="_", label="f")
>>> print(str(astr))
\ar@/^12mm/[dr]_{f}
```