Introducing the Domains of the poly module

This page introduces the idea of the "domains" that are used in SymPy's *sympy.polys* (page 2360) module. The emphasis is on introducing how to use the domains directly and on understanding how they are used internally as part of the *Poly* (page 2378) class. This is a relatively advanced topic so for a more introductory understanding of the *Poly* (page 2378) class and the *sympy.polys* (page 2360) module it is recommended to read *Basic functionality of the module* (page 2342) instead. The reference documentation for the domain classes is in *Reference docs for the Poly Domains* (page 2504). Internal functions that make use of the domains are documented in *Internals of the Polynomial Manipulation Module* (page 2569).

What are the domains?

For most users the domains are only really noticeable in the printed output of a *Poly* (page 2378):

```
>>> from sympy import Symbol, Poly
>>> x = Symbol('x')
>>> Poly(x**2 + x)
Poly(x**2 + x, x, domain='ZZ')
>>> Poly(x**2 + x/2)
Poly(x**2 + 1/2*x, x, domain='QQ')
```

We see here that one *Poly* (page 2378) has domain *ZZ* (page 2525) representing the integers and the other has domain *QQ* (page 2529) representing the rationals. These indicate the "domain" from which the coefficients of the polynomial are drawn.

From a high-level the domains represent formal concepts such as the set of integers $\mathbb{Z}$ or rationals $\mathbb{Q}$. The word "domain" here is a reference to the mathematical concept of an integral domain.

Internally the domains correspond to different computational implementations and representations of the expressions that the polynomials correspond to. The *Poly* (page 2378) object itself has an internal representation as a `list` of coefficients and a *domain* (page 2391) attribute representing the implementation of those coefficients:

```
>>> p = Poly(x**2 + x/2)
>>> p
Poly(x**2 + 1/2*x, x, domain='QQ')
>>> p.domain
QQ
>>> p.rep
DMP([1, 1/2, 0], QQ, None)
>>> p.rep.rep
[1, 1/2, 0]
>>> type(p.rep.rep[0])
<class 'sympy.external.pythonmpq.PythonMPQ'>
```

Here the domain is *QQ* (page 2529) which represents the implementation of the rational numbers in the domain system. The *Poly* (page 2378) instance itself has a *Poly.domain* (page 2391) attribute *QQ* (page 2529) and then a list of *PythonMPQ* (page 2532) coefficients where *PythonMPQ* (page 2532) is the class that implements the elements of the *QQ* (page 2529) domain. The list of coefficients `[1, 1/2, 0]` gives a standardised low-level representation of the polynomial expression `(1)*x**2 + (1/2)*x + (0)`.

This page looks at the different domains that are defined in SymPy, how they are implemented and how they can be used. It introduces how to use the domains and domain elements directly and explains how they are used internally as part of *Poly* (page 2378) objects. This information is more relevant for development in SymPy than it is for users of the *sympy.polys* (page 2360) module.

Representing expressions symbolically

There are many different ways that a mathematical expression can be represented symbolically. The purpose of the polynomial domains is to provide suitable implementations for different classes of expressions. This section considers the basic approaches to the symbolic representation of mathematical expressions: "tree", "dense polynomial" and "sparse polynomial".

Tree representation

The most general representation of symbolic expressions is as a tree and this is the representation used for most ordinary SymPy expressions which are instances of *Expr* (page 947) (a subclass of *Basic* (page 927)). We can see this representation using the *srepr()* (page 2178) function:

```
>>> from sympy import Symbol, srepr
>>> x = Symbol('x')
>>> e = 1 + 1/(2 + x**2)
>>> e
1 + 1/(x**2 + 2)
>>> print(srepr(e))
Add(Integer(1), Pow(Add(Pow(Symbol('x'), Integer(2)), Integer(2)), Integer(-
↪1)))
```

Here the expression `e` is represented as an *Add* (page 1013) node which has two children `1` and `1/(x**2 + 2)`. The child `1` is represented as an *Integer* (page 987) and the other child is represented as a *Pow* (page 1005) with base `x**2 + 2` and exponent `1`. Then `x**2 + 2` is represented as an *Add* (page 1013) with children `x**2` and `2` and so on. In this way the expression is represented as a tree where the internal nodes are operations like *Add* (page 1013), *Mul* (page 1009), *Pow* (page 1005) and so on and the leaf nodes are atomic expression types like *Integer* (page 987) and *Symbol* (page 976). See *Advanced Expression Manipulation* (page 61) for more about this representation.

The tree representation is core to the architecture of *Expr* (page 947) in SymPy. It is a highly flexible representation that can represent a very wide range of possible expressions. It can also represent equivalent expressions in different ways e.g.:

```
>>> e = x*(x + 1)
>>> e
x*(x + 1)
>>> e.expand()
x**2 + x
```

These two expression although equivalent have different tree representations:

```
>>> print(srepr(e))
Mul(Symbol('x'), Add(Symbol('x'), Integer(1)))
>>> print(srepr(e.expand()))
Add(Pow(Symbol('x'), Integer(2)), Symbol('x'))
```

Being able to represent the same expression in different ways is both a strength and a weakness. It is useful to be able to convert an expression in to different forms for different tasks but having non-unique representations makes it hard to tell when two expressions are equivalent which is in fact very important for many computational algorithms. The most important task is being able to tell when an expression is equal to zero which is undecidable in general (Richardon's theorem) but is decidable in many important special cases.

DUP representation

Restricting the set of allowed expressions to special cases allows for much more efficient symbolic representations. As we already saw *Poly* (page 2378) can represent a polynomial as a list of coefficients. This means that an expression like `x**4 + x + 1` could be represented simply as `[1, 0, 0, 1, 1]`. This list of coefficients representation of a polynomial expression is known as the "dense univariate polynomial" (DUP) representation. Working within that representation algorithms for multiplication, addition and crucially zero-testing can be much more efficient than with the corresponding tree representations. We can see this representation from a *Poly* (page 2378) instance by looking it its `rep.rep` attribute:

```
>>> p = Poly(x**4 + x + 1)
>>> p.rep.rep
[1, 0, 0, 1, 1]
```

In the DUP representation it is not possible to represent the same expression in different ways. There is no distinction between `x*(x + 1)` and `x**2 + x` because both are just `[1, 1, 0]`. This means that comparing two expressions is easy: they are equal if and only if all of their coefficients are equal. Zero-testing is particularly easy: the polynomial is zero if and only if all coefficients are zero (of course we need to have easy zero-testing for the coefficients themselves).

We can make functions that operate on the DUP representation much more efficiently than functions that operate on the tree representation. Many operations with standard sympy expressions are in fact computed by converting to a polynomial representation and then performing the calculation. An example is the *factor()* (page 2373) function:

```
>>> from sympy import factor
>>> e = 2*x**3 + 10*x**2 + 16*x + 8
>>> e
2*x**3 + 10*x**2 + 16*x + 8
>>> factor(e)
2*(x + 1)*(x + 2)**2
```

Internally *factor()* (page 2373) will convert the expression from the tree representation into the DUP representation and then use the function `dup_factor_list`:

```
>>> from sympy import ZZ
>>> from sympy.polys.factortools import dup_factor_list
>>> p = [ZZ(2), ZZ(10), ZZ(16), ZZ(8)]
```

(continues on next page)

(continued from previous page)

```
>>> p
[2, 10, 16, 8]
>>> dup_factor_list(p, ZZ)
(2, [([1, 1], 1), ([1, 2], 2)])
```

There are many more examples of functions with `dup_*` names for operating on the DUP representation that are documented in *Internals of the Polynomial Manipulation Module* (page 2569). There are also functions with the `dmp_*` prefix for operating on multivariate polynomials.

DMP representation

A multivariate polynomial (a polynomial in multiple variables) can be represented as a polynomial with coefficients that are themselves polynomials. For example `x**2*y + x**2 + x*y + y + 1` can be represented as polynomial in `x` where the coefficients are themselves polynomials in `y` i.e.: `(y + 1)*x**2 + (y)*x + (y+1)`. Since we can represent a polynomial with a list of coefficients a multivariate polynomial can be represented with a list of lists of coefficients:

```
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> p = Poly(x**2*y + x**2 + x*y + y + 1)
>>> p
Poly(x**2*y + x**2 + x*y + y + 1, x, y, domain='ZZ')
>>> p.rep.rep
[[1, 1], [1, 0], [1, 1]]
```

This list of lists of (lists of...) coefficients representation is known as the “dense multivariate polynomial” (DMP) representation.

Sparse polynomial representation

Instead of lists we can use a dict mapping nonzero monomial terms to their coefficients. This is known as the “sparse polynomial” representation. We can see what this would look like using the *`as_dict()`* (page 2382) method:

```
>>> Poly(7*x**20 + 8*x + 9).rep.rep
[7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 9]
>>> Poly(7*x**20 + 8*x + 9).as_dict()
{(0,): 9, (1,): 8, (20,): 7}
```

The keys of this dict are the exponents of the powers of `x` and the values are the coefficients so e.g. `7*x**20` becomes `(20,): 7` in the dict. The key is a tuple so that in the multivariate case something like `4*x**2*y**3` can be represented as `(2, 3): 4`. The sparse representation can be more efficient as it avoids the need to store and manipulate the zero coefficients. With a large number of generators (variables) the dense representation becomes particularly inefficient and it is better to use the sparse representation:

```
>>> from sympy import prod
>>> gens = symbols('x:10')
>>> gens
```

(continues on next page)

(continued from previous page)

```
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)
>>> p = Poly(prod(gens))
>>> p
Poly(x0*x1*x2*x3*x4*x5*x6*x7*x8*x9, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9,␣
 ↪domain='ZZ')
>>> p.rep.rep
[[[[[[[[[1, 0], []], [[]]], [[]]]], [[[]]]], [[[[]]]]], [[[[[]]]]], [[[[[[]]]]]],␣
 ↪[[[[[[[[]]]]]]]], [[[[[[[[]]]]]]]], [[[[[[[[]]]]]]]]]
>>> p.as_dict()
{(1, 1, 1, 1, 1, 1, 1, 1, 1, 1): 1}
```

The dict representation shown in the last output maps from the monomial which is represented as a tuple of powers (`(1, 1, 1, ...)` i.e. `x0**1 * x1**1, ...`) to the coefficient 1. Compared to the *DMP representation* (page 2480) we have a much more flattened data structure: it is a `dict` with only one key and value. Algorithms for working with sparse representations would likely be much more efficient than dense algorithms for this particular example polynomial.

SymPy's polynomial module has implementations of polynomial expressions based on both the dense and sparse representations. There are also other implementations of different special classes of expressions that can be used as the coefficients of those polynomials. The rest of this page discusses what those representations are and how to use them.

Basic usage of domains

Several domains are predefined and ready to be used such as *ZZ* (page 2525) and *QQ* (page 2529) which represent the ring of integers $\mathbb{Z}$ and the field of rationals $\mathbb{Q}$. The *Domain* (page 2504) object is used to construct elements which can then be used in ordinary arithmetic operations.:

```
>>> from sympy import ZZ
>>> z1 = ZZ(2)
>>> z1
2
>>> z1 + z1
4
>>> type(z1)
<class 'int'>
>>> z1 in ZZ
True
```

The basic operations `+`, `-`, and `*` for addition, subtraction and multiplication will work for the elements of any domain and will produce new domain elements. Division with `/` (Python's "true division" operator) is not possible for all domains and should not be used with domain elements unless the domain is known to be a field. For example dividing two elements of *ZZ* (page 2525) gives a `float` which is not an element of *ZZ* (page 2525):

```
>>> z1 / z1
1.0
>>> type(z1 / z1)
<class 'float'>
```

(continues on next page)

(continued from previous page)

```
>>> ZZ.is_Field
False
```

Most domains representing non-field rings allow floor and modulo division (remainder) with Python's floor division `//` and modulo division `%` operators. For example with *ZZ* (page 2525):

```
>>> z1 // z1
1
>>> z1 % z1
0
```

The *QQ* (page 2529) domain represents the field of rational numbers and does allow division:

```
>>> from sympy import QQ
>>> q1 = QQ(1, 2)
>>> q1
1/2
>>> q2 = QQ(2, 3)
>>> q2
2/3
>>> q1 / q2
3/4
>>> type(q1)
<class 'sympy.external.pythonmpq.PythonMPQ'>
```

In general code that is expected to work with elements of an arbitrary domain should not use the division operators `/`, `//` and `%`. Only the operators `+`, `-`, `*` and `**` (with nonnegative integer exponent) should be assumed to work with arbitrary domain elements. All other operations should be accessed as functions from the *Domain* (page 2504) object:

```
>>> ZZ.quo(ZZ(5), ZZ(3))  # 5 // 3
1
>>> ZZ.rem(ZZ(5), ZZ(3))  # 5 % 3
2
>>> ZZ.div(ZZ(5), ZZ(3))  # divmod(5, 3)
(1, 2)
>>> QQ.div(QQ(5), QQ(3))
(5/3, 0)
```

The *exquo()* (page 2511) function is used to compute an exact quotient. This is the analogue of `a / b` but where the division is expected to be exact (with no remainder) or an error will be raised:

```
>>> QQ.exquo(QQ(5), QQ(3))
5/3
>>> ZZ.exquo(ZZ(4), ZZ(2))
2
>>> ZZ.exquo(ZZ(5), ZZ(3))
Traceback (most recent call last):
...
ExactQuotientFailed: 3 does not divide 5 in ZZ
```

The exact methods and attributes of the domain elements are not guaranteed in general beyond the basic arithmetic operations. It should not be presumed that e.g. *ZZ* (page 2525) will

always be of type `int`. If `gmpy` or `gmpy2` is installed then the `mpz` or `mpq` types are used instead for *ZZ* (page 2525) and *QQ* (page 2529):

```
>>> from sympy import ZZ, QQ
>>> ZZ(2)
mpz(2)
>>> QQ(2, 3)
mpq(2, 3)
```

The `mpz` type is faster than Python's standard `int` type for operations with large integers although for smaller integers the difference is not so significant. The `mpq` type representing rational numbers is implemented in C rather than Python and is many times faster than the pure Python implementation of *QQ* (page 2529) that is used when gmpy is not installed.

In general the Python type used for the elements of a domain can be checked from the *dtype* (page 2510) attribute of the domain. When gmpy is installed the dtype for *ZZ* (page 2525) is mpz which is not an actual type and can not be used with $isinstance$. For this reason the *of_type()* (page 2516) method can be used to check if an object is an element of *dtype* (page 2510).:

```
>>> z = ZZ(2)
>>> type(z)
<class 'int'>
>>> ZZ.dtype
<class 'int'>
>>> ZZ.of_type(z)
True
```


Domain elements vs sympy expressions

Note that domain elements are not of the same type as ordinary sympy expressions which are subclasses of *Expr* (page 947) such as *Integer* (page 987). Ordinary sympy expressions are created with the *sympify()* (page 918) function.:

```
>>> from sympy import sympify
>>> z1_sympy = sympify(2)  # Normal sympy object
>>> z1_sympy
2
>>> type(z1_sympy)
<class 'sympy.core.numbers.Integer'>
>>> from sympy import Expr
>>> isinstance(z1_sympy, Expr)
True
```

It is important when working with the domains not to mix sympy expressions with domain elements even though it will sometimes work in simple cases. Each domain object has the methods *to_sympy()* (page 2517) and *from_sympy()* (page 2513) for converting back and forth between sympy expressions and domain elements:

```
>>> z_sympy = sympify(2)
>>> z_zz = ZZ.from_sympy(z_sympy)
>>> z_zz
2
```

(continues on next page)

(continued from previous page)

```
>>> type(z_sympy)
<class 'sympy.core.numbers.Integer'>
>>> type(z_zz)
<class 'int'>
>>> ZZ.to_sympy(z_zz)
2
>>> type(ZZ.to_sympy(z_zz))
<class 'sympy.core.numbers.Integer'>
```

Any particular domain will only be able to represent some sympy expressions so conversion will fail if the expression can not be represented in the domain:

```
>>> from sympy import sqrt
>>> e = sqrt(2)
>>> e
sqrt(2)
>>> ZZ.from_sympy(e)
Traceback (most recent call last):
...
CoercionFailed: expected an integer, got sqrt(2)
```

We have already seen that in some cases we can use the domain object itself as a constructor e.g. `QQ(2)`. This will generally work provided the arguments given are valid for the *dtype* (page 2510) of the domain. Although it is convenient to use this in interactive sessions and in demonstrations it is generally better to use the *from_sympy()* (page 2513) method for constructing domain elements from sympy expressions (or from objects that can be sympified to sympy expressions).

It is important not to mix domain elements with other Python types such as `int`, `float`, as well as standard sympy *Expr* (page 947) expressions. When working in a domain, care should be taken as some Python operations will do this implicitly. for example the `sum` function will use the regular `int` value of zero so that `sum([a, b])` is effectively evaluated as `(0 + a) + b` where `0` is of type `int`.

Every domain is at least a ring if not a field and as such is guaranteed to have two elements in particular corresponding to 1 and 0. The domain object provides domain elements for these as the attributes *one* (page 2516) and *zero* (page 2519). These are useful for something like Python's `sum` function which allows to provide an alternative object as the "zero":

```
>>> ZZ.one
1
>>> ZZ.zero
0
>>> sum([ZZ(1), ZZ(2)])  # don't do this (even it sometimes works)
3
>>> sum([ZZ(1), ZZ(2)], ZZ.zero) # provide the zero from the domain
3
```

A standard pattern then for performing calculations in a domain is:

1. Start with sympy *Expr* (page 947) instances representing expressions.
2. Choose an appropriate domain that can represent the expressions.
3. Convert all expressions to domain elements using *from_sympy()* (page 2513).

4. Perform the calculation with the domain elements.
5. Convert back to *Expr* (page 947) with *to_sympy()* (page 2517).

Here is an implementation of the `sum` function that illustrates these steps and sums some integers but performs the calculation using the domain elements rather than standard sympy expressions:

```
def sum_domain(expressions_sympy):
    """Sum sympy expressions but performing calculations in domain ZZ"""

    # Convert to domain
    expressions_dom = [ZZ.from_sympy(e) for e in expressions_sympy]

    # Perform calculations in the domain
    result_dom = ZZ.zero
    for e_dom in expressions_dom:
        result_dom += e_dom

    # Convert the result back to Expr
    result_sympy = ZZ.to_sympy(result_dom)
    return result_sympy
```

Gaussian integers and Gaussian rationals

The two example domains that we have seen so far are *ZZ* (page 2525) and *QQ* (page 2529) representing the integers and the rationals respectively. There are other simple domains such as *ZZ_I* (page 2534) and *QQ_I* (page 2536) representing the Gaussian integers and Gaussian rationals. The Gaussian integers are numbers of the form $a\sqrt{-1}+b$ where a and b are integers. The Gaussian rationals are defined similarly except that a and b can be rationals. We can use the Gaussian domains like:

```
>>> from sympy import ZZ_I, QQ_I, I
>>> z = ZZ_I.from_sympy(1 + 2*I)
>>> z
(1 + 2*I)
>>> z**2
(-3 + 4*I)
```

Note the contrast with the way this calculation works in the tree representation where *expand()* (page 1053) is needed to get the reduced form:

```
>>> from sympy import expand, I
>>> z = 1 + 2*I
>>> z**2
(1 + 2*I)**2
>>> expand(z**2)
-3 + 4*I
```

The *ZZ_I* (page 2534) and *QQ_I* (page 2536) domains are implemented by the classes *GaussianIntegerRing* (page 2534) and *GaussianRationalField* (page 2536) and their elements by *GaussianInteger* (page 2536) and *GaussianRational* (page 2539) respectively. The internal representation for an element of *ZZ_I* (page 2534) or *QQ_I* (page 2536) is simply as a pair `(a, b)` of elements of *ZZ* (page 2525) or *QQ* (page 2529) respectively. The domain

ZZ_I (page 2534) is a ring with similar properties to *ZZ* (page 2525) whereas *QQ_I* (page 2536) is a field much like *QQ* (page 2529):

```
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
>>> ZZ_I.is_Field
False
>>> QQ_I.is_Field
True
```

Since *QQ_I* (page 2536) is a field division by nonzero elements is always possible whereas in *ZZ_I* (page 2534) we have the important concept of the greatest common divisor (GCD):

```
>>> e1 = QQ_I.from_sympy(1+I)
>>> e2 = QQ_I.from_sympy(2-I/2)
>>> e1/e2
(6/17 + 10/17*I)
>>> ZZ_I.gcd(ZZ_I(5), ZZ_I.from_sympy(1+2*I))
(1 + 2*I)
```

Finite fields

So far we have seen the domains *ZZ* (page 2525), *QQ* (page 2529), *ZZ_I* (page 2534), and *QQ_I* (page 2536). There are also domains representing the Finite fields although the implementation of these is incomplete. A finite field *GF(p)* (page 2522) of *prime* order can be constructed with `FF` or `GF`. A domain for the finite field of prime order p can be constructed with *GF(p)* (page 2522):

```
>>> from sympy import GF
>>> K = GF(5)
>>> two = K(2)
>>> two
2 mod 5
>>> two ** 2
4 mod 5
>>> two ** 3
3 mod 5
```

There is also `FF` as an alias for `GF` (standing for "finite field" and "Galois field" respectively). These are equivalent and both `FF(n)` and `GF(n)` will create a domain which is an instance of *FiniteField* (page 2522). The associated domain elements will be instances of *PythonFiniteField* (page 2525) or *GMPYFiniteField* (page 2525) depending on whether or not `gmpy` is installed.

Finite fields of order p^n where $n \neq 1$ are not implemented. It is possible to use e.g. `GF(6)` or `GF(9)` but the resulting domain is *not* a field. It is just the integers modulo `6` or `9` and therefore has zero divisors and non-invertible elements:

```
>>> K = GF(6)
>>> K(3) * K(2)
0 mod 6
```

It would be good to have a proper implementation of prime-power order finite fields but this is not yet available in SymPy (contributions welcome!).

Real and complex fields

The fields *RR* (page 2545) and *CC* (page 2546) are intended mathematically to correspond to the reals and the complex numbers, $\mathbb{R}$ and $\mathbb{C}$ respectively. The implementation of these uses floating point arithmetic. In practice this means that these are the domains that are used to represent expressions containing floats. Elements of *RR* (page 2545) are instances of the class *RealElement* (page 2546) and have an `mpf` tuple which is used to represent a float in `mpmath`. Elements of *CC* (page 2546) are instances of *ComplexElement* (page 2546) and have an `mpc` tuple which is a pair of `mpf` tuples representing the real and imaginary parts. See the mpmath docs for more about how floating point numbers are represented:

```
>>> from sympy import RR, CC
>>> xr = RR(3)
>>> xr
3.0
>>> xr._mpf_
(0, 3, 0, 2)
>>> zc = CC(3+1j)
>>> zc
(3.0 + 1.0j)
>>> zc._mpc_
((0, 3, 0, 2), (0, 1, 0, 1))
```

The use of approximate floating point arithmetic in these domains comes with all of the usual pitfalls. Many algorithms in the *sympy.polys* (page 2360) module are fundamentally designed for exact arithmetic making the use of these domains potentially problematic:

```
>>> RR('0.1') + RR('0.2') == RR('0.3')
False
```

Since these are implemented using `mpmath` which is a multiprecision library it is possible to create different domains with different working precisions. The default domains *RR* (page 2545) and *CC* (page 2546) use 53 binary digits of precision much like standard double precision floating point which corresponds to approximately 15 decimal digits:

```
>>> from sympy.polys.domains.realfield import RealField
>>> RR.precision
53
>>> RR.dps
15
>>> RR(1) / RR(3)
0.333333333333333
>>> RR100 = RealField(100)
>>> RR100.precision
100
>>> RR100.dps
29
>>> RR100(1) / RR100(3)
0.33333333333333333333333333333
```

There is however a bug in the implementation of this so that actually a global precision setting is used by all *RealElement* (page 2546). This means that just creating `RR100` above has altered the global precision and we will need to restore it in the doctest here:

```
>>> RR(1) / RR(3)  # wrong result!
0.33333333333333333333333333333
>>> dummy = RealField(53)  # hack to restore precision
>>> RR(1) / RR(3)  # restored
0.333333333333333
```

(Obviously that should be fixed!)

Algebraic number fields

An algebraic extension of the rationals $\mathbb{Q}$ is known as an algebraic number field and these are implemented in sympy as *QQ<a>* (page 2539). The natural syntax for these would be something like `QQ(sqrt(2))` however `QQ()` is already overloaded as the constructor for elements of *QQ* (page 2529). These domains are instead created using the *algebraic_field()* (page 2508) method e.g. `QQ.algebraic_field(sqrt(2))`. The resulting domain will be an instance of *AlgebraicField* (page 2539) with elements that are instances of *ANP* (page 2568).

The printing support for these is less developed but we can use *to_sympy()* (page 2517) to take advantage of the corresponding *Expr* (page 947) printing support:

```
>>> K = QQ.algebraic_field(sqrt(2))
>>> K
QQ<sqrt(2)>
>>> b = K.one + K.from_sympy(sqrt(2))
>>> b
ANP([1, 1], [1, 0, -2], QQ)
>>> K.to_sympy(b)
1 + sqrt(2)
>>> b ** 2
ANP([2, 3], [1, 0, -2], QQ)
>>> K.to_sympy(b**2)
2*sqrt(2) + 3
```

The raw printed display immediately shows the internal representation of the elements as *ANP* (page 2568) instances. The field $\mathbb{Q}(\sqrt{2})$ consists of numbers of the form $a\sqrt{2} + b$ where a and b are rational numbers. Consequently every number in this field can be represented as a pair `(a, b)` of elements of *QQ* (page 2529). The domain element stores these two in a list and also stores a list representation of the *minimal polynomial* for the extension element $\sqrt{2}$. There is a sympy function *minpoly()* (page 2711) that can compute the minimal polynomial of any algebraic expression over the rationals:

```
>>> from sympy import minpoly, Symbol
>>> x = Symbol('x')
>>> minpoly(sqrt(2), x)
x**2 - 2
```

In the dense polynomial representation as a list of coefficients this polynomial is represented as `[1, 0, -2]` as seen in the *ANP* (page 2568) display for the elements of `QQ<sqrt(2)>` above.

It is also possible to create an algebraic number field with multiple generators such as $\mathbb{Q}(\sqrt{2}, \sqrt{3})$:

```
>>> K = QQ.algebraic_field(sqrt(2), sqrt(3))
>>> K
QQ<sqrt(2) + sqrt(3)>
>>> sqrt2 = K.from_sympy(sqrt(2))
>>> sqrt3 = K.from_sympy(sqrt(3))
>>> p = (K.one + sqrt2) * (K.one + sqrt3)
>>> p
ANP([1/2, 1, -3/2], [1, 0, -10, 0, 1], QQ)
>>> K.to_sympy(p)
1 + sqrt(2) + sqrt(3) + sqrt(6)
>>> K.to_sympy(p**2)
4*sqrt(6) + 6*sqrt(3) + 8*sqrt(2) + 12
```

Here the algebraic extension $\mathbb{Q}(\sqrt{2}, \sqrt{3})$ is converted to the (isomorphic) $\mathbb{Q}(\sqrt{2} + \sqrt{3})$ with a single generator $\sqrt{2} + \sqrt{3}$. It is always possible to find a single generator like this due to the primitive element theorem. There is a sympy function *primitive_element()* (page 2712) that can compute the minimal polynomial for a primitive element of an extension:

```
>>> from sympy import primitive_element, minpoly
>>> e = primitive_element([sqrt(2), sqrt(3)], x)
>>> e[0]
x**4 - 10*x**2 + 1
>>> e[0].subs(x, sqrt(2) + sqrt(3)).expand()
0
```

The minimal polynomial `x**4 - 10*x**2 + 1` has the dense list representation `[1, 0, -10, 0, 1]` as seen in the *ANP* (page 2568) output above. What the primitive element theorem means is that all algebraic number fields can be represented as an extension of the rationals by a single generator with some minimal polynomial. Calculations over the algebraic number field only need to take advantage of the minimal polynomial and that makes it possible to compute all arithmetic operations and also to carry out higher level operations like factorisation of polynomials.

Polynomial ring domains

There are also domains implemented to represent a polynomial ring like *K[x]* (page 2547) which is the domain of polynomials in the generator `x` with coefficients over another domain `K`:

```
>>> from sympy import ZZ, symbols
>>> x = symbols('x')
>>> K = ZZ[x]
>>> K
ZZ[x]
>>> x_dom = K(x)
>>> x_dom + K.one
x + 1
```

All the operations discussed before will work with elements of a polynomial ring:

```
>>> p = x_dom + K.one
>>> p
```

(continues on next page)

(continued from previous page)

```
x + 1
>>> p + p
2*x + 2
>>> p - p
0
>>> p * p
x**2 + 2*x + 1
>>> p ** 3
x**3 + 3*x**2 + 3*x + 1
>>> K.exquo(x_dom**2 - K.one, x_dom - K.one)
x + 1
```

The internal representation of elements of `K[x]` is different from the way that ordinary sympy (*Expr* (page 947)) expressions are represented. The *Expr* (page 947) representation of any expression is as a tree e.g.:

```
>>> from sympy import srepr
>>> K = ZZ[x]
>>> p_expr = x**2 + 2*x + 1
>>> p_expr
x**2 + 2*x + 1
>>> srepr(p_expr)
"Add(Pow(Symbol('x'), Integer(2)), Mul(Integer(2), Symbol('x')), Integer(1))"
```

Here the expression is a tree where the top node is an *Add* (page 1013) and its children nodes are *Pow* (page 1005) etc. This tree representation makes it possible to represent equivalent expressions in different ways e.g.:

```
>>> x = symbols('x')
>>> p_expr = x*(x + 1) + x
>>> p_expr
x*(x + 1) + x
>>> p_expr.expand()
x**2 + 2*x
```

By contrast the domain `ZZ[x]` represents only polynomials and does so by simply storing the non-zero coefficients of the expanded polynomial (the "sparse" polynomial representation). In particular elements of `ZZ[x]` are represented as a Python `dict`. Their type is *PolyElement* (page 2554) which is a subclass of `dict`. Converting to a normal dict shows the internal representation:

```
>>> x = symbols('x')
>>> K = ZZ[x]
>>> x_dom = K(x)
>>> p_dom = K(3)*x_dom**2 + K(2)*x_dom + K(7)
>>> p_dom
3*x**2 + 2*x + 7
>>> dict(p_dom)
{(0,): 7, (1,): 2, (2,): 3}
```

This internal form makes it impossible to represent unexpanded multiplications so any multiplication of elements of `ZZ[x]` will always be expanded:

```
>>> x = symbols('x')
>>> K = ZZ[x]
>>> x_dom = K(x)
>>> p_expr = x * (x + 1) + x
>>> p_expr
x*(x + 1) + x
>>> p_dom = x_dom * (x_dom + K.one) + x_dom
>>> p_dom
x**2 + 2*x
```

These same considerations apply to powers:

```
>>> (x + 1) ** 2
(x + 1)**2
>>> (x_dom + K.one) ** 2
x**2 + 2*x + 1
```

We can also construct multivariate polynomial rings:

```
>>> x, y = symbols('x, y')
>>> K = ZZ[x,y]
>>> xk = K(x)
>>> yk = K(y)
>>> xk**2*yk + xk + yk
x**2*y + x + y
```

It is also possible to construct nested polynomial rings (although it is less efficient). The ring `K[x][y]` is formally equivalent to `K[x,y]` although their implementations in sympy are different:

```
>>> K = ZZ[x][y]
>>> p = K(x**2 + x*y + y**2)
>>> p
y**2 + x*y + x**2
>>> dict(p)
{(0,): x**2, (1,): x, (2,): 1}
```

Here the coefficients like `x**2` are instances of *PolyElement* (page 2554) as well so this is a `dict` where the values are also dicts. The full representation is more like:

```
>>> {k: dict(v) for k, v in p.items()}
{(0,): {(2,): 1}, (1,): {(1,): 1}, (2,): {(0,): 1}}
```

The multivariate ring domain `ZZ[x,y]` has a more efficient representation as a single flattened `dict`:

```
>>> K = ZZ[x,y]
>>> p = K(x**2 + x*y + y**2)
>>> p
x**2 + x*y + y**2
>>> dict(p)
{(0, 2): 1, (1, 1): 1, (2, 0): 1}
```

The difference in efficiency between these representations grows as the number of generators increases i.e. `ZZ[x,y,z,t,...]` vs `ZZ[x][y][z][t]...`.

Old (dense) polynomial rings

In the last section we saw that the domain representation of a polynomial ring like *K[x]* (page 2547) uses a sparse representation of a polynomial as a dict mapping monomial exponents to coefficients. There is also an older version of *K[x]* (page 2547) that uses the dense *DMP representation* (page 2480). We can create these two versions of *K[x]* (page 2547) using *poly_ring()* (page 2516) and *old_poly_ring()* (page 2516) where the syntax `K[x]` is equivalent to `K.poly_ring(x)`:

```
>>> K1 = ZZ.poly_ring(x)
>>> K2 = ZZ.old_poly_ring(x)
>>> K1
ZZ[x]
>>> K2
ZZ[x]
>>> K1 == ZZ[x]
True
>>> K2 == ZZ[x]
False
>>> p1 = K1.from_sympy(x**2 + 1)
>>> p2 = K2.from_sympy(x**2 + 1)
>>> p1
x**2 + 1
>>> p2
x**2 + 1
>>> type(K1)
<class 'sympy.polys.domains.polynomialring.PolynomialRing'>
>>> type(p1)
<class 'sympy.polys.rings.PolyElement'>
>>> type(K2)
<class 'sympy.polys.domains.old_polynomialring.GlobalPolynomialRing'>
>>> type(p2)
<class 'sympy.polys.polyclasses.DMP'>
```

The internal representation of the old polynomial ring domain is the *DMP* (page 2561) representation as a list of (lists of) coefficients:

```
>>> repr(p2)
'DMP([1, 0, 1], ZZ, ZZ[x])'
```

The most notable use of the *DMP* (page 2561) representation of polynomials is as the internal representation used by *Poly* (page 2378) (this is discussed later in this page of the docs).

PolyRing vs PolynomialRing

You might just want to perform calculations in some particular polynomial ring without being concerned with implementing something that works for arbitrary domains. In that case you can construct the ring more directly with the *ring()* (page 2551) function:

```
>>> from sympy import ring
>>> K, xr, yr = ring([x, y], ZZ)
>>> K
```

(continues on next page)

(continued from previous page)

```
Polynomial ring in x, y over ZZ with lex order
>>> xr**2 - yr**2
x**2 - y**2
>>> (xr**2 - yr**2) // (xr - yr)
x + y
```

The object K here represents the ring and is an instance of *PolyRing* (page 2553) but is not a **polys domain** (it is not an instance of a subclass of *Domain* (page 2504) so it can not be used with *Poly* (page 2378)). In this way the implementation of polynomial rings that is used in the domain system can be used independently of the domain system.

The purpose of the domain system is to provide a unified interface for working with and converting between different representations of expressions. To make the *PolyRing* (page 2553) implementation usable in that context the *PolynomialRing* (page 2547) class is a wrapper around the *PolyRing* (page 2553) class that provides the interface expected in the domain system. That makes this implementation of polynomial rings usable as part of the broader codebase that is designed to work with expressions from different domains. The domain for polynomial rings is a distinct object from the ring returned by *ring()* (page 2551) although both have the same elements:

```
>>> K, xr, yr = ring([x, y], ZZ)
>>> K
Polynomial ring in x, y over ZZ with lex order
>>> K2 = ZZ[x,y]
>>> K2
ZZ[x,y]
>>> K2.ring
Polynomial ring in x, y over ZZ with lex order
>>> K2.ring == K
True
>>> K(x+y)
x + y
>>> K2(x+y)
x + y
>>> type(K(x+y))
<class 'sympy.polys.rings.PolyElement'>
>>> type(K2(x+y))
<class 'sympy.polys.rings.PolyElement'>
>>> K(x+y) == K2(x+y)
True
```

Rational function fields

Some domains are classified as fields and others are not. The principal difference between a field and a non-field domain is that in a field it is always possible to divide any element by any nonzero element. It is usually possible to convert any domain to a field that contains that domain with the *get_field()* (page 2514) method:

```
>>> from sympy import ZZ, QQ, symbols
>>> x, y = symbols('x, y')
>>> ZZ.is_Field
```

(continues on next page)

(continued from previous page)

```
False
>>> QQ.is_Field
True
>>> QQ[x]
QQ[x]
>>> QQ[x].is_Field
False
>>> QQ[x].get_field()
QQ(x)
>>> QQ[x].get_field().is_Field
True
>>> QQ.frac_field(x)
QQ(x)
```

This introduces a new kind of domain *K(x)* (page 2548) representing a rational function field in the generator `x` over another domain `K`. It is not possible to construct the domain `QQ(x)` with the `()` syntax so the easiest ways to create it are using the domain methods *frac_field()* (page 2512) (`QQ.frac_field(x)`) or *get_field()* (page 2514) (`QQ[x].get_field()`). The *frac_field()* (page 2512) method is the more direct approach.

The rational function field *K(x)* (page 2548) is an instance of *RationalField* (page 2530). This domain represents functions of the form $p(x)/q(x)$ for polynomials p and q. The domain elements are represented as a pair of polynomials in *K[x]* (page 2547):

```
>>> K = QQ.frac_field(x)
>>> xk = K(x)
>>> f = xk / (K.one + xk**2)
>>> f
x/(x**2 + 1)
>>> f.numer
x
>>> f.denom
x**2 + 1
>>> QQ[x].of_type(f.numer)
True
>>> QQ[x].of_type(f.denom)
True
```

Cancellation between the numerator and denominator is automatic in this field:

```
>>> p1 = xk**2 - 1
>>> p2 = xk - 1
>>> p1
x**2 - 1
>>> p2
x - 1
>>> p1 / p2
x + 1
```

Computing this cancellation can be slow which makes rational function fields potentially slower than polynomial rings or algebraic fields.

Just like in the case of polynomial rings there is both a new (sparse) and old (dense) version of fraction fields:

```
>>> K1 = QQ.frac_field(x)
>>> K2 = QQ.old_frac_field(x)
>>> K1
QQ(x)
>>> K2
QQ(x)
>>> type(K1)
<class 'sympy.polys.domains.fractionfield.FractionField'>
>>> type(K2)
<class 'sympy.polys.domains.old_fractionfield.FractionField'>
```

Also just like in the case of polynomials rings the implementation of rational function fields can be used independently of the domain system:

```
>>> from sympy import field
>>> K, xf, yf = field([x, y], ZZ)
>>> xf / (1 - yf)
-x/(y - 1)
```

Here K is an instance of *FracField* (page 2561) rather than *RationalField* (page 2530) as it would be for the domain ZZ(x,y).

Expression domain

The final domain to consider is the "expression domain" which is known as *EX* (page 2549). Expressions that can not be represented using the other domains can be always represented using the expression domain. An element of *EX* (page 2549) is actually just a wrapper around a *Expr* (page 947) instance:

```
>>> from sympy import EX
>>> p = EX.from_sympy(1 + x)
>>> p
EX(x + 1)
>>> type(p)
<class 'sympy.polys.domains.expressiondomain.ExpressionDomain.Expression'>
>>> p.ex
x + 1
>>> type(p.ex)
<class 'sympy.core.add.Add'>
```

For other domains the domain representation of expressions is usually more efficient than the tree representation used by *Expr* (page 947). In *EX* (page 2549) the internal representation is *Expr* (page 947) so it is clearly not more efficient. The purpose of the *EX* (page 2549) domain is to be able to wrap up arbitrary expressions in an interface that is consistent with the other domains. The *EX* (page 2549) domain is used as a fallback when an appropriate domain can not be found. Although this does not offer any particular efficiency it does allow the algorithms that are implemented to work over arbitrary domains to be usable when working with expressions that do not have an appropriate domain representation.

Choosing a domain

In the workflow described above the idea is to start with some sympy expressions, choose a domain and convert all the expressions into that domain in order to perform some calculation. The obvious question that arises is how to choose an appropriate domain to represent some sympy expressions. For this there is a function *construct_domain()* (page 2427) which takes a list of expressions and will choose a domain and convert all of the expressions to that domain:

```
>>> from sympy import construct_domain, Integer
>>> elements_sympy = [Integer(3), Integer(2)]  # elements as Expr instances
>>> elements_sympy
[3, 2]
>>> K, elements_K = construct_domain(elements_sympy)
>>> K
ZZ
>>> elements_K
[3, 2]
>>> type(elements_sympy[0])
<class 'sympy.core.numbers.Integer'>
>>> type(elements_K[0])
<class 'int'>
```

In this example we see that the two integers 3 and 2 can be represented in the domain *ZZ* (page 2525). The expressions have been converted to elements of that domain which in this case means the `int` type rather than instances of *Expr* (page 947). It is not necessary to explicitly create *Expr* (page 947) instances when the inputs can be sympified so e.g. `construct_domain([3, 2])` would give the same output as above.

Given more complicated inputs *construct_domain()* (page 2427) will choose more complicated domains:

```
>>> from sympy import Rational, symbols
>>> x, y = symbols('x, y')
>>> construct_domain([Rational(1, 2), Integer(3)])[0]
QQ
>>> construct_domain([2*x, 3])[0]
ZZ[x]
>>> construct_domain([x/2, 3])[0]
QQ[x]
>>> construct_domain([2/x, 3])[0]
ZZ(x)
>>> construct_domain([x, y])[0]
ZZ[x,y]
```

If any noninteger rational numbers are found in the inputs then the ground domain will be *QQ* (page 2529) rather than *ZZ* (page 2525). If any symbol is found in the inputs then a *PolynomialRing* (page 2547) will be created. A multivariate polynomial ring such as `QQ[x, y]` can also be created if there are multiple symbols in the inputs. If any symbols appear in the denominators then a *RationalField* (page 2530) like `QQ(x)` will be created instead.

Some of the domains above are fields and others are (non-field) rings. In some contexts it is necessary to have a field domain so that division is possible and for this *construct_domain()* (page 2427) has an option `field=True` which will force the construction of a field domain even if the expressions can all be represented in a non-field ring:

```
>>> construct_domain([1, 2], field=True)[0]
QQ
>>> construct_domain([2*x, 3], field=True)[0]
ZZ(x)
>>> construct_domain([x/2, 3], field=True)[0]
ZZ(x)
>>> construct_domain([2/x, 3], field=True)[0]
ZZ(x)
>>> construct_domain([x, y], field=True)[0]
ZZ(x,y)
```

By default *construct_domain()* (page 2427) will not construct an algebraic extension field and will instead use the *EX* (page 2549) domain (*ExpressionDomain* (page 2549)). The keyword argument `extension=True` can be used to construct an *AlgebraicField* (page 2539) if the inputs are irrational but algebraic:

```
>>> from sympy import sqrt
>>> construct_domain([sqrt(2)])[0]
EX
>>> construct_domain([sqrt(2)], extension=True)[0]
QQ<sqrt(2)>
>>> construct_domain([sqrt(2), sqrt(3)], extension=True)[0]
QQ<sqrt(2) + sqrt(3)>
```

When there are algebraically independent transcendentals in the inputs a *PolynomialRing* (page 2547) or *RationalField* (page 2530) will be constructed treating those transcendentals as generators:

```
>>> from sympy import sin, cos
>>> construct_domain([sin(x), y])[0]
ZZ[y,sin(x)]
```

However if there is a possibility that the inputs are not algebraically independent then the domain will be *EX* (page 2549):

```
>>> construct_domain([sin(x), cos(x)])[0]
EX
```

Here `sin(x)` and `cos(x)` are not algebraically independent since `sin(x)**2 + cos(x)**2 = 1`.

Converting elements between different domains

It is often useful to combine calculations performed over different domains. However just as it is important to avoid mixing domain elements with normal sympy expressions and other Python types it is also important to avoid mixing elements from different domains. The *convert_from()* (page 2508) method is used to convert elements from one domain into elements of another domain:

```
>>> num_zz = ZZ(3)
>>> ZZ.of_type(num_zz)
True
```

(continues on next page)

(continued from previous page)

```
>>> num_qq = QQ.convert_from(num_zz, ZZ)
>>> ZZ.of_type(num_qq)
False
>>> QQ.of_type(num_qq)
True
```

The *convert()* (page 2508) method can be called without specifying the source domain as the second argument e.g.:

```
>>> QQ.convert(ZZ(2))
2
```

This works because *convert()* (page 2508) can check the type of `ZZ(2)` and can try to work out what domain (*ZZ* (page 2525)) it is an element of. Certain domains like *ZZ* (page 2525) and *QQ* (page 2529) are treated as special cases to make this work. Elements of more complicated domains are instances of subclasses of *DomainElement* (page 2519) which has a *parent()* (page 2519) method that can identify the domain that the element belongs to. For example in the polynomial ring `ZZ[x]` we have:

```
>>> from sympy import ZZ, Symbol
>>> x = Symbol('x')
>>> K = ZZ[x]
>>> K
ZZ[x]
>>> p = K(x) + K.one
>>> p
x + 1
>>> type(p)
<class 'sympy.polys.rings.PolyElement'>
>>> p.parent()
ZZ[x]
>>> p.parent() == K
True
```

It is more efficient though to call *convert_from()* (page 2508) with the source domain specified as the second argument:

```
>>> QQ.convert_from(ZZ(2), ZZ)
2
```

Unifying domains

When we want to combine elements from two different domains and perform mixed calculations with them we need to

1. Choose a new domain that can represent all elements of both.
2. Convert all elements to the new domain.
3. Perform the calculation in the new domain.

The key question arising from point 1. is how to choose a domain that can represent the elements of both domains. For this there is the *unify()* (page 2519) method:

```
>>> x1, K1 = ZZ(2), ZZ
>>> y2, K2 = QQ(3, 2), QQ
>>> K1
ZZ
>>> K2
QQ
>>> K3 = K1.unify(K2)
>>> K3
QQ
>>> x3 = K3.convert_from(x1, K1)
>>> y3 = K3.convert_from(y2, K2)
>>> x3 + y3
7/2
```

The *unify()* (page 2519) method will find a domain that encompasses both domains so in this example `ZZ.unify(QQ)` gives *QQ* (page 2529) because every element of *ZZ* (page 2525) can be represented as an element of *QQ* (page 2529). This means that all inputs (`x1` and `y2`) can be converted to the elements of the common domain `K3` (as `x3` and `y3`). Once in the common domain we can safely use arithmetic operations like `+`. In this example one domain is a superset of the other and we see that `K1.unify(K2) == K2` so it is not actually necessary to convert `y2`. In general though `K1.unify(K2)` can give a new domain that is not equal to either `K1` or `K2`.

The *unify()* (page 2519) method understands how to combine different polynomial ring domains and how to unify the base domain:

```
>>> ZZ[x].unify(ZZ[y])
ZZ[x,y]
>>> ZZ[x,y].unify(ZZ[y])
ZZ[x,y]
>>> ZZ[x].unify(QQ)
QQ[x]
```

It is also possible to unify algebraic fields and rational function fields as well:

```
>>> K1 = QQ.algebraic_field(sqrt(2))[x]
>>> K2 = QQ.algebraic_field(sqrt(3))[y]
>>> K1
QQ<sqrt(2)>[x]
>>> K2
QQ<sqrt(3)>[y]
>>> K1.unify(K2)
QQ<sqrt(2) + sqrt(3)>[x,y]
>>> QQ.frac_field(x).unify(ZZ[y])
ZZ(x,y)
```

Internals of a Poly

We are now in a position to understand how the *Poly* (page 2378) class works internally. This is the public interface of *Poly* (page 2378):

```
>>> from sympy import Poly, symbols, ZZ
>>> x, y, z, t = symbols('x, y, z, t')
>>> p = Poly(x**2 + 1, x, domain=ZZ)
>>> p
Poly(x**2 + 1, x, domain='ZZ')
>>> p.gens
(x,)
>>> p.domain
ZZ
>>> p.all_coeffs()
[1, 0, 1]
>>> p.as_expr()
x**2 + 1
```

This is the internal implementation of *Poly* (page 2378):

```
>>> d = p.rep  # internal representation of Poly
>>> d
DMP([1, 0, 1], ZZ, None)
>>> d.rep      # internal representation of DMP
[1, 0, 1]
>>> type(d.rep)
<class 'list'>
>>> type(d.rep[0])
<class 'int'>
>>> d.dom
ZZ
```

The internal representation of a *Poly* (page 2378) instance is an instance of *DMP* (page 2561) which is the class used for domain elements in the old polynomial ring domain *old_poly_ring()* (page 2516). This represents the polynomial as a list of coefficients which are themselves elements of a domain and keeps a reference to their domain (*ZZ* (page 2525) in this example).

Choosing a domain for a Poly

If the domain is not specified for the *Poly* (page 2378) constructor then it is inferred using *construct_domain()* (page 2427). Arguments like `field=True` are passed along to *construct_domain()* (page 2427):

```
>>> from sympy import sqrt
>>> Poly(x**2 + 1, x)
Poly(x**2 + 1, x, domain='ZZ')
>>> Poly(x**2 + 1, x, field=True)
Poly(x**2 + 1, x, domain='QQ')
>>> Poly(x**2/2 + 1, x)
Poly(1/2*x**2 + 1, x, domain='QQ')
```

(continues on next page)

(continued from previous page)

```
>>> Poly(x**2 + sqrt(2), x)
Poly(x**2 + sqrt(2), x, domain='EX')
>>> Poly(x**2 + sqrt(2), x, extension=True)
Poly(x**2 + sqrt(2), x, domain='QQ<sqrt(2)>')
```

It is also possible to use the extension argument to specify generators of an extension even if no extension is required to represent the coefficients although this does not work when using *construct_domain()* (page 2427) directly. A list of extension elements will be passed to *primitive_element()* (page 2712) to create an appropriate *AlgebraicField* (page 2539) domain:

```
>>> from sympy import construct_domain
>>> Poly(x**2 + 1, x)
Poly(x**2 + 1, x, domain='ZZ')
>>> Poly(x**2 + 1, x, extension=sqrt(2))
Poly(x**2 + 1, x, domain='QQ<sqrt(2)>')
>>> Poly(x**2 + 1, x, extension=[sqrt(2), sqrt(3)])
Poly(x**2 + 1, x, domain='QQ<sqrt(2) + sqrt(3)>')
>>> construct_domain([1, 0, 1], extension=sqrt(2))[0]
ZZ
```

(Perhaps *construct_domain()* (page 2427) should do the same as *Poly* (page 2378) here...)

Choosing generators

If there are symbols other than the generators then a polynomial ring or rational function field domain will be created. The domain used for the coefficients in this case is the sparse ("new") polynomial ring:

```
>>> p = Poly(x**2*y + z, x)
>>> p
Poly(y*x**2 + z, x, domain='ZZ[y,z]')
>>> p.gens
(x,)
>>> p.domain
ZZ[y,z]
>>> p.domain == ZZ[y,z]
True
>>> p.domain == ZZ.poly_ring(y, z)
True
>>> p.domain == ZZ.old_poly_ring(y, z)
False
>>> p.rep.rep
[y, 0, z]
>>> p.rep.rep[0]
y
>>> type(p.rep.rep[0])
<class 'sympy.polys.rings.PolyElement'>
>>> dict(p.rep.rep[0])
{(1, 0): 1}
```

What we have here is a strange hybrid of dense and sparse implementations. The *Poly*

(page 2378) instance considers itself to be an univariate polynomial in the generator x but with coefficients from the domain ZZ[y,z]. The internal representation of the *Poly* (page 2378) is a list of coefficients in the "dense univariate polynomial" (DUP) format. However each coefficient is implemented as a sparse polynomial in y and z.

If we make x, y and z all be generators for the *Poly* (page 2378) then we get a fully dense DMP list of lists of lists representation:

```
>>> p = Poly(x**2*y + z, x, y, z)
>>> p
Poly(x**2*y + z, x, y, z, domain='ZZ')
>>> p.rep
DMP([[[1], []], [[]], [[1, 0]]], ZZ, None)
>>> p.rep.rep
[[[1], []], [[]], [[1, 0]]]
>>> p.rep.rep[0][0][0]
1
>>> type(p.rep.rep[0][0][0])
<class 'int'>
```

On the other hand we can make a *Poly* (page 2378) with a fully sparse representation by choosing a generator that is not in the expression at all:

```
>>> p = Poly(x**2*y + z, t)
>>> p
Poly(x**2*y + z, t, domain='ZZ[x,y,z]')
>>> p.rep
DMP([x**2*y + z], ZZ[x,y,z], None)
>>> p.rep.rep[0]
x**2*y + z
>>> type(p.rep.rep[0])
<class 'sympy.polys.rings.PolyElement'>
>>> dict(p.rep.rep[0])
{(0, 0, 1): 1, (2, 1, 0): 1}
```

If no generators are provided to the *Poly* (page 2378) constructor then it will attempt to choose generators so that the expression is polynomial in those. In the common case that the expression is a polynomial expression in some symbols then those symbols will be taken as generators. However other non-symbol expressions can also be taken as generators:

```
>>> Poly(x**2*y + z)
Poly(x**2*y + z, x, y, z, domain='ZZ')
>>> from sympy import pi, exp
>>> Poly(exp(x) + exp(2*x) + 1)
Poly((exp(x))**2 + (exp(x)) + 1, exp(x), domain='ZZ')
>>> Poly(pi*x)
Poly(x*pi, x, pi, domain='ZZ')
>>> Poly(pi*x, x)
Poly(pi*x, x, domain='ZZ[pi]')
```

Algebraically dependent generators

Taking `exp(x)` or `pi` as generators for a *Poly* (page 2378) or for its polynomial ring domain is mathematically valid because these objects are transcendental and so the ring extension containing them is isomorphic to a polynomial ring. Since `x` and `exp(x)` are algebraically independent it is also valid to use both as generators for the same *Poly* (page 2378). However some other combinations of generators are invalid such as `x` and `sqrt(x)` or `sin(x)` and `cos(x)`. These examples are invalid because the generators are not algebraically independent (e.g. `sqrt(x)**2 = x` and `sin(x)**2 + cos(x)**2 = 1`). The implementation is not able to detect these algebraic relationships though:

```
>>> from sympy import sin, cos, sqrt
>>> Poly(x*exp(x))      # fine
Poly(x*(exp(x)), x, exp(x), domain='ZZ')
>>> Poly(sin(x)+cos(x)) # not fine
Poly((cos(x)) + (sin(x)), cos(x), sin(x), domain='ZZ')
>>> Poly(x + sqrt(x))   # not fine
Poly(x + (sqrt(x)), x, sqrt(x), domain='ZZ')
```

Calculations with a *Poly* (page 2378) such as this are unreliable because zero-testing will not work properly in this implementation:

```
>>> p1 = Poly(x, x, sqrt(x))
>>> p2 = Poly(sqrt(x), x, sqrt(x))
>>> p1
Poly(x, x, sqrt(x), domain='ZZ')
>>> p2
Poly((sqrt(x)), x, sqrt(x), domain='ZZ')
>>> p3 = p1 - p2**2
>>> p3                  # should be zero...
Poly(x - (sqrt(x))**2, x, sqrt(x), domain='ZZ')
>>> p3.as_expr()
0
```

This aspect of *Poly* (page 2378) could be improved by:

1. Expanding the domain system with new domains that can represent more classes of algebraic extension.
2. Improving the detection of algebraic dependencies in *construct_domain()* (page 2427).
3. Improving the automatic selection of generators.

Examples of the above are that it would be useful to have a domain that can represent more general algebraic extensions (*AlgebraicField* (page 2539) is only for extensions of *QQ* (page 2529)). Improving the detection of algebraic dependencies is harder but at least common cases like `sin(x)` and `cos(x)` could be handled. When choosing generators it should be possible to recognise that `sqrt(x)` can be the only generator for `x + sqrt(x)`:

```
>>> Poly(x + sqrt(x))           # this could be improved!
Poly(x + (sqrt(x)), x, sqrt(x), domain='ZZ')
>>> Poly(x + sqrt(x), sqrt(x))   # this could be improved!
Poly((sqrt(x)) + x, sqrt(x), domain='ZZ[x]')
```

Reference docs for the Poly Domains

This page lists the reference documentation for the domains in the polys module. For a general introduction to the polys module it is recommended to read *Basic functionality of the module* (page 2342) instead. For an introductory explanation of the what the domain system is and how it is used it is recommended to read *Introducing the Domains of the poly module* (page 2477). This page lists the reference docs for the *Domain* (page 2504) class and its subclasses (the specific domains such as ZZ) as well as the classes that represent the domain elements.

Domains

Here we document the various implemented ground domains (see *Introducing the Domains of the poly module* (page 2477) for more of an explanation). There are three types of *Domain* (page 2504) subclass: abstract domains, concrete domains, and "implementation domains". Abstract domains cannot be (usefully) instantiated at all, and just collect together functionality shared by many other domains. Concrete domains are those meant to be instantiated and used in the polynomial manipulation algorithms. In some cases, there are various possible ways to implement the data type the domain provides. For example, depending on what libraries are available on the system, the integers are implemented either using the python built-in integers, or using gmpy. Note that various aliases are created automatically depending on the libraries available. As such e.g. ZZ always refers to the most efficient implementation of the integer ring available.

Abstract Domains

class sympy.polys.domains.domain.**Domain**

Superclass for all domains in the polys domains system.

See *Introducing the Domains of the poly module* (page 2477) for an introductory explanation of the domains system.

The *Domain* (page 2504) class is an abstract base class for all of the concrete domain types. There are many different *Domain* (page 2504) subclasses each of which has an associated `dtype` which is a class representing the elements of the domain. The coefficients of a *Poly* (page 2378) are elements of a domain which must be a subclass of *Domain* (page 2504).

Examples

The most common example domains are the integers *ZZ* (page 2525) and the rationals *QQ* (page 2529).

```
>>> from sympy import Poly, symbols, Domain
>>> x, y = symbols('x, y')
>>> p = Poly(x**2 + y)
>>> p
Poly(x**2 + y, x, y, domain='ZZ')
>>> p.domain
ZZ
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(p.domain, Domain)
True
>>> Poly(x**2 + y/2)
Poly(x**2 + 1/2*y, x, y, domain='QQ')
```

The domains can be used directly in which case the domain object e.g. (*ZZ* (page 2525) or *QQ* (page 2529)) can be used as a constructor for elements of `dtype`.

```
>>> from sympy import ZZ, QQ
>>> ZZ(2)
2
>>> ZZ.dtype
<class 'int'>
>>> type(ZZ(2))
<class 'int'>
>>> QQ(1, 2)
1/2
>>> type(QQ(1, 2))
<class 'sympy.polys.domains.pythonrational.PythonRational'>
```

The corresponding domain elements can be used with the arithmetic operations `+,-,*,**` and depending on the domain some combination of `/,//,%` might be usable. For example in *ZZ* (page 2525) both `//` (floor division) and `%` (modulo division) can be used but `/` (true division) cannot. Since *QQ* (page 2529) is a *Field* (page 2520) its elements can be used with `/` but `//` and `%` should not be used. Some domains have a *gcd()* (page 2514) method.

```
>>> ZZ(2) + ZZ(3)
5
>>> ZZ(5) // ZZ(2)
2
>>> ZZ(5) % ZZ(2)
1
>>> QQ(1, 2) / QQ(2, 3)
3/4
>>> ZZ.gcd(ZZ(4), ZZ(2))
2
>>> QQ.gcd(QQ(2,7), QQ(5,3))
1/21
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
```

There are also many other domains including:

1. *GF(p)* (page 2522) for finite fields of prime order.
2. *RR* (page 2545) for real (floating point) numbers.
3. *CC* (page 2546) for complex (floating point) numbers.
4. *QQ<a>* (page 2539) for algebraic number fields.
5. *K[x]* (page 2547) for polynomial rings.

6. *K(x)* (page 2548) for rational function fields.
7. *EX* (page 2549) for arbitrary expressions.

Each domain is represented by a domain object and also an implementation class (`dtype`) for the elements of the domain. For example the *K[x]* (page 2547) domains are represented by a domain object which is an instance of *PolynomialRing* (page 2547) and the elements are always instances of *PolyElement* (page 2554). The implementation class represents particular types of mathematical expressions in a way that is more efficient than a normal SymPy expression which is of type *Expr* (page 947). The domain methods *from_sympy()* (page 2513) and *to_sympy()* (page 2517) are used to convert from *Expr* (page 947) to a domain element and vice versa.

```
>>> from sympy import Symbol, ZZ, Expr
>>> x = Symbol('x')
>>> K = ZZ[x]           # polynomial ring domain
>>> K
ZZ[x]
>>> type(K)             # class of the domain
<class 'sympy.polys.domains.polynomialring.PolynomialRing'>
>>> K.dtype             # class of the elements
<class 'sympy.polys.rings.PolyElement'>
>>> p_expr = x**2 + 1   # Expr
>>> p_expr
x**2 + 1
>>> type(p_expr)
<class 'sympy.core.add.Add'>
>>> isinstance(p_expr, Expr)
True
>>> p_domain = K.from_sympy(p_expr)
>>> p_domain            # domain element
x**2 + 1
>>> type(p_domain)
<class 'sympy.polys.rings.PolyElement'>
>>> K.to_sympy(p_domain) == p_expr
True
```

The *convert_from()* (page 2508) method is used to convert domain elements from one domain to another.

```
>>> from sympy import ZZ, QQ
>>> ez = ZZ(2)
>>> eq = QQ.convert_from(ez, ZZ)
>>> type(ez)
<class 'int'>
>>> type(eq)
<class 'sympy.polys.domains.pythonrational.PythonRational'>
```

Elements from different domains should not be mixed in arithmetic or other operations: they should be converted to a common domain first. The domain method *unify()* (page 2519) is used to find a domain that can represent all the elements of two given domains.

```
>>> from sympy import ZZ, QQ, symbols
>>> x, y = symbols('x, y')
```

(continues on next page)

(continued from previous page)

```
>>> ZZ.unify(QQ)
QQ
>>> ZZ[x].unify(QQ)
QQ[x]
>>> ZZ[x].unify(QQ[y])
QQ[x,y]
```

If a domain is a *Ring* (page 2521) then is might have an associated *Field* (page 2520) and vice versa. The *get_field()* (page 2514) and *get_ring()* (page 2514) methods will find or create the associated domain.

```
>>> from sympy import ZZ, QQ, Symbol
>>> x = Symbol('x')
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ
>>> QQ.has_assoc_Ring
True
>>> QQ.get_ring()
ZZ
>>> K = QQ[x]
>>> K
QQ[x]
>>> K.get_field()
QQ(x)
```

See also:

DomainElement **(page 2519)**
abstract base class for domain elements

construct_domain **(page 2427)**
construct a minimal domain for some expressions

abs(*a*)
Absolute value of a, implies __abs__.

add(*a*, *b*)
Sum of a and b, implies __add__.

alg_field_from_poly(*poly*, *alias=None*, *root_index=-1*)
Convenience method to construct an algebraic extension on a root of a polynomial, chosen by root index.

Parameters

poly : *Poly* (page 2378)

The polynomial whose root generates the extension.

alias : str, optional (default=None)

Symbol name for the generator of the extension. E.g. “alpha” or “theta”.

root_index : int, optional (default=-1)

Specifies which root of the polynomial is desired. The ordering is as defined by the *ComplexRootOf* (page 2431) class. The default of -1 selects the most natural choice in the common cases of quadratic and cyclotomic fields (the square root on the positive real or imaginary axis, resp. $e^{2\pi i/n}$).

Examples

```
>>> from sympy import QQ, Poly
>>> from sympy.abc import x
>>> f = Poly(x**2 - 2)
>>> K = QQ.alg_field_from_poly(f)
>>> K.ext.minpoly == f
True
>>> g = Poly(8*x**3 - 6*x - 1)
>>> L = QQ.alg_field_from_poly(g, "alpha")
>>> L.ext.minpoly == g
True
>>> L.to_sympy(L([1, 1, 1]))
alpha**2 + alpha + 1
```

algebraic_field(**extension*, *alias=None*)

Returns an algebraic field, i.e. $K(\alpha,\ldots)$.

almosteq(*a*, *b*, *tolerance=None*)

Check if a and b are almost equal.

characteristic()

Return the characteristic of this domain.

cofactors(*a*, *b*)

Returns GCD and cofactors of a and b.

convert(*element*, *base=None*)

Convert element to self.dtype.

convert_from(*element*, *base*)

Convert element to self.dtype given the base domain.

cyclotomic_field(*n*, *ss=False*, *alias='zeta'*, *gen=None*, *root_index=-1*)

Convenience method to construct a cyclotomic field.

Parameters

n : int

Construct the nth cyclotomic field.

ss : boolean, optional (default=False)

If True, append *n* as a subscript on the alias string.

alias : str, optional (default="zeta")

Symbol name for the generator.

gen : *Symbol* (page 976), optional (default=None)

Desired variable for the cyclotomic polynomial that defines the field. If `None`, a dummy variable will be used.

root_index : int, optional (default=-1)

Specifies which root of the polynomial is desired. The ordering is as defined by the *ComplexRootOf* (page 2431) class. The default of `-1` selects the root $e^{2\pi i/n}$.

Examples

```
>>> from sympy import QQ, latex
>>> K = QQ.cyclotomic_field(5)
>>> K.to_sympy(K([-1, 1]))
1 - zeta
>>> L = QQ.cyclotomic_field(7, True)
>>> a = L.to_sympy(L([-1, 1]))
>>> print(a)
1 - zeta7
>>> print(latex(a))
1 - \zeta_{7}
```

denom(*a*)

Returns denominator of `a`.

div(*a*, *b*)

Quotient and remainder for *a* and *b*. Analogue of `divmod(a, b)`

Parameters

a: domain element

The dividend

b: domain element

The divisor

Returns

(q, r): tuple of domain elements

The quotient and remainder

Raises

ZeroDivisionError: when the divisor is zero.

Explanation

This is essentially the same as `divmod(a, b)` except that is more consistent when working over some *Field* (page 2520) domains such as *QQ* (page 2529). When working over an arbitrary *Domain* (page 2504) the *div()* (page 2509) method should be used instead of `divmod`.

The key invariant is that if `q, r = K.div(a, b)` then `a == b*q + r`.

The result of `K.div(a, b)` is the same as the tuple `(K.quo(a, b), K.rem(a, b))` except that if both quotient and remainder are needed then it is more efficient to use *div()* (page 2509).

Examples

We can use `K.div` instead of `divmod` for floor division and remainder.

```
>>> from sympy import ZZ, QQ
>>> ZZ.div(ZZ(5), ZZ(2))
(2, 1)
```

If K is a *Field* (page 2520) then the division is always exact with a remainder of *zero* (page 2519).

```
>>> QQ.div(QQ(5), QQ(2))
(5/2, 0)
```

Notes

If `gmpy` is installed then the `gmpy.mpq` type will be used as the *dtype* (page 2510) for *QQ* (page 2529). The `gmpy.mpq` type defines `divmod` in a way that is undesirable so *div()* (page 2509) should be used instead of `divmod`.

```
>>> a = QQ(1)
>>> b = QQ(3, 2)
>>> a
mpq(1,1)
>>> b
mpq(3,2)
>>> divmod(a, b)
(mpz(0), mpq(1,1))
>>> QQ.div(a, b)
(mpq(2,3), mpq(0,1))
```

Using `//` or `%` with *QQ* (page 2529) will lead to incorrect results so *div()* (page 2509) should be used instead.

See also:

***quo* (page 2516)**
Analogue of `a // b`

***rem* (page 2517)**
Analogue of `a % b`

***exquo* (page 2511)**
Analogue of `a / b`

drop(**symbols*)
Drop generators from this domain.

dtype: Optional[Type] = None
The type (class) of the elements of this *Domain* (page 2504):

```
>>> from sympy import ZZ, QQ, Symbol
>>> ZZ.dtype
<class 'int'>
```

(continues on next page)

(continued from previous page)

```
>>> z = ZZ(2)
>>> z
2
>>> type(z)
<class 'int'>
>>> type(z) == ZZ.dtype
True
```

Every domain has an associated **dtype** ("datatype") which is the class of the associated domain elements.

See also:

of_type (page 2516)

evalf(*a*, *prec=None*, ***options*)

Returns numerical approximation of `a`.

exquo(*a*, *b*)

Exact quotient of *a* and *b*. Analogue of `a / b`.

Parameters

a: domain element

The dividend

b: domain element

The divisor

Returns

q: domain element

The exact quotient

Raises

ExactQuotientFailed: if exact division is not possible.

ZeroDivisionError: when the divisor is zero.

Explanation

This is essentially the same as `a / b` except that an error will be raised if the division is inexact (if there is any remainder) and the result will always be a domain element. When working in a *Domain* (page 2504) that is not a *Field* (page 2520) (e.g. *ZZ* (page 2525) or *K[x]* (page 2547)) `exquo` should be used instead of `/`.

The key invariant is that if `q = K.exquo(a, b)` (and `exquo` does not raise an exception) then `a == b*q`.

Examples

We can use `K.exquo` instead of `/` for exact division.

```
>>> from sympy import ZZ
>>> ZZ.exquo(ZZ(4), ZZ(2))
2
>>> ZZ.exquo(ZZ(5), ZZ(2))
Traceback (most recent call last):
    ...
ExactQuotientFailed: 2 does not divide 5 in ZZ
```

Over a *Field* (page 2520) such as *QQ* (page 2529), division (with nonzero divisor) is always exact so in that case `/` can be used instead of *exquo()* (page 2511).

```
>>> from sympy import QQ
>>> QQ.exquo(QQ(5), QQ(2))
5/2
>>> QQ(5) / QQ(2)
5/2
```

Notes

Since the default *dtype* (page 2510) for *ZZ* (page 2525) is `int` (or `mpz`) division as `a / b` should not be used as it would give a `float`.

```
>>> ZZ(4) / ZZ(2)
2.0
>>> ZZ(5) / ZZ(2)
2.5
```

Using `/` with *ZZ* (page 2525) will lead to incorrect results so *exquo()* (page 2511) should be used instead.

See also:

***quo* (page 2516)**
Analogue of `a // b`

***rem* (page 2517)**
Analogue of `a % b`

***div* (page 2509)**
Analogue of `divmod(a, b)`

frac_field(**symbols*, *order=LexOrder()*)
Returns a fraction field, i.e. $K(X)$.

from_AlgebraicField(*a*, *K0*)
Convert an algebraic number to `dtype`.

from_ComplexField(*a*, *K0*)
Convert a complex element to `dtype`.

from_ExpressionDomain(*a*, *K0*)
Convert a EX object to dtype.

from_ExpressionRawDomain(*a*, *K0*)
Convert a EX object to dtype.

from_FF(*a*, *K0*)
Convert ModularInteger(int) to dtype.

from_FF_gmpy(*a*, *K0*)
Convert ModularInteger(mpz) to dtype.

from_FF_python(*a*, *K0*)
Convert ModularInteger(int) to dtype.

from_FractionField(*a*, *K0*)
Convert a rational function to dtype.

from_GlobalPolynomialRing(*a*, *K0*)
Convert a polynomial to dtype.

from_MonogenicFiniteExtension(*a*, *K0*)
Convert an ExtensionElement to dtype.

from_PolynomialRing(*a*, *K0*)
Convert a polynomial to dtype.

from_QQ_gmpy(*a*, *K0*)
Convert a GMPY mpq object to dtype.

from_QQ_python(*a*, *K0*)
Convert a Python Fraction object to dtype.

from_RealField(*a*, *K0*)
Convert a real element object to dtype.

from_ZZ_gmpy(*a*, *K0*)
Convert a GMPY mpz object to dtype.

from_ZZ_python(*a*, *K0*)
Convert a Python int object to dtype.

from_sympy(*a*)
Convert a SymPy expression to an element of this domain.

Parameters
expr: Expr
A normal SymPy expression of type *Expr* (page 947).

Returns
a: domain element
An element of this *Domain* (page 2504).

Explanation

See *to_sympy()* (page 2517) for explanation and examples.

See also:

to_sympy (page 2517), *convert_from* (page 2508)

gcd(*a*, *b*)
Returns GCD of `a` and `b`.

gcdex(*a*, *b*)
Extended GCD of `a` and `b`.

get_exact()
Returns an exact domain associated with `self`.

get_field()
Returns a field associated with `self`.

get_ring()
Returns a ring associated with `self`.

half_gcdex(*a*, *b*)
Half extended GCD of `a` and `b`.

has_assoc_Field = False
Boolean flag indicating if the domain has an associated *Field* (page 2520).

```
>>> from sympy import ZZ
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ
```

See also:

is_Field (page 2514), *get_field* (page 2514)

has_assoc_Ring = False
Boolean flag indicating if the domain has an associated *Ring* (page 2521).

```
>>> from sympy import QQ
>>> QQ.has_assoc_Ring
True
>>> QQ.get_ring()
ZZ
```

See also:

is_Field (page 2514), *get_ring* (page 2514)

inject(**symbols*)
Inject generators into this domain.

invert(*a*, *b*)
Returns inversion of `a mod b`, implies something.

is_Field = False

Boolean flag indicating if the domain is a *Field* (page 2520).

```
>>> from sympy import ZZ, QQ
>>> ZZ.is_Field
False
>>> QQ.is_Field
True
```

See also:

is_PID (page 2515), *is_Ring* (page 2515), *get_field* (page 2514), *has_assoc_Field* (page 2514)

is_PID = False

Boolean flag indicating if the domain is a principal ideal domain.

```
>>> from sympy import ZZ
>>> ZZ.has_assoc_Field
True
>>> ZZ.get_field()
QQ
```

See also:

is_Field (page 2514), *get_field* (page 2514)

is_Ring = False

Boolean flag indicating if the domain is a *Ring* (page 2521).

```
>>> from sympy import ZZ
>>> ZZ.is_Ring
True
```

Basically every *Domain* (page 2504) represents a ring so this flag is not that useful.

See also:

is_PID (page 2515), *is_Field* (page 2514), *get_ring* (page 2514), *has_assoc_Ring* (page 2514)

is_negative(*a*)

Returns True if a is negative.

is_nonnegative(*a*)

Returns True if a is non-negative.

is_nonpositive(*a*)

Returns True if a is non-positive.

is_one(*a*)

Returns True if a is one.

is_positive(*a*)

Returns True if a is positive.

is_zero(*a*)

Returns True if a is zero.

lcm(*a*, *b*)
Returns LCM of `a` and `b`.

log(*a*, *b*)
Returns b-base logarithm of `a`.

map(*seq*)
Rersively apply `self` to all elements of `seq`.

mul(*a*, *b*)
Product of `a` and `b`, implies `__mul__`.

n(*a*, *prec=None*, ***options*)
Returns numerical approximation of `a`.

neg(*a*)
Returns `a` negated, implies `__neg__`.

numer(*a*)
Returns numerator of `a`.

of_type(*element*)
Check if `a` is of type `dtype`.

old_frac_field(**symbols*, ***kwargs*)
Returns a fraction field, i.e. $K(X)$.

old_poly_ring(**symbols*, ***kwargs*)
Returns a polynomial ring, i.e. $K[X]$.

one: Optional[Any] = None
The one element of the *Domain* (page 2504):

```
>>> from sympy import QQ
>>> QQ.one
1
>>> QQ.of_type(QQ.one)
True
```

See also:

of_type (page 2516), *zero* (page 2519)

poly_ring(**symbols*, *order=LexOrder()*)
Returns a polynomial ring, i.e. $K[X]$.

pos(*a*)
Returns `a` positive, implies `__pos__`.

pow(*a*, *b*)
Raise `a` to power `b`, implies `__pow__`.

quo(*a*, *b*)
Quotient of *a* and *b*. Analogue of `a // b`.

`K.quo(a, b)` is equivalent to `K.div(a, b)[0]`. See *div()* (page 2509) for more explanation.

See also: