(continued from previous page)

```
'infinite': False,
'integer': True,
'irrational': False,
'negative': False,
'noninteger': False,
'nonnegative': True,
'nonpositive': False,
'nonzero': True,
'positive': True,
'rational': True,
'real': True,
'transcendental': False,
'zero': False}
```

We can see that there are many more predicates listed than the two that were used to create `x`. This is because the assumptions system can infer some predicates from combinations of other predicates. For example if a symbol is declared with `positive=True` then it is possible to infer that it should have `negative=False` because a positive number can never be negative. Similarly if a symbol is created with `integer=True` then it is possible to infer that is should have `rational=True` because every integer is a rational number.

A full table of the possible predicates and their definitions is given below.

Table1: Assumptions predicates for the (old) assumptions

Predicate	Definition	Implications
`commutative`	A commutative expression. A `commutative` expression commutes with all other expressions under multiplication. If an expression `a` has `commutative=True` then `a * b == b * a` for any other expression `b` (even if `b` is not `commutative`). Unlike all other assumptions predicates `commutative` must always be `True` or `False` and can never be `None`. Also unlike all other predicates `commutative` defaults to `True` in e.g. `Symbol('x')`. [commutative]	
`infinite`	An infinite expression such as `oo`, `-oo` or `zoo`. [infinite]	`== !finite`
`finite`	A finite expression. Any expression that is not `infinite` is considered `finite`. [infinite]	`== !infinite`
`hermitian`	An element of the field of Hermitian operators. [antihermitian]	
`antihermitian`	An element of the field of antihermitian operators. [antihermitian]	

continues on next page

Table 1 – continued from previous page

Predicate	Definition	Implications
`complex`	A complex number, $z \in \mathbb{C}$. Any number of the form $x + iy$ where x and y are `real` and $i = \sqrt{-1}$. All `complex` numbers are `finite`. Includes all `real` numbers. [complex]	`-> commutative` `-> finite`
`algebraic`	An algebraic number, $z \in \overline{\mathbb{Q}}$. Any number that is a root of a non-zero polynomial $p(z) \in \mathbb{Q}[z]$ having rational coefficients. All `algebraic` numbers are `complex`. An `algebraic` number may or may not be `real`. Includes all `rational` numbers. [algebraic]	`-> complex`
`transcendental`	A complex number that is not algebraic, $z \in \mathbb{C} - \overline{\mathbb{Q}}$. All `transcendental` numbers are `complex`. A `transcendental` number may or may not be `real` but can never be `rational`. [transcendental]	`== (complex & !algebraic)`
`extended_real`	An element of the extended real number line, $x \in \overline{\mathbb{R}}$ where $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$. An `extended_real` number is either `real` or $\pm\infty$. The relational operators `<`, `<=`, `>=` and `>` are defined only for expressions that are `extended_real`. [extended_real]	`-> commutative`

`real`	A real number, $x \in \mathbb{R}$. All `real` numbers are `finite` and `complex` (the set of reals is a subset of the set of complex numbers). Includes all `rational` numbers. A `real` number is either `negative`, `zero` or `positive`. [real]	`-> complex`
		`== (extended_real & finite)`
		`== (negative | zero | positive)`
		`-> hermitian`

`imaginary`	An imaginary number, $z \in \mathbb{I} - \{0\}$. A number of the form $z = yi$ where y is real, $y \neq 0$ and $i = \sqrt{-1}$. All `imaginary` numbers are `complex` and not `real`. Note in particular that `zero` is *not* considered `imaginary` in SymPy. [imaginary]	`-> complex`
		`-> antihermitian`
		`-> !extended_real`

`rational`	A rational number, $q \in \mathbb{Q}$. Any number of the form $\frac{a}{b}$ where a and b are integers and $b \neq 0$. All `rational` numbers are `real` and `algebraic`. Includes all `integer` numbers. [rational]	`-> real` `-> algebraic`
`irrational`	A real number that is not rational, $x \in \mathbb{R} - \mathbb{Q}$. [irrational]	`== (real & !rational)`

continues on next page

Table 1 – continued from previous page

Predicate	Definition	Implications
integer	An integer, $a \in \mathbb{Z}$. All integers are rational. Includes zero and all prime, composite, even and odd numbers. [integer]	-> rational
noninteger	An extended real number that is not an integer, $x \in \overline{\mathbb{R}} - \mathbb{Z}$.	== (extended_real & !integer)
even	An even number, $e \in \{2k : k \in \mathbb{Z}\}$. All even numbers are integer numbers. Includes zero. [parity]	-> integer -> !odd
odd	An odd number, $o \in \{2k + 1 : k \in \mathbb{Z}\}$. All odd numbers are integer numbers. [parity]	-> integer -> !even
prime	A prime number, $p \in \mathbb{P}$. All prime numbers are positive and integer. [prime]	-> integer -> positive
composite	A composite number, $c \in \mathbb{N} - (\mathbb{P} \cup \{1\})$. A positive integer that is the product of two or more primes. A composite number is always a positive integer and is not prime. [composite]	-> (integer & positive & !prime) !composite -> (!positive | !even | prime)
zero	The number 0. An expression with zero=True represents the number 0 which is an integer. [zero]	-> even & finite == (extended_nonnegative & extended_nonpositive) == (nonnegative & nonpositive)
nonzero	A nonzero real number, $x \in \mathbb{R} - \{0\}$. A nonzero number is always real and can not be zero.	-> real == (extended_nonzero & finite)
extended_nonzero	A member of the extended reals that is not zero, $x \in \overline{\mathbb{R}} - \{0\}$.	== (extended_real & !zero)
positive	A positive real number, $x \in \mathbb{R}, x > 0$. All positive numbers are finite so oo is not positive. [positive]	== (nonnegative & nonzero) == (extended_positive & finite)

continues on next page

Table 1 – continued from previous page

Predicate	Definition	Implications
nonnegative	A nonnegative real number, $x \in \mathbb{R}, x \geq 0$. All nonnegative numbers are finite so oo is not nonnegative. [positive]	== (real & !negative) == (extended_nonnegative & finite)
negative	A negative real number, $x \in \mathbb{R}, x < 0$. All negative numbers are finite so -oo is not negative. [negative]	== (nonpositive & nonzero) == (extended_negative & finite)
nonpositive	A nonpositive real number, $x \in \mathbb{R}, x \leq 0$. All nonpositive numbers are finite so -oo is not nonpositive. [negative]	== (real & !positive) == (extended_nonpositive & finite)
extended_positive	A positive extended real number, $x \in \overline{\mathbb{R}}, x > 0$. An extended_positive number is either positive or oo. [extended_real]	== (extended_nonnegative & extended_nonzero)
extended_nonnegative	A nonnegative extended real number, $x \in \overline{\mathbb{R}}, x \geq 0$. An extended_nonnegative number is either nonnegative or oo. [extended_real]	== (extended_real & !extended_negative)
extended_negative	A negative extended real number, $x \in \overline{\mathbb{R}}, x < 0$. An extended_negative number is either negative or -oo. [extended_real]	== (extended_nonpositive & extended_nonzero)
extended_nonpositive	A nonpositive extended real number, $x \in \overline{\mathbb{R}}, x \leq 0$. An extended_nonpositive number is either nonpositive or -oo. [extended_real]	== (extended_real & !extended_positive)

References for the above definitions

3.1.5 Implications

The assumptions system uses the inference rules to infer new predicates beyond those immediately specified when creating a symbol:

```
>>> x = Symbol('x', real=True, negative=False, zero=False)
>>> x.is_positive
True
```

Although x was not explicitly declared positive it can be inferred from the predicates that were given explicitly. Specifically one of the inference rules is real == negative | zero | positive so if real is True and both negative and zero are False then positive must be True.

In practice the assumption inference rules mean that it is not necessary to include redundant

predicates for example a positive real number can be simply be declared as positive:

```
>>> x1 = Symbol('x1', positive=True, real=True)
>>> x2 = Symbol('x2', positive=True)
>>> x1.is_real
True
>>> x2.is_real
True
>>> x1.assumptions0 == x2.assumptions0
True
```

Combining predicates that are inconsistent will give an error:

```
>>> x = Symbol('x', commutative=False, real=True)
Traceback (most recent call last):
...
InconsistentAssumptions: {
      algebraic: False,
      commutative: False,
      complex: False,
      composite: False,
      even: False,
      extended_negative: False,
      extended_nonnegative: False,
      extended_nonpositive: False,
      extended_nonzero: False,
      extended_positive: False,
      extended_real: False,
      imaginary: False,
      integer: False,
      irrational: False,
      negative: False,
      noninteger: False,
      nonnegative: False,
      nonpositive: False,
      nonzero: False,
      odd: False,
      positive: False,
      prime: False,
      rational: False,
      real: False,
      transcendental: False,
      zero: False}, real=True
```

3.1.6 Interpretation of the predicates

Although the predicates are defined in the table above it is worth taking some time to think about how to interpret them. Firstly many of the concepts referred to by the predicate names like “zero”, “prime”, “rational” etc have a basic meaning in mathematics but can also have more general meanings. For example when dealing with matrices a matrix of all zeros might be referred to as “zero”. The predicates in the assumptions system do not allow any generalizations such as this. The predicate `zero` is strictly reserved for the plain number 0. Instead matrices have an *is_zero_matrix()* (page 1344) property for this purpose (although that property is not strictly part of the assumptions system):

```
>>> from sympy import Matrix
>>> M = Matrix([[0, 0], [0, 0]])
>>> M.is_zero
False
>>> M.is_zero_matrix
True
```

Similarly there are generalisations of the integers such as the Gaussian integers which have a different notion of prime number. The `prime` predicate in the assumptions system does not include those and strictly refers only to the standard prime numbers $\mathbb{P} = \{2, 3, 5, 7, 11, \cdots\}$. Likewise `integer` only means the standard concept of the integers $\mathbb{Z} = \{0, \pm1, \pm2, \cdots\}$, `rational` only means the standard concept of the rational numbers $\mathbb{Q}$ and so on.

The predicates set up schemes of subsets such as the chain beginning with the complex numbers which are considered as a superset of the reals which are in turn a superset of the rationals and so on. The chain of subsets

$$\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

corresponds to the chain of implications in the assumptions system

```
integer -> rational -> real -> complex
```

A “vanilla” symbol with no assumptions explicitly attached is not known to belong to any of these sets and is not even known to be finite:

```
>>> x = Symbol('x')
>>> x.assumptions0
{'commutative': True}
>>> print(x.is_commutative)
True
>>> print(x.is_rational)
None
>>> print(x.is_complex)
None
>>> print(x.is_real)
None
>>> print(x.is_integer)
None
>>> print(x.is_finite)
None
```

It is hard for SymPy to know what it can do with such a symbol that is not even known to be finite or complex so it is generally better to give some assumptions to the symbol explicitly.

Many parts of SymPy will implicitly treat such a symbol as complex and in some cases SymPy will permit manipulations that would not strictly be valid given that x is not known to be finite. In a formal sense though very little is known about a vanilla symbol which makes manipulations involving it difficult.

Defining *something* about a symbol can make a big difference. For example if we declare the symbol to be an integer then this implies a suite of other predicates that will help in further manipulations:

```
>>> n = Symbol('n', integer=True)
>>> n.assumptions0
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
 'infinite': False,
 'integer': True,
 'irrational': False,
 'noninteger': False,
 'rational': True,
 'real': True,
 'transcendental': False}
```

These assumptions can lead to very significant simplifications e.g. `integer=True` gives:

```
>>> from sympy import sin, pi
>>> n1 = Symbol('n1')
>>> n2 = Symbol('n2', integer=True)
>>> sin(n1 * pi)
sin(pi*n1)
>>> sin(n2 * pi)
0
```

Replacing a whole expression with 0 is about as good as simplification can get!

It is normally advisable to set as many assumptions as possible on any symbols so that expressions can be simplified as much as possible. A common misunderstanding leads to defining a symbol with a `False` predicate e.g.:

```
>>> x = Symbol('x', negative=False)
>>> print(x.is_negative)
False
>>> print(x.is_nonnegative)
None
>>> print(x.is_real)
None
>>> print(x.is_complex)
None
>>> print(x.is_finite)
None
```

If the intention is to say that x is a real number that is not positive then that needs to be explicitly stated. In the context that the symbol is known to be real, the predicate `positive=False`

becomes much more meaningful:

```
>>> x = Symbol('x', real=True, negative=False)
>>> print(x.is_negative)
False
>>> print(x.is_nonnegative)
True
>>> print(x.is_real)
True
>>> print(x.is_complex)
True
>>> print(x.is_finite)
True
```

A symbol declared as `Symbol('x', real=True, negative=False)` is equivalent to a symbol declared as `Symbol('x', nonnegative=True)`. Simply declaring a symbol as `Symbol('x', positive=False)` does not allow the assumptions system to conclude much about it because a vanilla symbol is not known to be finite or even complex.

A related confusion arises with `Symbol('x', complex=True)` and `Symbol('x', real=False)`. Often when either of these is used neither is what is actually wanted. The first thing to understand is that all real numbers are complex so a symbol created with `real=True` will also have `complex=True` and a symbol created with `complex=True` will not have `real=False`. If the intention was to create a complex number that is not a real number then it should be `Symbol('x', complex=True, real=False)`. On the other hand declaring `real=False` alone is not sufficient to conclude that `complex=True` because knowing that it is not a real number does not tell us whether it is finite or whether or not it is some completely different kind of object from a complex number.

A vanilla symbol is defined by not knowing whether it is `finite` etc but there is no clear definition of what it *should* actually represent. It is tempting to think of it as an "arbitrary complex number or possibly one of the infinities" but there is no way to query an arbitrary (non-symbol) expression in order to determine if it meets those criteria. It is important to bear in mind that within the SymPy codebase and potentially in downstream libraries many other kinds of mathematical objects can be found that might also have `commutative=True` while being something very different from an ordinary number (in this context even SymPy's standard infinities are considered "ordinary").

The only predicate that is applied by default for a symbol is `commutative`. We can also declare a symbol to be *noncommutative* e.g.:

```
>>> x, y = symbols('x, y', commutative=False)
>>> z = Symbol('z')  # defaults to commutative=True
>>> x*y + y*x
x*y + y*x
>>> x*z + z*x
2*z*x
```

Note here that since `x` and `y` are both noncommutative `x` and `y` do not commute so `x*y != y*x`. On the other hand since `z` is commutative `x` and `z` commute and `x*z == z*x` even though `x` is noncommutative.

The interpretation of what a vanilla symbol represents is unclear but the interpretation of an expression with `commutative=False` is entirely obscure. Such an expression is necessarily not a complex number or an extended real or any of the standard infinities (even `zoo` is commutative). We are left with very little that we can say about what such an expression *does*

represent.

3.1.7 Other is_* properties

There are many properties and attributes in SymPy that that have names beginning with `is_` that look similar to the properties used in the (old) assumptions system but are not in fact part of the assumptions system. Some of these have a similar meaning and usage as those of the assumptions system such as the *is_zero_matrix()* (page 1344) property shown above. Another example is the `is_empty` property of sets:

```
>>> from sympy import FiniteSet, Intersection
>>> S1 = FiniteSet(1, 2)
>>> S1
{1, 2}
>>> print(S1.is_empty)
False
>>> S2 = Intersection(FiniteSet(1), FiniteSet(Symbol('x')))
>>> S2
Intersection({1}, {x})
>>> print(S2.is_empty)
None
```

The `is_empty` property gives a fuzzy-bool indicating whether or not a *Set* (page 1185) is the empty set. In the example of `S2` it is not possible to know whether or not the set is empty without knowing whether or not `x` is equal to `1` so `S2.is_empty` gives `None`. The `is_empty` property for sets plays a similar role to the `is_zero` property for numbers in the assumptions system: `is_empty` is normally only `True` for the *EmptySet* (page 1202) object but it is still useful to be able to distinguish between the cases where `is_empty=False` and `is_empty=None`.

Although `is_zero_matrix` and `is_empty` are used for similar purposes to the assumptions properties such as `is_zero` they are not part of the (old) assumptions system. There are no associated inference rules connecting e.g. `Set.is_empty` and `Set.is_finite_set` because the inference rules are part of the (old) assumptions system which only deals with the predicates listed in the table above. It is not possible to declare a *MatrixSymbol* (page 1372) with e.g. `zero_matrix=False` and there is no `SetSymbol` class but if there was it would not have a system for understanding predicates like `empty=False`.

The properties *is_zero_matrix()* (page 1344) and `is_empty` are similar to those of the assumptions system because they concern *semantic* aspects of an expression. There are a large number of other properties that focus on *structural* aspects such as `is_Number`, *is_number()* (page 967), *is_comparable()* (page 935). Since these properties refer to structural aspects of an expression they will always give `True` or `False` rather than a fuzzy bool that also has the possibility of being `None`. Capitalised properties such as `is_Number` are usually shorthand for `isinstance` checks e.g.:

```
>>> from sympy import Number, Rational
>>> x = Rational(1, 2)
>>> isinstance(x, Number)
True
>>> x.is_Number
True
>>> y = Symbol('y', rational=True)
>>> isinstance(y, Number)
```

(continues on next page)

(continued from previous page)

```
False
>>> y.is_Number
False
```

The *Number* (page 981) class is the superclass for *Integer* (page 987), *Rational* (page 985) and *Float* (page 982) so any instance of *Number* (page 981) represents a concrete number with a known value. A symbol such as `y` that is declared with `rational=True` might represent the same value as `x` but it is not a concrete number with a known value so this is a structural rather than a semantic distinction. Properties like `is_Number` are sometimes used in SymPy in place of e.g. `isinstance(obj, Number)` because they do not have problems with circular imports and checking `x.is_Number` can be faster than a call to `isinstance`.

The *is_number* (page 967) (lower-case) property is very different from `is_Number`. The *is_number* (page 967) property is `True` for any expression that can be numerically evaluated to a floating point complex number with *evalf()* (page 1065):

```
>>> from sympy import I
>>> expr1 = I + sqrt(2)
>>> expr1
sqrt(2) + I
>>> expr1.is_number
True
>>> expr1.evalf()
1.4142135623731 + 1.0*I
>>> x = Symbol('x')
>>> expr2 = 1 + x
>>> expr2
x + 1
>>> expr2.is_number
False
>>> expr2.evalf()
x + 1.0
```

The primary reason for checking `expr.is_number` is to predict whether a call to *evalf()* (page 1065) will fully evaluate. The *is_comparable()* (page 935) property is similar to *is_number()* (page 967) except that if `is_comparable` gives `True` then the expression is guaranteed to numerically evaluate to a *real* *Float* (page 982). When `a.is_comparable` and `b.is_comparable` the inequality `a < b` should be resolvable as something like `a.evalf() < b.evalf()`.

The full set of `is_*` properties, attributes and methods in SymPy is large. It is important to be clear though that only those that are listed in the table of predicates above are actually part of the assumptions system. It is only those properties that are involved in the *mechanism* that implements the assumptions system which is explained below.

3.1.8 Implementing assumptions handlers

We will now work through an example of how to implement a SymPy symbolic function so that we can see how the old assumptions are used internally. SymPy already has an `exp` function which is defined for all complex numbers but we will define an `expreal` function which is restricted to real arguments.

```
>>> from sympy import Function
>>> from sympy.core.logic import fuzzy_and, fuzzy_or
>>>
>>> class expreal(Function):
...     """exponential function E**x restricted to the extended reals"""
...
...     is_extended_nonnegative = True
...
...     @classmethod
...     def eval(cls, x):
...         # Validate the argument
...         if x.is_extended_real is False:
...             raise ValueError("non-real argument to expreal")
...         # Evaluate for special values
...         if x.is_zero:
...             return S.One
...         elif x.is_infinite:
...             if x.is_extended_negative:
...                 return S.Zero
...             elif x.is_extended_positive:
...                 return S.Infinity
...
...     @property
...     def x(self):
...         return self.args[0]
...
...     def _eval_is_finite(self):
...         return fuzzy_or([self.x.is_real, self.x.is_extended_nonpositive])
...
...     def _eval_is_algebraic(self):
...         if fuzzy_and([self.x.is_rational, self.x.is_nonzero]):
...             return False
...
...     def _eval_is_integer(self):
...         if self.x.is_zero:
...             return True
...
...     def _eval_is_zero(self):
...         return fuzzy_and([self.x.is_infinite, self.x.is_extended_
↪negative])
```

The `Function.eval` method is used to pick up on special values of the function so that we can return a different object if it would be a simplification. When `expreal(x)` is called the `expreal.__new__` class method (defined in the superclass `Function`) will call `expreal.eval(x)`. If `expreal.eval` returns something other than `None` then that will be returned instead of an unevaluated `expreal(x)`:

```
>>> from sympy import oo
>>> expreal(1)
expreal(1)
>>> expreal(0)
1
>>> expreal(-oo)
0
>>> expreal(oo)
oo
```

Note that the `expreal.eval` method does not compare the argument using `==`. The special values are verified using the assumptions system to query the properties of the argument. That means that the `expreal` method can also evaluate for different forms of expression that have matching properties e.g.

```
>>> x = Symbol('x', extended_negative=True, infinite=True)
>>> x
x
>>> expreal(x)
0
```

Of course the assumptions system can only resolve a limited number of special values so most `eval` methods will also check against some special values with `==` but it is preferable to check e.g. `x.is_zero` rather than `x==0`.

Note also that the `expreal.eval` method validates that the argument is real. We want to allow $\pm\infty$ as arguments to `expreal` so we check for `extended_real` rather than `real`. If the argument is not extended real then we raise an error:

```
>>> expreal(I)
Traceback (most recent call last):
...
ValueError: non-real argument to expreal
```

Importantly we check `x.is_extended_real is False` rather than `not x.is_extended_real` which means that we only reject the argument if it is *definitely* not extended real: if `x.is_extended_real` gives `None` then the argument will not be rejected. The first reason for allowing `x.is_extended_real=None` is so that a vanilla symbol can be used with `expreal`. The second reason is that an assumptions query can always give `None` even in cases where an argument is definitely real e.g.:

```
>>> x = Symbol('x')
>>> print(x.is_extended_real)
None
>>> expreal(x)
expreal(x)
>>> expr = (1 + I)/sqrt(2) + (1 - I)/sqrt(2)
>>> print(expr.is_extended_real)
None
>>> expr.expand()
sqrt(2)
>>> expr.expand().is_extended_real
True
```

(continues on next page)

(continued from previous page)

```
>>> expreal(expr)
expreal(sqrt(2)*(1 - I)/2 + sqrt(2)*(1 + I)/2)
```

Validating the argument in `expreal.eval` does mean that it will not be validated when `evaluate=False` is passed but there is not really a better place to perform the validation:

```
>>> expreal(I, evaluate=False)
expreal(I)
```

The `extended_nonnegative` class attribute and the `_eval_is_*` methods on the `expreal` class implement queries in the assumptions system for instances of `expreal`:

```
>>> expreal(2)
expreal(2)
>>> expreal(2).is_finite
True
>>> expreal(2).is_integer
False
>>> expreal(2).is_rational
False
>>> expreal(2).is_algebraic
False
>>> z = expreal(-oo, evaluate=False)
>>> z
expreal(-oo)
>>> z.is_integer
True
>>> x = Symbol('x', real=True)
>>> expreal(x)
expreal(x)
>>> expreal(x).is_nonnegative
True
```

The assumptions system resolves queries like `expreal(2).is_finite` using the corresponding handler `expreal._eval_is_finite` and *also* the implication rules. For example it is known that `expreal(2).is_rational` is `False` because `expreal(2)._eval_is_algebraic` returns `False` and there is an implication rule `rational -> algebraic`. This means that an `is_rational` query can be resolved in this case by the `_eval_is_algebraic` handler. It is actually better not to implement assumptions handlers for every possible predicate but rather to try and identify a minimal set of handlers that can resolve as many queries as possible with as few checks as possible.

Another point to note is that the `_eval_is_*` methods only make assumptions queries on the argument `x` and do not make any assumptions queries on `self`. Recursive assumptions queries on the same object will interfere with the assumptions implications resolver potentially leading to non-deterministic behaviour so they should not be used (there are examples of this in the SymPy codebase but they should be removed).

Many of the `expreal` methods implicitly return `None`. This is a common pattern in the assumptions system. The `eval` method and the `_eval_is_*` methods can all return `None` and often will. A Python function that ends without reaching a `return` statement will implicitly return `None`. We take advantage of this by leaving out many of the `else` clauses from the `if` statements and allowing `None` to be returned implicitly. When following the control flow of these methods it is important to bear in mind firstly that any queried property can give `True`,

`False` or `None` and also that any function will implicitly return `None` if all of the conditionals fail.

3.1.9 Mechanism of the assumptions system

Note: This section describes internal details that could change in a future SymPy version.

This section will explain the inner workings of the assumptions system. It is important to understand that these inner workings are implementation details and could change from one SymPy version to another. This explanation is written as of SymPy 1.7. Although the (old) assumptions system has many limitations (discussed in the next section) it is a mature system that is used extensively in SymPy and has been well optimised for its current usage. The assumptions system is used implicitly in most SymPy operations to control evaluation of elementary expressions.

There are several stages in the implementation of the assumptions system within a SymPy process that lead up to the evaluation of a single query in the assumptions system. Briefly these are:

1. At import time the assumptions rules defined in `sympy/core/assumptions.py` are processed into a canonical form ready for efficiently applying the implication rules. This happens once when SymPy is imported before even the *Basic* (page 927) class is defined.
2. The `ManagedProperties` metaclass is defined which is the metaclass for all *Basic* (page 927) subclasses. This class will post-process every *Basic* (page 927) subclass to add the relevant properties needed for assumptions queries. This also adds the `default_assumptions` attribute to the class. This happens each time a *Basic* (page 927) subclass is defined.
3. Every *Basic* (page 927) instance initially uses the `default_assumptions` class attribute. When an assumptions query is made on a *Basic* (page 927) instance in the first instance the query will be answered from the `default_assumptions` for the class.
4. If there is no cached value for the assumptions query in the `default_assumptions` for the class then the default assumptions will be copied to make an assumptions cache for the instance. Then the `_ask()` function is called to resolve the query which will firstly call the relevant instance handler `_eval_is` method. If the handler returns non-None then the result will be cached and returned.
5. If the handler does not exist or gives None then the implications resolver is tried. This will enumerate (in a randomised order) all possible combinations of predicates that could potentially be used to resolve the query under the implication rules. In each case the handler `_eval_is` method will be called to see if it gives non-None. If any combination of handlers and implication rules leads to a definitive result for the query then that result is cached in the instance cache and returned.
6. Finally if the implications resolver failed to resolve the query then the query is considered unresolvable. The value of `None` for the query is cached in the instance cache and returned.

The assumptions rules defined in `sympy/core/assumptions.py` are given in forms like `real == negative | zero | positive`. When this module is imported these are converted into a `FactRules` instance called `_assume_rules`. This preprocesses the implication rules into the form of "A" and "B" rules that can be used for the implications resolver. This is explained in

the code in `sympy/core/facts.py`. We can access this internal object directly like (full output omitted):

```
>>> from sympy.core.assumptions import _assume_rules
>>> _assume_rules.defined_facts
{'algebraic',
 'antihermitian',
 'commutative',
 'complex',
 'composite',
 'even',
 ...
>>> _assume_rules.full_implications
defaultdict(set,
            {('extended_positive', False): {('composite', False),
  ('positive', False),
  ('prime', False)},
 ('finite', False): {('algebraic', False),
  ('complex', False),
  ('composite', False),
  ...
```

The `ManagedProperties` metaclass will inspect the attributes of each `Basic` class to see if any assumptions related attributes are defined. An example of these is the `is_extended_nonnegative = True` attribute defined in the `expreal` class. The implications of any such attributes will be used to precompute any statically knowable assumptions. For example `is_extended_nonnegative=True` implies `real=True` etc. A `StdFactKB` instance is created for the class which stores those assumptions whose values are known at this stage. The `StdFactKB` instance is assigned as the class attribute `default_assumptions`. We can see this with

```
>>> from sympy import Expr
...
>>> class A(Expr):
...     is_positive = True
...
...     def _eval_is_rational(self):
...         # Let's print something to see when this method is called...
...         print('!!! calling _eval_is_rational')
...         return True
...
>>> A.is_positive
True
>>> A.is_real  # inferred from is_positive
True
```

Although only `is_positive` was defined in the class `A` it also has attributes such as `is_real` which are inferred from `is_positive`. The set of all such assumptions for class `A` can be seen in `default_assumptions` which looks like a `dict` but is in fact a `StdFactKB` instance:

```
>>> type(A.default_assumptions)
<class 'sympy.core.assumptions.StdFactKB'>
>>> A.default_assumptions
{'commutative': True,
```

(continues on next page)

(continued from previous page)

```
 'complex': True,
 'extended_negative': False,
 'extended_nonnegative': True,
 'extended_nonpositive': False,
 'extended_nonzero': True,
 'extended_positive': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
 'infinite': False,
 'negative': False,
 'nonnegative': True,
 'nonpositive': False,
 'nonzero': True,
 'positive': True,
 'real': True,
 'zero': False}
```

When an instance of any *Basic* (page 927) subclass is created `Basic.__new__` will assign its `_assumptions` attribute which will initially be a reference to `cls.default_assumptions` shared amongst all instances of the same class. The instance will use this to resolve any assumptions queries until that fails to give a definitive result at which point a copy of `cls.default_assumptions` will be created and assigned to the instance's `_assumptions` attribute. The copy will be used as a cache to store any results computed for the instance by its `_eval_is` handlers.

When the `_assumptions` attribute fails to give the relevant result it is time to call the `_eval_is` handlers. At this point the `_ask()` function is called. The `_ask()` function will initially try to resolve a query such as `is_rational` by calling the corresponding method i.e. `_eval_is_rational`. If that gives non-None then the result is stored in `_assumptions` and any implications of that result are computed and stored as well. At that point the query is resolved and the value returned.

```
>>> a = A()
>>> a._assumptions is A.default_assumptions
True
>>> a.is_rational
!!! calling _eval_is_rational
True
>>> a._assumptions is A.default_assumptions
False
>>> a._assumptions   # rational now shows as True
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'extended_negative': False,
 'extended_nonnegative': True,
 'extended_nonpositive': False,
 'extended_nonzero': True,
 'extended_positive': True,
 'extended_real': True,
```

(continues on next page)

(continued from previous page)

```
 'finite': True,
 'hermitian': True,
 'imaginary': False,
 'infinite': False,
 'irrational': False,
 'negative': False,
 'nonnegative': True,
 'nonpositive': False,
 'nonzero': True,
 'positive': True,
 'rational': True,
 'real': True,
 'transcendental': False,
 'zero': False}
```

If e.g. `_eval_is_rational` does not exist or gives `None` then `_ask()` will try all possibilities to use the implication rules and any other handler methods such as `_eval_is_integer`, `_eval_is_algebraic` etc that might possibly be able to give an answer to the original query. If any method leads to a definite result being known for the original query then that is returned. Otherwise once all possibilities for using a handler and the implication rules to resolve the query are exhausted `None` will be cached and returned.

```
>>> b = A()
>>> b.is_algebraic    # called _eval_is_rational indirectly
!!! calling _eval_is_rational
True
>>> c = A()
>>> print(c.is_prime)   # called _eval_is_rational indirectly
!!! calling _eval_is_rational
None
>>> c._assumptions   # prime now shows as None
{'algebraic': True,
 'commutative': True,
 'complex': True,
 'extended_negative': False,
 'extended_nonnegative': True,
 'extended_nonpositive': False,
 'extended_nonzero': True,
 'extended_positive': True,
 'extended_real': True,
 'finite': True,
 'hermitian': True,
 'imaginary': False,
 'infinite': False,
 'irrational': False,
 'negative': False,
 'nonnegative': True,
 'nonpositive': False,
 'nonzero': True,
 'positive': True,
 'prime': None,
 'rational': True,
```

(continues on next page)

(continued from previous page)

```
 'real': True,
 'transcendental': False,
 'zero': False}
```

Note: In the `_ask()` function the handlers are called in a randomised order which can mean that execution at this point is non-deterministic. Provided all of the different handler methods are consistent (i.e. there are no bugs) then the end result will still be deterministic. However a bug where two handlers are inconsistent can manifest in non-deterministic behaviour because this randomisation might lead to the handlers being called in different orders when the same program is run multiple times.

3.1.10 Limitations

Combining predicates with or

In the old assumptions we can easily combine predicates with *and* when creating a Symbol e.g.:

```
>>> x = Symbol('x', integer=True, positive=True)
>>> x.is_positive
True
>>> x.is_integer
True
```

We can also easily query whether two conditions are jointly satisfied with

```
>>> fuzzy_and([x.is_positive, x.is_integer])
True
>>> x.is_positive and x.is_integer
True
```

However there is no way in the old assumptions to create a *Symbol* (page 976) with assumptions predicates combined with *or*. For example if we wanted to say that "x is positive or x is an integer" then it is not possible to create a *Symbol* (page 976) with those assumptions.

It is also not possible to ask an assumptions query based on *or* e.g. "is expr an expression that is positive or an integer". We can use e.g.

```
>>> fuzzy_or([x.is_positive, x.is_integer])
True
```

However if all that is known about `x` is that it is possibly positive or otherwise a negative integer then both queries `x.is_positive` and `x.is_integer` will resolve to `None`. That means that the query becomes

```
>>> fuzzy_or([None, None])
```

which then also gives `None`.

Relations between different symbols

A fundamental limitation of the old assumptions system is that all explicit assumptions are properties of an individual symbol. There is no way in this system to make an assumption about the *relationship* between two symbols. One of the most common requests is the ability to assume something like `x < y` but there is no way to even specify that in the old assumptions.

The new assumptions have the theoretical capability that relational assumptions can be specified. However the algorithms to make use of that information are not yet implemented and the exact API for specifying relational assumptions has not been decided upon.

3.2 Symbolic and fuzzy booleans

This page describes what a symbolic *Boolean* (page 1163) in SymPy is and also how that relates to three-valued fuzzy-bools that are used in many parts of SymPy. It also discusses some common problems that arise when writing code that uses three-valued logic and how to handle them correctly.

3.2.1 Symbolic Boolean vs three valued bool

Assumptions queries like `x.ispositive` give fuzzy-bool `True`, `False` or `None` results[1]. These are low-level Python objects rather than SymPy's symbolic *Boolean* (page 1163) expressions.

```
>>> from sympy import Symbol, symbols
>>> xpos = Symbol('xpos', positive=True)
>>> xneg = Symbol('xneg', negative=True)
>>> x = Symbol('x')
>>> print(xpos.is_positive)
True
>>> print(xneg.is_positive)
False
>>> print(x.is_positive)
None
```

A `None` result as a fuzzy-bool should be interpreted as meaning "maybe" or "unknown".

An example of a symbolic *Boolean* (page 1163) class in SymPy can be found when using inequalities. When an inequality is not known to be true or false a *Boolean* (page 1163) can represent indeterminate results symbolically:

```
>>> xpos > 0
True
>>> xneg > 0
False
>>> x > 0
x > 0
>>> type(x > 0)
<class 'sympy.core.relational.StrictGreaterThan'>
```

[1] Note that what is referred to in SymPy as a "fuzzy bool" is really about using three-valued logic. In normal usage "fuzzy logic" refers to a system where logical values are continuous in between zero and one which is something different from three-valued logic.

The last example shows what happens when an inequality is indeterminate: we get an instance of *StrictGreaterThan* (page 1032) which represents the inequality as a symbolic expression. Internally when attempting to evaluate an inequality like `a > b` SymPy will compute `(a - b).is_extended_positive`. If the result is `True` or `False` then SymPy's symbolic `S.true` or `S.false` will be returned. If the result is `None` then an unevaluated *StrictGreaterThan* (page 1032) is returned as shown for `x > 0` above.

It is not obvious that queries like `xpos > 0` return `S.true` rather than `True` because both objects display in the same way but we can check this using the Python `is` operator:

```
>>> from sympy import S
>>> xpos.is_positive is True
True
>>> xpos.is_positive is S.true
False
>>> (xpos > 0) is True
False
>>> (xpos > 0) is S.true
True
```

There is no general symbolic analogue of `None` in SymPy. In the cases where a low-level assumptions query gives `None` the symbolic query will result in an unevaluated symbolic *Boolean* (page 1163) (e.g, `x > 0`). We can use a symbolic *Boolean* (page 1163) as part of a symbolic expression such as a *Piecewise* (page 415):

```
>>> from sympy import Piecewise
>>> p = Piecewise((1, x > 0), (2, True))
>>> p
Piecewise((1, x > 0), (2, True))
>>> p.subs(x, 3)
1
```

Here `p` represents an expression that will be equal to `1` if `x > 0` or otherwise it will be equal to `2`. The unevaluated *Boolean* (page 1163) inequality `x > 0` represents the condition for deciding the value of the expression symbolically. When we substitute a value for `x` the inequality will resolve to `S.true` and then the *Piecewise* (page 415) can evaluate to `1` or `2`.

The same will not work when using a fuzzy-bool instead of a symbolic *Boolean* (page 1163):

```
>>> p2 = Piecewise((1, x.is_positive), (2, True))
Traceback (most recent call last):
...
TypeError: Second argument must be a Boolean, not `NoneType`
```

The *Piecewise* (page 415) can not use `None` as the condition because unlike the inequality `x > 0` it gives no information. With the inequality it is possible to decide in future if the condition might `True` or `False` once a value for `x` is known. A value of `None` can not be used in that way so it is rejected.

Note: We can use `True` in the *Piecewise* (page 415) because `True` sympifies to `S.true`. Sympifying `None` just gives `None` again which is not a valid symbolic SymPy object.

There are many other symbolic *Boolean* (page 1163) types in SymPy. The same considerations about the differences between fuzzy bool and symbolic *Boolean* (page 1163) apply to all other

SymPy *Boolean* (page 1163) types. To give a different example there is *Contains* (page 1216) which represents the statement that an object is contained in a set:

```
>>> from sympy import Reals, Contains
>>> x = Symbol('x', real=True)
>>> y = Symbol('y')
>>> Contains(x, Reals)
True
>>> Contains(y, Reals)
Contains(y, Reals)
>>> Contains(y, Reals).subs(y, 1)
True
```

The Python operator corresponding to *Contains* (page 1216) is `in`. A quirk of `in` is that it can only evaluate to a `bool` (`True` or `False`) so if the result is indeterminate then an exception will be raised:

```
>>> from sympy import I
>>> 2 in Reals
True
>>> I in Reals
False
>>> x in Reals
True
>>> y in Reals
Traceback (most recent call last):
...
TypeError: did not evaluate to a bool: (-oo < y) & (y < oo)
```

The exception can be avoided by using `Contains(x, Reals)` or `Reals.contains(x)` rather than `x in Reals`.

3.2.2 Three-valued logic with fuzzy bools

Whether we use the fuzzy-bool or symbolic *Boolean* (page 1163) we always need to be aware of the possibility that a query might be indeterminate. How to write code that handles this is different in the two cases though. We will look at fuzzy-bools first.

Consider the following function:

```
>>> def both_positive(a, b):
...     """ask whether a and b are both positive"""
...     if a.is_positive and b.is_positive:
...         return True
...     else:
...         return False
```

The `both_positive` function is supposed to tell us whether or not `a` and `b` are both positive. However the `both_positive` function will fail if either of the `is_positive` queries gives `None`:

```
>>> print(both_positive(S(1), S(1)))
True
>>> print(both_positive(S(1), S(-1)))
```

(continues on next page)

(continued from previous page)

```
False
>>> print(both_positive(S(-1), S(-1)))
False
>>> x = Symbol('x') # may or may not be positive
>>> print(both_positive(S(1), x))
False
```

Note: We need to sympify the arguments to this function using `S` because the assumptions are only defined on SymPy objects and not regular Python `int` objects.

Here `False` is incorrect because it is *possible* that `x` is positive in which case both arguments would be positive. We get `False` here because `x.is_positive` gives `None` and Python will treat `None` as "falsey".

In order to handle all possible cases correctly we need to separate the logic for identifying the `True` and `False` cases. An improved function might be:

```
>>> def both_positive_better(a, b):
...     """ask whether a and b are both positive"""
...     if a.is_positive is False or b.is_positive is False:
...         return False
...     elif a.is_positive is True and b.is_positive is True:
...         return True
...     else:
...         return None
```

This function now can handle all cases of `True`, `False` or `None` for both `a` and `b` and will always return a fuzzy bool representing whether the statement "`a` and `b` are both positive" is true, false or unknown:

```
>>> print(both_positive_better(S(1), S(1)))
True
>>> print(both_positive_better(S(1), S(-1)))
False
>>> x = Symbol('x')
>>> y = Symbol('y', positive=True)
>>> print(both_positive_better(S(1), x))
None
>>> print(both_positive_better(S(-1), x))
False
>>> print(both_positive_better(S(1), y))
True
```

Another case that we need to be careful of when using fuzzy-bools is negation with Python's `not` operator e.g.:

```
>>> x = Symbol('x')
>>> print(x.is_positive)
None
>>> not x.is_positive
True
```

The correct negation of a fuzzy bool `None` is `None` again. If we do not know whether the statement "`x` is positive" is `True` or `False` then we also do not know whether its negation "`x` is not positive" is `True` or `False`. The reason we get `True` instead is again because `None` is considered "falsey". When `None` is used with a logical operator such as `not` it will first be converted to a `bool` and then negated:

```
>>> bool(None)
False
>>> not bool(None)
True
>>> not None
True
```

The fact that `None` is treated as falsey can be useful if used correctly. For example we may want to do something only if `x` is known to positive in which case we can do

```
>>> x = Symbol('x', positive=True)
>>> if x.is_positive:
...     print("x is definitely positive")
... else:
...     print("x may or may not be positive")
x is definitely positive
```

Provided we understand that an alternate condition branch refers to two cases (`False` and `None`) then this can be a useful way of writing conditionals. When we really do need to distinguish all cases then we need to use things like `x.is_positive is False`. What we need to be careful of though is using Python's binary logic operators like `not` or `and` with fuzzy bools as they will not handle the indeterminate case correctly.

In fact SymPy has internal functions that are designed to handle fuzzy-bools correctly:

```
>>> from sympy.core.logic import fuzzy_not, fuzzy_and
>>> print(fuzzy_not(True))
False
>>> print(fuzzy_not(False))
True
>>> print(fuzzy_not(None))
None
>>> print(fuzzy_and([True, True]))
True
>>> print(fuzzy_and([True, None]))
None
>>> print(fuzzy_and([False, None]))
False
```

Using the `fuzzy_and` function we can write the `both_positive` function more simply:

```
>>> def both_positive_best(a, b):
...     """ask whether a and b are both positive"""
...     return fuzzy_and([a.is_positive, b.is_positive])
```

Making use of `fuzzy_and`, `fuzzy_or` and `fuzzy_not` leads to simpler code and can also reduce the chance of introducing a logic error because the code can look more like it would in the case of ordinary binary logic.

3.2.3 Three-valued logic with symbolic Booleans

When working with symbolic *Boolean* (page 1163) rather than fuzzy-bool the issue of `None` silently being treated as falsey does not arise so it is easier not to end up with a logic error. However instead the indeterminate case will often lead to an exception being raised if not handled carefully.

We will try to implement the `both_positive` function this time using symbolic *Boolean* (page 1163):

```
>>> def both_positive(a, b):
...     """ask whether a and b are both positive"""
...     if a > 0 and b > 0:
...         return S.true
...     else:
...         return S.false
```

The first difference is that we return the symbolic *Boolean* (page 1163) objects `S.true` and `S.false` rather than `True` and `False`. The second difference is that we test e.g. `a > 0` rather than `a.is_positive`. Trying this out we get

```
>>> both_positive(1, 2)
True
>>> both_positive(-1, 1)
False
>>> x = Symbol('x')  # may or may not be positive
>>> both_positive(x, 1)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

What happens now is that testing `x > 0` gives an exception when `x` is not known to be positive or not positive. More precisely `x > 0` does not give an exception but `if x > 0` does and that is because the `if` statement implicitly calls `bool(x > 0)` which raises.

```
>>> x > 0
x > 0
>>> bool(x > 0)
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
>>> if x > 0:
...     print("x is positive")
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
```

The Python expression `x > 0` creates a SymPy *Boolean* (page 1163). Since in this case the *Boolean* (page 1163) can not evaluate to `True` or `False` we get an unevaluated *StrictGreaterThan* (page 1032). Attempting to force that into a `bool` with `bool(x > 0)` raises an exception. That is because a regular Python `bool` must be either `True` or `False` and neither of those are known to be correct in this case.

The same kind of issue arises when using `and`, `or` or `not` with symbolic *Boolean* (page 1163). The solution is to use SymPy's symbolic *And* (page 1166), *Or* (page 1167) and *Not* (page 1167) or equivalently Python's bitwise logical operators `&`, `|` and `~`:

```
>>> from sympy import And, Or, Not
>>> x > 0
x > 0
>>> x > 0 and x < 1
Traceback (most recent call last):
...
TypeError: cannot determine truth value of Relational
>>> And(x > 0, x < 1)
(x > 0) & (x < 1)
>>> (x > 0) & (x < 1)
(x > 0) & (x < 1)
>>> Or(x < 0, x > 1)
(x > 1) | (x < 0)
>>> Not(x < 0)
x >= 0
>>> ~(x < 0)
x >= 0
```

As before we can make a better version of `both_positive` if we avoid directly using a SymPy *Boolean* (page 1163) in an `if`, `and`, `or`, or `not`. Instead we can test whether or not the *Boolean* (page 1163) has evaluated to `S.true` or `S.false`:

```
>>> def both_positive_better(a, b):
...     """ask whether a and b are both positive"""
...     if (a > 0) is S.false or (b > 0) is S.false:
...         return S.false
...     elif (a > 0) is S.true and (b > 0) is S.true:
...         return S.true
...     else:
...         return And(a > 0, b > 0)
```

Now with this version we don't get any exceptions and if the result is indeterminate we will get a symbolic *Boolean* (page 1163) representing the conditions under which the statement "a and b are both positive" would be true:

```
>>> both_positive_better(S(1), S(2))
True
>>> both_positive_better(S(1), S(-1))
False
>>> x, y = symbols("x, y")
>>> both_positive_better(x, y + 1)
(x > 0) & (y + 1 > 0)
>>> both_positive_better(x, S(3))
x > 0
```

The last case shows that actually using the *And* (page 1166) with a condition that is known to be true simplifies the *And* (page 1166). In fact we have

```
>>> And(x > 0, 3 > 0)
x > 0
>>> And(4 > 0, 3 > 0)
True
>>> And(-1 > 0, 3 > 0)
False
```

What this means is that we can improve `both_positive_better`. The different cases are not needed at all. Instead we can simply return the *And* (page 1166) and let it simplify if possible:

```
>>> def both_positive_best(a, b):
...     """ask whether a and b are both positive"""
...     return And(a > 0, b > 0)
```

Now this will work with any symbolic real objects and produce a symbolic result. We can also substitute into the result to see how it would work for particular values:

```
>>> both_positive_best(2, 1)
True
>>> both_positive_best(-1, 2)
False
>>> both_positive_best(x, 3)
x > 0
>>> condition = both_positive_best(x/y, x + y)
>>> condition
(x + y > 0) & (x/y > 0)
>>> condition.subs(x, 1)
(1/y > 0) & (y + 1 > 0)
>>> condition.subs(x, 1).subs(y, 2)
True
```

The idea when working with symbolic *Boolean* (page 1163) objects is as much as possible to avoid trying to branch on them with `if/else` and other logical operators like `and` etc. Instead think of computing a condition and passing it around as a variable. The elementary symbolic operations like *And* (page 1166), *Or* (page 1167) and *Not* (page 1167) can then take care of the logic for you.

3.3 Writing Custom Functions

This guide will describe how to create custom function classes in SymPy. Custom user defined functions use the same mechanisms as the *functions* (page 381) that are included with SymPy such as the common *elementary functions* (page 382) like *exp()* (page 411) or *sin()* (page 389), *special functions* (page 450) like *gamma()* (page 459) or *Si()* (page 491), and *combinatorial functions* (page 427) and *number theory functions* (page 1476) like *factorial()* (page 434) or *primepi()* (page 1479). Consequently, this guide serves both as a guide to end users who want to define their own custom functions and to SymPy developers wishing to extend the functions included with SymPy.

This guide describes how to define complex valued functions, that is functions that map a subset of $\mathbb{C}^n$ to $\mathbb{C}$. Functions that accept or return other kinds of objects than complex numbers should subclass another class, such as *Boolean* (page 1163), *MatrixExpr* (page 1370), *Expr* (page 947), or *Basic* (page 927). Some of what is written here will apply to general *Basic* (page 927) or *Expr* (page 947) subclasses, but much of it only applies to *Function* (page 1050) subclasses.

3.3.1 Easy Cases: Fully Symbolic or Fully Evaluated

Before digging into the more advanced functionality for custom functions, we should mention two common cases, the case where the function is fully symbolic, and the case where the function is fully evaluated. Both of these cases have much simpler alternatives than the full mechanisms described in ths guide.

The Fully Symbolic Case

If your function `f` has no mathematical properties you want to define on it, and should never evaluate on any arguments, you can create an undefined function using `Function('f')`

```
>>> from sympy import symbols, Function
>>> x = symbols('x')
>>> f = Function('f')
```

```
>>> f(x)
f(x)
>>> f(0)
f(0)
```

This is useful, for instance, when solving *ODEs* (page 755).

This is also useful if you only wish to create a symbol that depends on another symbol for the purposes of differentiation. By default, SymPy assumes all symbols are independent of one another:

```
>>> from sympy.abc import x, y
>>> y.diff(x)
0
```

To make a symbol that depends on another symbol, you can use a function that explicitly depends on that symbol.

```
>>> y = Function('y')
>>> y(x).diff(x)
Derivative(y(x), x)
```

If you want your function to have additional behavior, for example, to have a custom derivative, or to evaluate on certain arguments, you should create a custom `Function` subclass as *described below* (page 105). However, undefined functions do support one additional feature, which is that assumptions can be defined on them, using the same syntax as used by symbols. This defines the assumptions of the output of the function, not the input (that is, it defines the function's range, not its domain).

```
>>> g = Function('g', real=True)
```

```
>>> g(x)
g(x)
>>> g(x).is_real
True
```

To make a function's assumptions depend on its input in some way, you should create a custom `Function` subclass and define assumptions handlers as *described below* (page 111).

The Fully Evaluated Case

At the other end of the spectrum are functions that always evaluate to something no matter what their inputs are. These functions are never left in an unevaluated, symbolic form like `f(x)`.

In this case, you should use a normal Python function using the `def` keyword:

```
>>> def f(x):
...     if x == 0:
...         return 0
...     else:
...         return x + 1
```

```
>>> f(0)
0
>>> f(1)
2
>>> f(x)
x + 1
```

If you find yourself defining an *eval()* (page 106) method on a `Function` subclass where you always return a value and never return `None`, you should consider just using a normal Python function instead, as there is no benefit to using a symbolic `Function` subclass in that case (see the *Best Practices for eval()* (page 107) section below)

Note that in many cases, functions like these can be represented directly using SymPy classes. For example, the above function can be represented symbolically using *Piecewise* (page 415). The `Piecewise` expression can be evaluated for specific values of `x` using *subs()* (page 941).

```
>>> from sympy import Piecewise, Eq, pprint
>>> f = Piecewise((0, Eq(x, 0)), (x + 1, True))
```

```
>>> pprint(f, use_unicode=True)
⎧  0    for x = 0
⎨
⎩x + 1  otherwise
>>> f.subs(x, 0)
0
>>> f.subs(x, 1)
2
```

Fully symbolic representations like `Piecewise` have the advantage that they accurately represent symbolic values. For example, in the above Python `def` definition of `f`, `f(x)` implicitly assumes that `x` is nonzero. The `Piecewise` version handles this case correctly and won't evaluate to the $x \neq 0$ case unless `x` is known to not be zero.

Another option, if you want a function that not only evaluates, but always evaluates to a numerical value, is to use *lambdify()* (page 2100). This will convert a SymPy expression into a function that can be evaluated using NumPy.

```
>>> from sympy import lambdify
>>> func = lambdify(x, Piecewise((0, Eq(x, 0)), (x + 1, True)))
>>> import numpy as np
```

(continues on next page)

(continued from previous page)

```
>>> func(np.arange(5))
array([0., 2., 3., 4., 5.])
```

Ultimately, the correct tool for the job depends on what you are doing and what exact behavior you want.

3.3.2 Creating a Custom Function

The first step to creating a custom function is to subclass *Function* (page 1050). The name of the subclass will be the name of the function. Different methods should then be defined on this subclass, depending on what functionality you want to provide.

As a motivating example for this document, let's create a custom function class representing the versine function. Versine is a trigonometric function which was used historically alongside some of the more familiar trigonometric functions like sine and cosine. It is rarely used today. Versine can be defined by the identity

$$\operatorname{versin}(x) = 1 - \cos(x).$$

SymPy does not already include versine because it is used so rarely in modern mathematics and because it is so easily defined in terms of the more familiar cosine.

Let us start by subclassing `Function`.

```
>>> class versin(Function):
...     pass
```

At this point, `versin` has no behaviors defined on it. It is very similar to the *undefined functions* (page 103) we discussed above. Note that `versin` is a class, and `versin(x)` is an instance of this class.

```
>>> versin(x)
versin(x)
>>> isinstance(versin(x), versin)
True
```

Note: All the methods described below are optional. They can be included if you want to define the given behavior, but if they are omitted, SymPy will default to leaving things unevaluated. For example, if you do not define *differentiation* (page 118), *diff()* (page 1048) will just return an unevaluated *Derivative* (page 1042).

Defining Automatic Evaluation with `eval()`

Reminder

Remember that `eval()` should be defined with the `@classmethod` decorator.

The first and most common thing we might want to define on our custom function is automatic evaluation, that is, the cases where it will return an actual value instead of just remaining unevaluated as-is.

This is done by defining the class method `eval()`. `eval()` should take the arguments of the function and return either a value or `None`. If it returns `None`, the function will remain unevaluated in that case. This also serves to define the signature of the function (by default, without an `eval()` method, a `Function` subclass will accept any number of arguments).

For our function `versin`, we might recall that $\cos(n\pi) = (-1)^n$ for integer n, so $\text{versin}(n\pi) = 1 - (-1)^n$. We can make `versin` automatically evaluate to this value when passed an integer multiple of `pi`:

```
>>> from sympy import pi, Integer
>>> class versin(Function):
...     @classmethod
...     def eval(cls, x):
...         # If x is an integer multiple of pi, x/pi will cancel and be an␣
↪Integer
...         n = x/pi
...         if isinstance(n, Integer):
...             return 1 - (-1)**n
```

```
>>> versin(pi)
2
>>> versin(2*pi)
0
```

Here we make use of the fact that if a Python function does not explicitly return a value, it automatically returns `None`. So in the cases where the `if isinstance(n, Integer)` statement is not triggered, `eval()` returns `None` and `versin` remains unevaluated.

```
>>> versin(x*pi)
versin(pi*x)
```

Note: `Function` subclasses should not redefine `__new__` or `__init__`. If you want to implement behavior that isn't possible with `eval()`, it might make more sense to subclass *Expr* (page 947) rather than `Function`.

`eval()` can take any number of arguments, including an arbitrary number with `*args` and optional keyword arguments. The `.args` of the function will always be the arguments that were passed in by the user. For example

```
>>> class f(Function):
...     @classmethod
```

(continues on next page)

(continued from previous page)

```
...     def eval(cls, x, y=1, *args):
...         return None
```

```
>>> f(1).args
(1,)
>>> f(1, 2).args
(1, 2)
>>> f(1, 2, 3).args
(1, 2, 3)
```

Finally, note that automatic evaluation on floating-point inputs happens automatically once *evalf() is defined* (page 114), so you do not need to handle it explicitly in `eval()`.

Best Practices for `eval()`

Certain antipatterns are common when defining `eval()` methods and should be avoided.

- **Don't just return an expression.**

  In the above example, we might have been tempted to write

  ```
  >>> from sympy import cos
  >>> class versin(Function):
  ...     @classmethod
  ...     def eval(cls, x):
  ...         # !! Not actually a good eval() method !!
  ...         return 1 - cos(x)
  ```

  However, this would make it so that `versin(x)` would *always* return `1 - cos(x)`, regardless of what `x` is. If all you want is a quick shorthand to `1 - cos(x)`, that is fine, but would be much simpler and more explicit to just *use a Python function as described above* (page 104). If we defined `versin` like this, it would never actually be represented as `versin(x)`, and none of the other behavior we define below would matter, because the other behaviors we are going to define on the `versin` class only apply when the returned object is actually a `versin` instance. So for example, `versin(x).diff(x)` would actually just be `(1 - cos(x)).diff(x)`, instead of calling *the fdiff() method we define below* (page 118).

  Key Point

  The purpose of `eval()` is not to define what the function *is*, mathematically, but rather to specify on what inputs it should automatically evaluate. The mathematical definition of a function is determined through the specification of various mathematical properties with the methods outlined below, like *numerical evaluation* (page 114), *differentiation* (page 118), and so on.

  If you find yourself doing this, you should think about what you actually want to achieve. If you just want a shorthand function for an expression, it will be simpler to just *define a Python function* (page 104). If you really do want a symbolic function, think about when you want it to evaluate to something else and when you want it to stay unevaluated. One option is to make your function unevaluated in `eval()` and define a *doit() method* (page 116) to evaluate it.

- **Avoid too much automatic evaluation.**

  It is recommended to minimize what is evaluated automatically by `eval()`. It is typically better to put more advanced simplifications in *other methods* (page 115), like *`doit()`* (page 116). Remember that whatever you define for automatic evaluation will *always* evaluate.[1] As in the previous point, if you evaluate every value, there is little point to even having a symbolic function in the first place. For example, we might be tempted to evaluate some trig identities on `versin` in `eval()`, but then these identities would always evaluate, and it wouldn't be possible to represent one half of the identity.

  One should also avoid doing anything in `eval()` that is slow to compute. SymPy generally assumes that it is cheap to create expressions, and if this is not true, it can lead to performance issues.

  Finally, it is recommended to avoid performing automatic evaluation in `eval()` based on assumptions. Instead, `eval()` should typically only evaluate explicit numerical special values and return `None` for everything else. You might have noticed in *the example above* (page 106) that we used `isinstance(n, Integer)` instead of checking `n.is_integer` using the assumptions system. We could have done that instead, which would make `versin(n*pi)` evaluate even if `n = Symbol('n', integer=True)`. But this is a case where we might not always want evaluation to happen, and if `n` is a more complicated expression, `n.is_integer` might be more expensive to compute.

  Let's consider an example. Using the identity $\cos(x+y) = \cos(x)\cos(y) - \sin(x)\sin(y)$, we can derive the identity

$$\operatorname{versin}(x+y) = \operatorname{versin}(x)\operatorname{versin}(y) - \operatorname{versin}(x) - \operatorname{versin}(y) - \sin(x)\sin(y) + 1.$$

  Suppose we decided to automatically expand this in `eval()`:

```
>>> from sympy import Add, sin
>>> class versin(Function):
...     @classmethod
...     def eval(cls, x):
...         # !! Not actually a good eval() method !!
...         if isinstance(x, Add):
...             a, b = x.as_two_terms()
...             return (versin(a)*versin(b) - versin(a) - versin(b)
...                     - sin(a)*sin(b) + 1)
```

  This method recursively splits `Add` terms into two parts and applies the above identity.

```
>>> x, y, z = symbols('x y z')
>>> versin(x + y)
-sin(x)*sin(y) + versin(x)*versin(y) - versin(x) - versin(y) + 1
```

  But now it's impossible to represent `versin(x + y)` without it expanding. This will affect other methods too. For example, suppose we define *differentiation (see below)* (page 118):

[1] While it is technically possible to bypass automatic evaluation by using `evaluate=False`, this is recommended against for two reasons. Firstly, `evaluate=False` is fragile because any function that rebuilds the expression from its `.args` will not keep the `evaluate=False` flag, causing it to evaluate. Secondly, `evaluate=False` tends to be bug prone, because other code may be written expecting the invariants from the automatic evaluation to hold. It is much better to not evaluate such cases at all in `eval()`, and move such simplifications to *`doit()`* (page 116) instead.

```
>>> class versin(Function):
...     @classmethod
...     def eval(cls, x):
...         # !! Not actually a good eval() method !!
...         if isinstance(x, Add):
...             a, b = x.as_two_terms()
...             return (versin(a)*versin(b) - versin(a) - versin(b)
...                     - sin(a)*sin(b) + 1)
...
...     def fdiff(self, argindex=1):
...         return sin(self.args[0])
```

We would expect `versin(x + y).diff(x)` to return `sin(x + y)`, and indeed, if we hadn't expanded this identity in `eval()`, *it would* (page 119). But with this version, `versin(x + y)` gets automatically expanded before `diff()` gets called, instead we get a more complicated expression:

```
>>> versin(x + y).diff(x)
sin(x)*versin(y) - sin(x) - sin(y)*cos(x)
```

And things are even worse than that. Let's try an `Add` with three terms:

```
>>> versin(x + y + z)
(-sin(y)*sin(z) + versin(y)*versin(z) - versin(y) - versin(z) +
1)*versin(x) - sin(x)*sin(y + z) + sin(y)*sin(z) - versin(x) -
versin(y)*versin(z) + versin(y) + versin(z)
```

We can see that things are getting out of control quite quickly. In fact, `versin(Add(*symbols('x:100')))` (`versin()` on an `Add` with 100 terms) takes over a second to evaluate, and that's just to *create* the expression, without even doing anything with it yet.

Identities like this are better left out of `eval` and implemented in other methods instead (in the case of this identity, *`expand_trig()`* (page 118)).

- **When restricting the input domain: allow `None` input assumptions.**

  Our example function versin(x) is a function from $\mathbb{C}$ to $\mathbb{C}$, so it can accept any input. But suppose we had a function that only made sense with certain inputs. As a second example, let's define a function `divides` as

$$\text{divides}(m, n) = \begin{cases} 1 & \text{for } m \mid n \\ 0 & \text{for } m \not\mid n \end{cases}.$$

  That is, `divides(m, n)` will be `1` if `m` divides `n` and `0` otherwise. `divides` clearly only makes sense if `m` and `n` are integers.

  We might be tempted to define the `eval()` method for `divides` like this:

```
>>> class divides(Function):
...     @classmethod
...     def eval(cls, m, n):
...         # !! Not actually a good eval() method !!
...
```

(continues on next page)

(continued from previous page)

```
...         # Evaluate for explicit integer m and n. This part is fine.
...         if isinstance(m, Integer) and isinstance(n, Integer):
...             return int(n % m == 0)
...
...         # For symbolic arguments, require m and n to be integer.
...         # If we write the logic this way, we will run into trouble.
...         if not m.is_integer or not n.is_integer:
...             raise TypeError("m and n should be integers")
```

The problem here is that by using `if not m.is_integer`, we are requiring `m.is_integer` to be `True`. If it is `None`, it will fail (see the *guide on booleans and three-valued logic* (page 95) for details on what it means for an assumption to be `None`). This is problematic for two reasons. Firstly, it forces the user to define assumptions on any input variable. If the user omits them, it will fail:

```
>>> n, m = symbols('n m')
>>> print(n.is_integer)
None
>>> divides(m, n)
Traceback (most recent call last):
...
TypeError: m and n should be integers
```

Instead they have to write

```
>>> n, m = symbols('n m', integer=True)
>>> divides(m, n)
divides(m, n)
```

This may seem like an acceptable restriction, but there is a bigger problem. Sometimes, SymPy's assumptions system cannot deduce an assumption, even though it is mathematically true. In this case, it will give `None` (`None` means both "undefined" and "cannot compute" in SymPy's assumptions). For example

```
>>> # n and m are still defined as integer=True as above
>>> divides(2, (m**2 + m)/2)
Traceback (most recent call last):
...
TypeError: m and n should be integers
```

Here the expression `(m**2 + m)/2` is always an integer, but SymPy's assumptions system is not able to deduce this:

```
>>> print(((m**2 + m)/2).is_integer)
None
```

SymPy's assumptions system is always improving, but there will always be cases like this that it cannot deduce, due to the fundamental computational complexity of the problem, and the fact that the general problem is often undecidable.

Consequently, one should always test *negated* assumptions for input variables, that is, fail if the assumption is `False` but allow the assumption to be `None`.

```
>>> class divides(Function):
...     @classmethod
...     def eval(cls, m, n):
...         # Evaluate for explicit integer m and n. This part is fine.
...         if isinstance(m, Integer) and isinstance(n, Integer):
...             return int(n % m == 0)
...
...         # For symbolic arguments, require m and n to be integer.
...         # This is the better way to write this logic.
...         if m.is_integer is False or n.is_integer is False:
...             raise TypeError("m and n should be integers")
```

This still disallows non-integer inputs as desired:

```
>>> divides(1.5, 1)
Traceback (most recent call last):
...
TypeError: m and n should be integers
```

But it does not fail in cases where the assumption is `None`:

```
>>> divides(2, (m**2 + m)/2)
divides(2, m**2/2 + m/2)
>>> _.subs(m, 2)
0
>>> n, m = symbols('n m') # Redefine n and m without the integer␣
↪assumption
>>> divides(m, n)
divides(m, n)
```


Note: This rule of allowing `None` assumptions only applies to instances where an exception would be raised, such as type checking an input domain. In cases where simplifications or other operations are done, one should treat a `None` assumption as meaning "can be either `True` or `False`" and not perform an operation that might not be mathematically valid.

Assumptions

The next thing you might want to define are the assumptions on our function. The assumptions system allows defining what mathematical properties your function has given its inputs, for example, "$f(x)$ is *positive* when x is *real*."

The *guide on the assumptions system* (page 71) goes into the assumptions system in great detail. It is recommended to read through that guide first to understand what the different assumptions mean and how the assumptions system works.

The simplest case is a function that always has a given assumption regardless of its input. In this case, you can define is_*assumption* directly on the class.

For example, our *example `divides` function* (page 109) is always an integer, because its value is always either 0 or 1:

Note

From here on out in this guide, in the interest of space, we will omit the previous method definitions in the examples unless they are needed for the given example to work. There are *complete examples* (page 122) at the end of this guide with all the methods.

```
>>> class divides(Function):
...     is_integer = True
...     is_negative = False
```

```
>>> divides(m, n).is_integer
True
>>> divides(m, n).is_nonnegative
True
```

In general, however, the assumptions of a function depend on the assumptions of its inputs. In this case, you should define an _eval_*assumption* method.

For our versin(x) *example* (page 105), the function is always in $[0, 2]$ when x is real, and it is 0 exactly when x is an even multiple of π. So `versin(x)` should be *nonnegative* whenever `x` is *real* and *positive* whenever `x` is *real* and not an *even* multiple of π. Remember that by default, a function's domain is all of $\mathbb{C}$, and indeed `versin(x)` makes perfect sense with non-real `x`.

To see if `x` is an even multiple of `pi`, we can use *`as_independent()`* (page 952) to match `x` structurally as `coeff*pi`. Pulling apart subexpressions structurally like this in assumptions handlers is preferable to using something like `(x/pi).is_even`, because that will create a new expression `x/pi`. The creation of a new expression is much slower. Furthermore, whenever an expression is created, the constructors that are called when creating the expression will often themselves cause assumptions to be queried. If you are not careful, this can lead to infinite recursion. So a good general rule for assumptions handlers is, **never create a new expression in an assumptions handler**. Always pull apart the args of the function using structural methods like `as_independent`.

Note that versin(x) can be nonnegative for nonreal x, for example:

```
>>> from sympy import I
>>> 1 - cos(pi + I*pi)
1 + cosh(pi)
>>> (1 - cos(pi + I*pi)).evalf()
12.5919532755215
```

So for the `_eval_is_nonnegative` handler, we want to return `True` if `x.is_real` is `True` but `None` if `x.is_real` is either `False` or `None`. It is left as an exercise to the reader to handle the cases for nonreal `x` that make `versin(x)` nonnegative, using similar logic from the `_eval_is_positive` handler.

In the assumptions handler methods, as in all methods, we can access the arguments of the function using `self.args`.

```
>>> from sympy.core.logic import fuzzy_and, fuzzy_not
>>> class versin(Function):
...     def _eval_is_nonnegative(self):
...         # versin(x) is nonnegative if x is real
...         x = self.args[0]
```

(continues on next page)

(continued from previous page)

```
...         if x.is_real is True:
...             return True
...
...     def _eval_is_positive(self):
...         # versin(x) is positive iff x is real and not an even multiple of␣
↪pi
...         x = self.args[0]
...
...         # x.as_independent(pi, as_Add=False) will split x as a Mul of the
...         # form coeff*pi
...         coeff, pi_ = x.as_independent(pi, as_Add=False)
...         # If pi_ = pi, x = coeff*pi. Otherwise x is not (structurally) of
...         # the form coeff*pi.
...         if pi_ == pi:
...             return fuzzy_and([x.is_real, fuzzy_not(coeff.is_even)])
...         elif x.is_real is False:
...             return False
...         # else: return None. We do not know for sure whether x is an even
...         # multiple of pi
```

```
>>> versin(1).is_nonnegative
True
>>> versin(2*pi).is_positive
False
>>> versin(3*pi).is_positive
True
```

Note the use of `fuzzy_` functions in the more complicated `_eval_is_positive()` handler, and the careful handling of the `if`/`elif`. It is important when working with assumptions to always be careful about *handling three-valued logic correctly* (page 95). This ensures that the method returns the correct answer when `x.is_real` or `coeff.is_even` are `None`.

Warning: Never define is_*assumption* as a `@property` method. Doing so will break the automatic deduction of other assumptions. is_*assumption* should only ever be defined as a class variable equal to `True` or `False`. If the assumption depends on the `.args` of the function somehow, define the _eval_*assumption* method.

In this example, it is not necessary to define `_eval_is_real()` because it is deduced automatically from the other assumptions, since `nonnegative -> real`. In general, you should avoid defining assumptions that the assumptions system can deduce automatically given its *known facts* (page 77).

```
>>> versin(1).is_real
True
```

The assumptions system is often able to deduce more than you might think. For example, from the above, it can deduce that `versin(2*n*pi)` is zero when `n` is an integer.

```
>>> n = symbols('n', integer=True)
>>> versin(2*n*pi).is_zero
True
```

It's always worth checking if the assumptions system can deduce something automatically before manually coding it.

Finally, a word of warning: be very careful about correctness when coding assumptions. Make sure to use the exact *definitions* (page 77) of the various assumptions, and always check that you're handling `None` cases correctly with the fuzzy three-valued logic functions. Incorrect or inconsistent assumptions can lead to subtle bugs. It's recommended to use unit tests to check all the various cases whenever your function has a nontrivial assumption handler. All functions defined in SymPy itself are required to be extensively tested.

Numerical Evaluation with `evalf()`

Here we show how to define how a function should numerically evaluate to a floating point *`Float`* (page 982) value, for instance, via `evalf()`. Implementing numerical evaluation enables several behaviors in SymPy. For example, once `evalf()` is defined, you can plot your function, and things like inequalities can evaluate to explicit values.

If your function has the same name as a function in mpmath, which is the case for most functions included with SymPy, numerical evaluation will happen automatically and you do not need to do anything.

If this is not the case, numerical evaluation can be specified by defining the method `_eval_evalf(self, prec)`, where `prec` is the binary precision of the input. The method should return the expression evaluated to the given precision, or `None` if this is not possible.

Note: The `prec` argument to `_eval_evalf()` is the *binary* precision, that is, the number of bits in the floating-point representation. This differs from the first argument to the `evalf()` method, which is the *decimal* precision, or `dps`. For example, the default binary precision of `Float` is 53, corresponding to a decimal precision of 15. Therefore, if your `_eval_evalf()` method recursively calls evalf on another expression, it should call `expr._eval_evalf(prec)` rather than `expr.evalf(prec)`, as the latter will incorrectly use `prec` as the decimal precision.

We can define numerical evaluation for *our example* $\operatorname{versin}(x)$ *function* (page 105) by recursively evaluating $2\sin^2\left(\frac{x}{2}\right)$, which is a more numerically stable way of writing $1 - \cos(x)$.

```
>>> from sympy import sin
>>> class versin(Function):
...     def _eval_evalf(self, prec):
...         return (2*sin(self.args[0]/2)**2)._eval_evalf(prec)
```

```
>>> versin(1).evalf()
0.459697694131860
```

Once `_eval_evalf()` is defined, this enables the automatic evaluation of floating-point inputs. It is not required to implement this manually in *`eval()`* (page 106).

```
>>> versin(1.)
0.459697694131860
```

Note that `evalf()` may be passed any expression, not just one that can be evaluated numerically. In this case, it is expected that the numerical parts of an expression will be evaluated. A general pattern to follow is to recursively call `_eval_evalf(prec)` on the arguments of the function.

Whenever possible, it's best to reuse the evalf functionality defined in existing SymPy functions. However, in some cases it will be necessary to use mpmath directly.

Rewriting and Simplification

Various simplification functions and methods allow specifying their behavior on custom subclasses. Not every function in SymPy has such hooks. See the documentation of each individual function for details.

rewrite()

The *rewrite()* (page 940) method allows rewriting an expression in terms of a specific function or rule. For example,

```
>>> sin(x).rewrite(cos)
cos(x - pi/2)
```

To implement rewriting, define a method `_eval_rewrite(self, rule, args, **hints)`, where

- `rule` is the *rule* passed to the `rewrite()` method. Typically `rule` will be the class of the object to be rewritten to, although for more complex rewrites, it can be anything. Each object that defines `_eval_rewrite()` defines what rule(s) it supports. Many SymPy functions rewrite to common classes, like `expr.rewrite(Add)`, to perform simplifications or other computations.
- `args` are the arguments of the function to be used for rewriting. This should be used instead of `self.args` because any recursive expressions in the args will be rewritten in `args` (assuming the caller used `rewrite(deep=True)`, which is the default).
- `**hints` are additional keyword arguments which may be used to specify the behavior of the rewrite. Unknown hints should be ignored as they may be passed to other `_eval_rewrite()` methods.

The method should return a rewritten expression, using `args` as the arguments to the function, or `None` if the expression should be unchanged.

For our *versin example* (page 105), an obvious rewrite we can implement is rewriting `versin(x)` as `1 - cos(x)`:

```
>>> class versin(Function):
...     def _eval_rewrite(self, rule, args, **hints):
...         if rule == cos:
...             return 1 - cos(*args)
>>> versin(x).rewrite(cos)
1 - cos(x)
```

Once we've defined this, *simplify()* (page 661) is now able to simplify some expressions containing `versin`:

```
>>> from sympy import simplify
>>> simplify(versin(x) + cos(x))
1
```

doit()

The *doit()* (page 932) method is used to evaluate "unevaluated" functions. To define `doit()` implement `doit(self, deep=True, **hints)`. If `deep=True`, `doit()` should recursively call `doit()` on the arguments. `**hints` will be any other keyword arguments passed to the user, which should be passed to any recursive calls to `doit()`. You can use `hints` to allow the user to specify specific behavior for `doit()`.

The typical usage of `doit()` in custom `Function` subclasses is to perform more advanced evaluation which is not performed in *eval()* (page 106).

For example, for our *divides example* (page 109), there are several instances that could be simplified using some identities. For example, we defined `eval()` to evaluate on explicit integers, but we might also want to evaluate examples like `divides(k, k*n)` where the divisibility is symbolically true. One of the *best practices for eval()* (page 107) is to avoid too much automatic evaluation. Automatically evaluating in this case might be considered too much, as it would make use of the assumptions system, which could be expensive. Furthermore, we might want to be able to represent `divides(k, k*n)` without it always evaluating.

The solution is to implement these more advanced evaluations in `doit()`. That way, we can explicitly perform them by calling `expr.doit()`, but they won't happen by default. An example `doit()` for `divides` that performs this simplification (along with the *above definition of eval()* (page 111)) might look like this:

Note: If `doit()` returns a Python `int` literal, convert it to an `Integer` so that the returned object is a SymPy type.

```
>>> from sympy import Integer
>>> class divides(Function):
...     # Define evaluation on basic inputs, as well as type checking that the
...     # inputs are not nonintegral.
...     @classmethod
...     def eval(cls, m, n):
...         # Evaluate for explicit integer m and n.
...         if isinstance(m, Integer) and isinstance(n, Integer):
...             return int(n % m == 0)
...
...         # For symbolic arguments, require m and n to be integer.
...         if m.is_integer is False or n.is_integer is False:
...             raise TypeError("m and n should be integers")
...
...     # Define doit() as further evaluation on symbolic arguments using
...     # assumptions.
...     def doit(self, deep=False, **hints):
...         m, n = self.args
...         # Recursively call doit() on the args whenever deep=True.
...         # Be sure to pass deep=True and **hints through here.
...         if deep:
...             m, n = m.doit(deep=deep, **hints), n.doit(deep=deep, **hints)
...
...         # divides(m, n) is 1 iff n/m is an integer. Note that m and n are
...         # already assumed to be integers because of the logic in eval().
...         isint = (n/m).is_integer
```

(continues on next page)