2. We walk over x memory twice when once would have been sufficient

A better solution would fuse both element-wise operations into a single for loop

```
for(int i = i; i < n; i++)
{
    y[i] = sin(x[i]) / x[i];
}
```

Statically compiled projects like NumPy are unable to take advantage of such optimizations. Fortunately, SymPy is able to generate efficient low-level C or Fortran code. It can then depend on projects like Cython or f2py to compile and reconnect that code back up to Python. Fortunately this process is well automated and a SymPy user wishing to make use of this code generation should call the ufuncify function.

ufuncify is the third method available with Autowrap module. It basically implies 'Universal functions' and follows an ideology set by NumPy. The main point of ufuncify as compared to autowrap is that it allows arrays as arguments and can operate in an element-by-element fashion. The core operation done element-wise is in accordance to Numpy's array broadcasting rules. See this for more.

```
>>> from sympy import *
>>> from sympy.abc import x
>>> expr = sin(x)/x
```

```
>>> from sympy.utilities.autowrap import ufuncify
>>> f = ufuncify([x], expr)
```

This function f consumes and returns a NumPy array. Generally ufuncify performs at least as well as lambdify. If the expression is complicated then ufuncify often significantly outperforms the NumPy backed solution. Jensen has a good blog post on this topic.

Let us see an example for some quantitative analysis:

```
>>> from sympy.physics.hydrogen import R_nl
>>> expr = R_nl(3, 1, x, 6)
>>> expr
                -2⋅x
8⋅x⋅(4 - 4⋅x)⋅e
───────────────────
         3
```

The lambdify function translates SymPy expressions into Python functions, leveraging a variety of numerical libraries. By default lambdify relies on implementations in the math standard library. Naturally, Raw Python is faster than SymPy. However it also supports mpmath and most notably, numpy. Using the NumPy library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

Let us compare the speeds:

```
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.utilities.lambdify import lambdify
>>> fn_numpy = lambdify(x, expr, 'numpy')
>>> fn_fortran = ufuncify([x], expr, backend='f2py')
>>> from numpy import linspace
>>> xx = linspace(0, 1, 5)
```

(continues on next page)

(continued from previous page)

```
>>> fn_numpy(xx)
[ 0.          1.21306132  0.98101184  0.44626032  0.        ]
>>> fn_fortran(xx)
[ 0.          1.21306132  0.98101184  0.44626032  0.        ]
>>> import timeit
>>> timeit.timeit('fn_numpy(xx)', 'from __main__ import fn_numpy, xx',␣
↪number=10000)
0.18891601900395472
>>> timeit.timeit('fn_fortran(xx)', 'from __main__ import fn_fortran, xx',␣
↪number=10000)
0.004707066000264604
```

The options available with ufuncify are more or less the same as those available with `autowrap`.

There are other facilities available with SymPy to do efficient numeric computation. See *this* (page 1102) page for a comparison among them.

Classes and functions for rewriting expressions (sympy.codegen.rewriting)

Classes and functions useful for rewriting expressions for optimized code generation. Some languages (or standards thereof), e.g. C99, offer specialized math functions for better performance and/or precision.

Using the `optimize` function in this module, together with a collection of rules (represented as instances of `Optimization`), one can rewrite the expressions for this purpose:

```
>>> from sympy import Symbol, exp, log
>>> from sympy.codegen.rewriting import optimize, optims_c99
>>> x = Symbol('x')
>>> optimize(3*exp(2*x) - 3, optims_c99)
3*expm1(2*x)
>>> optimize(exp(2*x) - 1 - exp(-33), optims_c99)
expm1(2*x) - exp(-33)
>>> optimize(log(3*x + 3), optims_c99)
log1p(x) + log(3)
>>> optimize(log(2*x + 3), optims_c99)
log(2*x + 3)
```

The `optims_c99` imported above is tuple containing the following instances (which may be imported from `sympy.codegen.rewriting`):

- `expm1_opt`
- `log1p_opt`
- `exp2_opt`
- `log2_opt`
- `log2const_opt`

class `sympy.codegen.rewriting.`**`FuncMinusOneOptim`**(*func*, *func_m_1*, *opportunistic=True*)

Specialization of ReplaceOptim for functions evaluating “f(x) - 1”.

Parameters

func :

The function which is subtracted by one.

func_m_1 :

The specialized function evaluating `func(x) - 1`.

opportunistic : bool

When `True`, apply the transformation as long as the magnitude of the remaining number terms decreases. When `False`, only apply the transformation if it completely eliminates the number term.

Explanation

Numerical functions which go toward one as x go toward zero is often best implemented by a dedicated function in order to avoid catastrophic cancellation. One such example is `expm1(x)` in the C standard library which evaluates `exp(x) - 1`. Such functions preserves many more significant digits when its argument is much smaller than one, compared to subtracting one afterwards.

Examples

```
>>> from sympy import symbols, exp
>>> from sympy.codegen.rewriting import FuncMinusOneOptim
>>> from sympy.codegen.cfunctions import expm1
>>> x, y = symbols('x y')
>>> expm1_opt = FuncMinusOneOptim(exp, expm1)
>>> expm1_opt(exp(x) + 2*exp(5*y) - 3)
expm1(x) + 2*expm1(5*y)
```


replace_in_Add(*e*)

passed as second argument to Basic.replace(...)

class sympy.codegen.rewriting.**Optimization**(*cost_function=None*, *priority=1*)

Abstract base class for rewriting optimization.

Subclasses should implement `__call__` taking an expression as argument.

Parameters

cost_function : callable returning number

priority : number

class sympy.codegen.rewriting.**ReplaceOptim**(*query*, *value*, ***kwargs*)

Rewriting optimization calling replace on expressions.

Parameters

query :

First argument passed to replace.

value :

Second argument passed to replace.

Explanation

The instance can be used as a function on expressions for which it will apply the `replace` method (see *sympy.core.basic.Basic.replace()* (page 937)).

Examples

```
>>> from sympy import Symbol
>>> from sympy.codegen.rewriting import ReplaceOptim
>>> from sympy.codegen.cfunctions import exp2
>>> x = Symbol('x')
>>> exp2_opt = ReplaceOptim(lambda p: p.is_Pow and p.base == 2,
...     lambda p: exp2(p.exp))
>>> exp2_opt(2**x)
exp2(x)
```

`sympy.codegen.rewriting.`**`create_expand_pow_optimization`**`(`*limit*, *, *base_req=<function <lambda>>*`)`

Creates an instance of *ReplaceOptim* (page 1119) for expanding `Pow`.

Parameters

limit : int

The highest power which is expanded into multiplication.

base_req : function returning bool

Requirement on base for expansion to happen, default is to return the `is_symbol` attribute of the base.

Explanation

The requirements for expansions are that the base needs to be a symbol and the exponent needs to be an Integer (and be less than or equal to `limit`).

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.codegen.rewriting import create_expand_pow_optimization
>>> x = Symbol('x')
>>> expand_opt = create_expand_pow_optimization(3)
>>> expand_opt(x**5 + x**3)
x**5 + x*x*x
>>> expand_opt(x**5 + x**3 + sin(x)**3)
x**5 + sin(x)**3 + x*x*x
>>> opt2 = create_expand_pow_optimization(3, base_req=lambda b: not b.is_
→Function)
>>> opt2((x+1)**2 + sin(x)**2)
sin(x)**2 + (x + 1)*(x + 1)
```

sympy.codegen.rewriting.**optimize**(*expr*, *optimizations*)

Apply optimizations to an expression.

Parameters

expr : expression

optimizations : iterable of `Optimization` instances

The optimizations will be sorted with respect to `priority` (highest first).

Examples

```
>>> from sympy import log, Symbol
>>> from sympy.codegen.rewriting import optims_c99, optimize
>>> x = Symbol('x')
>>> optimize(log(x+3)/log(2) + log(x**2 + 1), optims_c99)
log1p(x**2) + log2(x + 3)
```

Additional AST nodes for operations on matrices. The nodes in this module are meant to represent optimization of matrix expressions within codegen's target languages that cannot be represented by SymPy expressions.

As an example, we can use *sympy.codegen.rewriting.optimize()* (page 1120) and the `matin_opt` optimization provided in *sympy.codegen.rewriting* (page 1118) to transform matrix multiplication under certain assumptions:


```
>>> from sympy import symbols, MatrixSymbol
>>> n = symbols('n', integer=True)
>>> A = MatrixSymbol('A', n, n)
>>> x = MatrixSymbol('x', n, 1)
>>> expr = A**(-1) * x
>>> from sympy import assuming, Q
>>> from sympy.codegen.rewriting import matinv_opt, optimize
>>> with assuming(Q.fullrank(A)):
...     optimize(expr, [matinv_opt])
MatrixSolve(A, vector=x)
```

class sympy.codegen.matrix_nodes.**MatrixSolve**(**args*, ***kwargs*)

Represents an operation to solve a linear matrix equation.

Parameters

matrix : MatrixSymbol

Matrix representing the coefficients of variables in the linear equation. This matrix must be square and full-rank (i.e. all columns must be linearly independent) for the solving operation to be valid.

vector : MatrixSymbol

One-column matrix representing the solutions to the equations represented in `matrix`.

Examples

```
>>> from sympy import symbols, MatrixSymbol
>>> from sympy.codegen.matrix_nodes import MatrixSolve
>>> n = symbols('n', integer=True)
>>> A = MatrixSymbol('A', n, n)
>>> x = MatrixSymbol('x', n, 1)
>>> from sympy.printing.numpy import NumPyPrinter
>>> NumPyPrinter().doprint(MatrixSolve(A, x))
'numpy.linalg.solve(A, x)'
>>> from sympy import octave_code
>>> octave_code(MatrixSolve(A, x))
'A \\ x'
```

Tools for simplifying expressions using approximations (sympy.codegen.approximations)

class sympy.codegen.approximations.**SeriesApprox**(*bounds*, *reltol*, *max_order=4*, *n_point_checks=4*, ***kwargs*)

Approximates functions by expanding them as a series.

Parameters

bounds : dict

Mapping expressions to length 2 tuple of bounds (low, high).

reltol : number

Threshold for when to ignore a term. Taken relative to the largest lower bound among bounds.

max_order : int

Largest order to include in series expansion

n_point_checks : int (even)

The validity of an expansion (with respect to reltol) is checked at discrete points (linearly spaced over the bounds of the variable). The number of points used in this numerical check is given by this number.

Examples

```
>>> from sympy import sin, pi
>>> from sympy.abc import x, y
>>> from sympy.codegen.rewriting import optimize
>>> from sympy.codegen.approximations import SeriesApprox
>>> bounds = {x: (-.1, .1), y: (pi-1, pi+1)}
>>> series_approx2 = SeriesApprox(bounds, reltol=1e-2)
>>> series_approx3 = SeriesApprox(bounds, reltol=1e-3)
>>> series_approx8 = SeriesApprox(bounds, reltol=1e-8)
>>> expr = sin(x)*sin(y)
```

(continues on next page)

(continued from previous page)

```
>>> optimize(expr, [series_approx2])
x*(-y + (y - pi)**3/6 + pi)
>>> optimize(expr, [series_approx3])
(-x**3/6 + x)*sin(y)
>>> optimize(expr, [series_approx8])
sin(x)*sin(y)
```

class sympy.codegen.approximations.**SumApprox**(*bounds*, *reltol*, ***kwargs*)

Approximates sum by neglecting small terms.

Parameters

bounds : dict

Mapping expressions to length 2 tuple of bounds (low, high).

reltol : number

Threshold for when to ignore a term. Taken relative to the largest lower bound among bounds.

Explanation

If terms are expressions which can be determined to be monotonic, then bounds for those expressions are added.

Examples

```
>>> from sympy import exp
>>> from sympy.abc import x, y, z
>>> from sympy.codegen.rewriting import optimize
>>> from sympy.codegen.approximations import SumApprox
>>> bounds = {x: (-1, 1), y: (1000, 2000), z: (-10, 3)}
>>> sum_approx3 = SumApprox(bounds, reltol=1e-3)
>>> sum_approx2 = SumApprox(bounds, reltol=1e-2)
>>> sum_approx1 = SumApprox(bounds, reltol=1e-1)
>>> expr = 3*(x + y + exp(z))
>>> optimize(expr, [sum_approx3])
3*(x + y + exp(z))
>>> optimize(expr, [sum_approx2])
3*y + 3*exp(z)
>>> optimize(expr, [sum_approx1])
3*y
```

Classes for abstract syntax trees (sympy.codegen.ast)

Types used to represent a full function/module as an Abstract Syntax Tree.

Most types are small, and are merely used as tokens in the AST. A tree diagram has been included below to illustrate the relationships between the AST types.

AST Type Tree

```
*Basic*
     |
     |
 CodegenAST
     |
     |--->AssignmentBase
     |             |--->Assignment
     |             |--->AugmentedAssignment
     |                                    |--->AddAugmentedAssignment
     |                                    |--->SubAugmentedAssignment
     |                                    |--->MulAugmentedAssignment
     |                                    |--->DivAugmentedAssignment
     |                                    |--->ModAugmentedAssignment
     |
     |--->CodeBlock
     |
     |
     |--->Token
              |--->Attribute
              |--->For
              |--->String
              |       |--->QuotedString
              |       |--->Comment
              |--->Type
              |       |--->IntBaseType
              |       |              |--->_SizedIntType
              |       |                               |--->SignedIntType
              |       |                               |--->UnsignedIntType
              |       |--->FloatBaseType
              |                        |--->FloatType
              |                        |--->ComplexBaseType
              |                                           |--->ComplexType
              |--->Node
              |       |--->Variable
              |       |           |---> Pointer
              |       |--->FunctionPrototype
              |                            |--->FunctionDefinition
              |--->Element
              |--->Declaration
              |--->While
              |--->Scope
              |--->Stream
              |--->Print
```

(continues on next page)

(continued from previous page)

```
              |--->FunctionCall
              |--->BreakToken
              |--->ContinueToken
              |--->NoneToken
              |--->Return
```

Predefined types

A number of Type instances are provided in the sympy.codegen.ast module for convenience. Perhaps the two most common ones for code-generation (of numeric codes) are float32 and float64 (known as single and double precision respectively). There are also precision generic versions of Types (for which the codeprinters selects the underlying data type at time of printing): real, integer, complex_, bool_.

The other Type instances defined are:

- intc: Integer type used by C's "int".
- intp: Integer type used by C's "unsigned".
- int8, int16, int32, int64: n-bit integers.
- uint8, uint16, uint32, uint64: n-bit unsigned integers.
- float80: known as "extended precision" on modern x86/amd64 hardware.
- complex64: Complex number represented by two float32 numbers
- complex128: Complex number represented by two float64 numbers

Using the nodes

It is possible to construct simple algorithms using the AST nodes. Let's construct a loop applying Newton's method:

```
>>> from sympy import symbols, cos
>>> from sympy.codegen.ast import While, Assignment, aug_assign, Print
>>> t, dx, x = symbols('tol delta val')
>>> expr = cos(x) - x**3
>>> whl = While(abs(dx) > t, [
...     Assignment(dx, -expr/expr.diff(x)),
...     aug_assign(x, '+', dx),
...     Print([x])
... ])
>>> from sympy import pycode
>>> py_str = pycode(whl)
>>> print(py_str)
while (abs(delta) > tol):
    delta = (val**3 - math.cos(val))/(-3*val**2 - math.sin(val))
    val += delta
    print(val)
>>> import math
>>> tol, val, delta = 1e-5, 0.5, float('inf')
```

(continues on next page)

(continued from previous page)

```
>>> exec(py_str)
1.1121416371
0.909672693737
0.867263818209
0.865477135298
0.865474033111
>>> print('%3.1g' % (math.cos(val) - val**3))
-3e-11
```

If we want to generate Fortran code for the same while loop we simple call `fcode`:

```
>>> from sympy import fcode
>>> print(fcode(whl, standard=2003, source_format='free'))
do while (abs(delta) > tol)
   delta = (val**3 - cos(val))/(-3*val**2 - sin(val))
   val = val + delta
   print *, val
end do
```

There is a function constructing a loop (or a complete function) like this in *sympy.codegen.algorithms* (page 1157).

class sympy.codegen.ast.**Assignment**(*lhs*, *rhs*)

Represents variable assignment for code generation.

Parameters

lhs : Expr

SymPy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include Symbol, MatrixSymbol, MatrixElement, and Indexed. Types that subclass these types are also supported.

rhs : Expr

SymPy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a Matrix type can be assigned to MatrixSymbol, but not to Symbol, as the dimensions will not align.

Examples

```
>>> from sympy import symbols, MatrixSymbol, Matrix
>>> from sympy.codegen.ast import Assignment
>>> x, y, z = symbols('x, y, z')
>>> Assignment(x, y)
Assignment(x, y)
>>> Assignment(x, 0)
Assignment(x, 0)
>>> A = MatrixSymbol('A', 1, 3)
>>> mat = Matrix([x, y, z]).T
>>> Assignment(A, mat)
Assignment(A, Matrix([[x, y, z]]))
```

(continues on next page)

(continued from previous page)

```
>>> Assignment(A[0, 1], x)
Assignment(A[0, 1], x)
```

class sympy.codegen.ast.**AssignmentBase**(*lhs*, *rhs*)

Abstract base class for Assignment and AugmentedAssignment.

Attributes:

op

[str] Symbol for assignment operator, e.g. "=", "+=", etc.

class sympy.codegen.ast.**Attribute**(*possibly parametrized*)

For use with *sympy.codegen.ast.Node* (page 1134) (which takes instances of `Attribute` as `attrs`).

Parameters

name : str

parameters : Tuple

Examples

```
>>> from sympy.codegen.ast import Attribute
>>> volatile = Attribute('volatile')
>>> volatile
volatile
>>> print(repr(volatile))
Attribute(String('volatile'))
>>> a = Attribute('foo', [1, 2, 3])
>>> a
foo(1, 2, 3)
>>> a.parameters == (1, 2, 3)
True
```


class sympy.codegen.ast.**AugmentedAssignment**(*lhs*, *rhs*)

Base class for augmented assignments.

Attributes:

binop

[str] Symbol for binary operation being applied in the assignment, such as "+", "*", etc.

class sympy.codegen.ast.**BreakToken**(**args*, ***kwargs*)

Represents 'break' in C/Python ('exit' in Fortran).

Use the premade instance `break_` or instantiate manually.

Examples

```
>>> from sympy import ccode, fcode
>>> from sympy.codegen.ast import break_
>>> ccode(break_)
'break'
>>> fcode(break_, source_format='free')
'exit'
```

class sympy.codegen.ast.**CodeBlock**(**args*)

Represents a block of code.

Explanation

For now only assignments are supported. This restriction will be lifted in the future.

Useful attributes on this object are:

left_hand_sides:
Tuple of left-hand sides of assignments, in order.

left_hand_sides:
Tuple of right-hand sides of assignments, in order.

free_symbols: Free symbols of the expressions in the right-hand sides
which do not appear in the left-hand side of an assignment.

Useful methods on this object are:

topological_sort:
Class method. Return a CodeBlock with assignments sorted so that variables are assigned before they are used.

cse:
Return a new CodeBlock with common subexpressions eliminated and pulled out as assignments.

Examples

```
>>> from sympy import symbols, ccode
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y = symbols('x y')
>>> c = CodeBlock(Assignment(x, 1), Assignment(y, x + 1))
>>> print(ccode(c))
x = 1;
y = x + 1;
```

cse(*symbols=None, optimizations=None, postprocess=None, order='canonical'*)

Return a new code block with common subexpressions eliminated.

Explanation

See the docstring of *sympy.simplify.cse_main.cse()* (page 685) for more information.

Examples

```
>>> from sympy import symbols, sin
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y, z = symbols('x y z')
```

```
>>> c = CodeBlock(
...     Assignment(x, 1),
...     Assignment(y, sin(x) + 1),
...     Assignment(z, sin(x) - 1),
... )
...
>>> c.cse()
CodeBlock(
    Assignment(x, 1),
    Assignment(x0, sin(x)),
    Assignment(y, x0 + 1),
    Assignment(z, x0 - 1)
)
```

classmethod topological_sort(*assignments*)

Return a CodeBlock with topologically sorted assignments so that variables are assigned before they are used.

Examples

```
>>> from sympy import symbols
>>> from sympy.codegen.ast import CodeBlock, Assignment
>>> x, y, z = symbols('x y z')
```

```
>>> assignments = [
...     Assignment(x, y + z),
...     Assignment(y, z + 1),
...     Assignment(z, 2),
... ]
>>> CodeBlock.topological_sort(assignments)
CodeBlock(
    Assignment(z, 2),
    Assignment(y, z + 1),
    Assignment(x, y + z)
)
```

class sympy.codegen.ast.**Comment**(**args*, ***kwargs*)

Represents a comment.

class sympy.codegen.ast.**ComplexType**(**args, **kwargs*)

Represents a complex floating point number.

class sympy.codegen.ast.**ContinueToken**(**args, **kwargs*)

Represents 'continue' in C/Python ('cycle' in Fortran)

Use the premade instance `continue_` or instantiate manually.

Examples

```
>>> from sympy import ccode, fcode
>>> from sympy.codegen.ast import continue_
>>> ccode(continue_)
'continue'
>>> fcode(continue_, source_format='free')
'cycle'
```

class sympy.codegen.ast.**Declaration**(**args, **kwargs*)

Represents a variable declaration

Parameters

variable : Variable

Examples

```
>>> from sympy.codegen.ast import Declaration, NoneToken, untyped
>>> z = Declaration('z')
>>> z.variable.type == untyped
True
>>> # value is special NoneToken() which must be tested with == operator
>>> z.variable.value is None  # won't work
False
>>> z.variable.value == None  # not PEP-8 compliant
True
>>> z.variable.value == NoneToken()  # OK
True
```


class sympy.codegen.ast.**Element**(**args, **kwargs*)

Element in (a possibly N-dimensional) array.

Examples

```
>>> from sympy.codegen.ast import Element
>>> elem = Element('x', 'ijk')
>>> elem.symbol.name == 'x'
True
>>> elem.indices
(i, j, k)
>>> from sympy import ccode
>>> ccode(elem)
```

(continues on next page)

(continued from previous page)

```
'x[i][j][k]'
>>> ccode(Element('x', 'ijk', strides='lmn', offset='o'))
'x[i*l + j*m + k*n + o]'
```

class sympy.codegen.ast.**FloatBaseType**(**args*, ***kwargs*)

Represents a floating point number type.

cast_nocheck

alias of *Float* (page 982)

class sympy.codegen.ast.**FloatType**(**args*, ***kwargs*)

Represents a floating point type with fixed bit width.

Base 2 & one sign bit is assumed.

Parameters

name : str

Name of the type.

nbits : integer

Number of bits used (storage).

nmant : integer

Number of bits used to represent the mantissa.

nexp : integer

Number of bits used to represent the mantissa.

Examples

```
>>> from sympy import S
>>> from sympy.codegen.ast import FloatType
>>> half_precision = FloatType('f16', nbits=16, nmant=10, nexp=5)
>>> half_precision.max
65504
>>> half_precision.tiny == S(2)**-14
True
>>> half_precision.eps == S(2)**-10
True
>>> half_precision.dig == 3
True
>>> half_precision.decimal_dig == 5
True
>>> half_precision.cast_check(1.0)
1.0
>>> half_precision.cast_check(1e5)
Traceback (most recent call last):
  ...
ValueError: Maximum value for data type smaller than new value.
```

cast_nocheck(*value*)

Casts without checking if out of bounds or subnormal.

property decimal_dig

Number of digits needed to store & load without loss.

Explanation

Number of decimal digits needed to guarantee that two consecutive conversions (float -> text -> float) to be idempotent. This is useful when one do not want to loose precision due to rounding errors when storing a floating point value as text.

property dig

Number of decimal digits that are guaranteed to be preserved in text.

When converting text -> float -> text, you are guaranteed that at least `dig` number of digits are preserved with respect to rounding or overflow.

property eps

Difference between 1.0 and the next representable value.

property max

Maximum value representable.

property max_exponent

The largest positive number n, such that 2**(n - 1) is a representable finite value.

property min_exponent

The lowest negative number n, such that 2**(n - 1) is a valid normalized number.

property tiny

The minimum positive normalized value.

class sympy.codegen.ast.**For**(**args*, ***kwargs*)

Represents a 'for-loop' in the code.

Expressions are of the form:

"for target in iter:
body..."

Parameters

target : symbol

iter : iterable body : CodeBlock or iterable

! When passed an iterable it is used to instantiate a CodeBlock.

Examples

```
>>> from sympy import symbols, Range
>>> from sympy.codegen.ast import aug_assign, For
>>> x, i, j, k = symbols('x i j k')
>>> for_i = For(i, Range(10), [aug_assign(x, '+', i*j*k)])
>>> for_i
For(i, iterable=Range(0, 10, 1), body=CodeBlock(
    AddAugmentedAssignment(x, i*j*k)
))
>>> for_ji = For(j, Range(7), [for_i])
>>> for_ji
For(j, iterable=Range(0, 7, 1), body=CodeBlock(
    For(i, iterable=Range(0, 10, 1), body=CodeBlock(
        AddAugmentedAssignment(x, i*j*k)
    ))
))
>>> for_kji =For(k, Range(5), [for_ji])
>>> for_kji
For(k, iterable=Range(0, 5, 1), body=CodeBlock(
    For(j, iterable=Range(0, 7, 1), body=CodeBlock(
        For(i, iterable=Range(0, 10, 1), body=CodeBlock(
            AddAugmentedAssignment(x, i*j*k)
        ))
    ))
))
```


class sympy.codegen.ast.**FunctionCall**(**args*, ***kwargs*)

Represents a call to a function in the code.

Parameters

name : str

function_args : Tuple

Examples

```
>>> from sympy.codegen.ast import FunctionCall
>>> from sympy import pycode
>>> fcall = FunctionCall('foo', 'bar baz'.split())
>>> print(pycode(fcall))
foo(bar, baz)
```

class sympy.codegen.ast.**FunctionDefinition**(**args*, ***kwargs*)

Represents a function definition in the code.

Parameters

return_type : Type

name : str

parameters: iterable of Variable instances

body : CodeBlock or iterable

attrs : iterable of Attribute instances

Examples

```
>>> from sympy import ccode, symbols
>>> from sympy.codegen.ast import real, FunctionPrototype
>>> x, y = symbols('x y', real=True)
>>> fp = FunctionPrototype(real, 'foo', [x, y])
>>> ccode(fp)
'double foo(double x, double y)'
>>> from sympy.codegen.ast import FunctionDefinition, Return
>>> body = [Return(x*y)]
>>> fd = FunctionDefinition.from_FunctionPrototype(fp, body)
>>> print(ccode(fd))
double foo(double x, double y){
    return x*y;
}
```

class sympy.codegen.ast.**FunctionPrototype**(**args*, ***kwargs*)

Represents a function prototype

Allows the user to generate forward declaration in e.g. C/C++.

Parameters

return_type : Type

name : str

parameters: iterable of Variable instances

attrs : iterable of Attribute instances

Examples

```
>>> from sympy import ccode, symbols
>>> from sympy.codegen.ast import real, FunctionPrototype
>>> x, y = symbols('x y', real=True)
>>> fp = FunctionPrototype(real, 'foo', [x, y])
>>> ccode(fp)
'double foo(double x, double y)'
```

class sympy.codegen.ast.**IntBaseType**(**args*, ***kwargs*)

Integer base type, contains no size information.

class sympy.codegen.ast.**Node**(**args*, ***kwargs*)

Subclass of Token, carrying the attribute 'attrs' (Tuple)

Examples

```
>>> from sympy.codegen.ast import Node, value_const, pointer_const
>>> n1 = Node([value_const])
>>> n1.attr_params('value_const')  # get the parameters of attribute (by␣
↪name)
()
>>> from sympy.codegen.fnodes import dimension
>>> n2 = Node([value_const, dimension(5, 3)])
>>> n2.attr_params(value_const)  # get the parameters of attribute (by␣
↪Attribute instance)
()
>>> n2.attr_params('dimension')  # get the parameters of attribute (by␣
↪name)
(5, 3)
>>> n2.attr_params(pointer_const) is None
True
```

attr_params(*looking_for*)

Returns the parameters of the Attribute with name `looking_for` in self.attrs

class sympy.codegen.ast.**NoneToken**(**args*, ***kwargs*)

The AST equivalence of Python's NoneType

The corresponding instance of Python's `None` is `none`.

Examples

```
>>> from sympy.codegen.ast import none, Variable
>>> from sympy import pycode
>>> print(pycode(Variable('x').as_Declaration(value=none)))
x = None
```

class sympy.codegen.ast.**Pointer**(**args*, ***kwargs*)

Represents a pointer. See `Variable`.

Examples

Can create instances of `Element`:

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Pointer
>>> i = Symbol('i', integer=True)
>>> p = Pointer('x')
>>> p[i+1]
Element(x, indices=(i + 1,))
```

class sympy.codegen.ast.**Print**(**args*, ***kwargs*)

Represents print command in the code.

Parameters

formatstring : str

***args** : Basic instances (or convertible to such through sympify)

Examples

```
>>> from sympy.codegen.ast import Print
>>> from sympy import pycode
>>> print(pycode(Print('x y'.split(), "coordinate: %12.5g %12.5g")))
print("coordinate: %12.5g %12.5g" % (x, y))
```

class sympy.codegen.ast.**QuotedString**(**args*, ***kwargs*)

Represents a string which should be printed with quotes.

class sympy.codegen.ast.**Return**(**args*, ***kwargs*)

Represents a return command in the code.

Parameters

return : Basic

Examples

```
>>> from sympy.codegen.ast import Return
>>> from sympy.printing.pycode import pycode
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> print(pycode(Return(x)))
return x
```


class sympy.codegen.ast.**Scope**(**args*, ***kwargs*)

Represents a scope in the code.

Parameters

body : CodeBlock or iterable

When passed an iterable it is used to instantiate a CodeBlock.

class sympy.codegen.ast.**SignedIntType**(**args*, ***kwargs*)

Represents a signed integer type.

class sympy.codegen.ast.**Stream**(**args*, ***kwargs*)

Represents a stream.

There are two predefined Stream instances `stdout` & `stderr`.

Parameters

name : str

Examples

```
>>> from sympy import pycode, Symbol
>>> from sympy.codegen.ast import Print, stderr, QuotedString
>>> print(pycode(Print(['x'], file=stderr)))
print(x, file=sys.stderr)
>>> x = Symbol('x')
>>> print(pycode(Print([QuotedString('x')], file=stderr)))  # print␣
↪literally "x"
print("x", file=sys.stderr)
```

class sympy.codegen.ast.**String**(**args*, ***kwargs*)

SymPy object representing a string.

Atomic object which is not an expression (as opposed to Symbol).

Parameters

text : str

Examples

```
>>> from sympy.codegen.ast import String
>>> f = String('foo')
>>> f
foo
>>> str(f)
'foo'
>>> f.text
'foo'
>>> print(repr(f))
String('foo')
```


class sympy.codegen.ast.**Token**(**args*, ***kwargs*)

Base class for the AST types.

Explanation

Defining fields are set in `_fields`. Attributes (defined in _fields) are only allowed to contain instances of Basic (unless atomic, see `String`). The arguments to `__new__()` correspond to the attributes in the order defined in `_fields`. The ``defaults` class attribute is a dictionary mapping attribute names to their default values.

Subclasses should not need to override the `__new__()` method. They may define a class or static method named `_construct_<attr>` for each attribute to process the value passed to `__new__()`. Attributes listed in the class attribute `not_in_args` are not passed to *Basic* (page 927).

kwargs(*exclude=()*, *apply=None*)

Get instance's attributes as dict of keyword arguments.

Parameters

exclude : collection of str

Collection of keywords to exclude.

apply : callable, optional

Function to apply to all values.

class sympy.codegen.ast.**Type**(**args*, ***kwargs*)

Represents a type.

Parameters

name : str

Name of the type, e.g. `object`, `int16`, `float16` (where the latter two would use the `Type` sub-classes `IntType` and `FloatType` respectively). If a `Type` instance is given, the said instance is returned.

Explanation

The naming is a super-set of NumPy naming. Type has a classmethod `from_expr` which offer type deduction. It also has a method `cast_check` which casts the argument to its type, possibly raising an exception if rounding error is not within tolerances, or if the value is not representable by the underlying data type (e.g. unsigned integers).

Examples

```
>>> from sympy.codegen.ast import Type
>>> t = Type.from_expr(42)
>>> t
integer
>>> print(repr(t))
IntBaseType(String('integer'))
>>> from sympy.codegen.ast import uint8
>>> uint8.cast_check(-1)
Traceback (most recent call last):
  ...
ValueError: Minimum value for data type bigger than new value.
>>> from sympy.codegen.ast import float32
>>> v6 = 0.123456
>>> float32.cast_check(v6)
0.123456
>>> v10 = 12345.67894
>>> float32.cast_check(v10)
Traceback (most recent call last):
  ...
ValueError: Casting gives a significantly different value.
>>> boost_mp50 = Type('boost::multiprecision::cpp_dec_float_50')
>>> from sympy import cxxcode
>>> from sympy.codegen.ast import Declaration, Variable
>>> cxxcode(Declaration(Variable('x', type=boost_mp50)))
'boost::multiprecision::cpp_dec_float_50 x'
```

References

[R37]

cast_check(*value, rtol=None, atol=0, precision_targets=None*)

Casts a value to the data type of the instance.

Parameters

value : number

rtol : floating point number

Relative tolerance. (will be deduced if not given).

atol : floating point number

Absolute tolerance (in addition to `rtol`).

type_aliases : dict

Maps substitutions for Type, e.g. {integer: int64, real: float32}

Examples

```
>>> from sympy.codegen.ast import integer, float32, int8
>>> integer.cast_check(3.0) == 3
True
>>> float32.cast_check(1e-40)
Traceback (most recent call last):
  ...
ValueError: Minimum value for data type bigger than new value.
>>> int8.cast_check(256)
Traceback (most recent call last):
  ...
ValueError: Maximum value for data type smaller than new value.
>>> v10 = 12345.67894
>>> float32.cast_check(v10)
Traceback (most recent call last):
  ...
ValueError: Casting gives a significantly different value.
>>> from sympy.codegen.ast import float64
>>> float64.cast_check(v10)
12345.67894
>>> from sympy import Float
>>> v18 = Float('0.123456789012345646')
>>> float64.cast_check(v18)
Traceback (most recent call last):
  ...
ValueError: Casting gives a significantly different value.
>>> from sympy.codegen.ast import float80
>>> float80.cast_check(v18)
0.123456789012345649
```

classmethod from_expr(*expr*)

Deduces type from an expression or a `Symbol`.

Parameters

expr : number or SymPy object

The type will be deduced from type or properties.

Raises

ValueError when type deduction fails.

Examples

```
>>> from sympy.codegen.ast import Type, integer, complex_
>>> Type.from_expr(2) == integer
True
>>> from sympy import Symbol
>>> Type.from_expr(Symbol('z', complex=True)) == complex_
True
>>> Type.from_expr(sum)
Traceback (most recent call last):
  ...
ValueError: Could not deduce type from expr.
```

class sympy.codegen.ast.**UnsignedIntType**(**args*, ***kwargs*)

Represents an unsigned integer type.

class sympy.codegen.ast.**Variable**(**args*, ***kwargs*)

Represents a variable.

Parameters

symbol : Symbol

type : Type (optional)

Type of the variable.

attrs : iterable of Attribute instances

Will be stored as a Tuple.

Examples

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Variable, float32, integer
>>> x = Symbol('x')
>>> v = Variable(x, type=float32)
>>> v.attrs
()
>>> v == Variable('x')
False
>>> v == Variable('x', type=float32)
True
>>> v
Variable(x, type=float32)
```

One may also construct a `Variable` instance with the type deduced from assumptions about the symbol using the deduced classmethod:

```
>>> i = Symbol('i', integer=True)
>>> v = Variable.deduced(i)
>>> v.type == integer
True
>>> v == Variable('i')
False
>>> from sympy.codegen.ast import value_const
>>> value_const in v.attrs
False
>>> w = Variable('w', attrs=[value_const])
>>> w
Variable(w, attrs=(value_const,))
>>> value_const in w.attrs
True
>>> w.as_Declaration(value=42)
Declaration(Variable(w, value=42, attrs=(value_const,)))
```

as_Declaration(***kwargs*)

Convenience method for creating a Declaration instance.

Explanation

If the variable of the Declaration need to wrap a modified variable keyword arguments may be passed (overriding e.g. the value of the Variable instance).

Examples

```
>>> from sympy.codegen.ast import Variable, NoneToken
>>> x = Variable('x')
>>> decl1 = x.as_Declaration()
>>> # value is special NoneToken() which must be tested with ==␣
↪operator
>>> decl1.variable.value is None  # won't work
False
>>> decl1.variable.value == None  # not PEP-8 compliant
True
>>> decl1.variable.value == NoneToken()  # OK
True
>>> decl2 = x.as_Declaration(value=42.0)
>>> decl2.variable.value == 42
True
```

classmethod deduced(*symbol, value=None, attrs=(), cast_check=True*)

Alt. constructor with type deduction from `Type.from_expr`.

Deduces type primarily from `symbol`, secondarily from `value`.

Parameters

symbol : Symbol

value : expr

(optional) value of the variable.

attrs : iterable of Attribute instances

cast_check : bool

Whether to apply `Type.cast_check` on `value`.

Examples

```
>>> from sympy import Symbol
>>> from sympy.codegen.ast import Variable, complex_
>>> n = Symbol('n', integer=True)
>>> str(Variable.deduced(n).type)
'integer'
>>> x = Symbol('x', real=True)
>>> v = Variable.deduced(x)
>>> v.type
real
>>> z = Symbol('z', complex=True)
>>> Variable.deduced(z).type == complex_
True
```

class `sympy.codegen.ast.`**`While`**(**args*, ***kwargs*)

Represents a 'for-loop' in the code.

Expressions are of the form:

"while condition:
body..."

Parameters

condition : expression convertible to Boolean

body : CodeBlock or iterable

When passed an iterable it is used to instantiate a CodeBlock.

Examples

```
>>> from sympy import symbols, Gt, Abs
>>> from sympy.codegen import aug_assign, Assignment, While
>>> x, dx = symbols('x dx')
>>> expr = 1 - x**2
>>> whl = While(Gt(Abs(dx), 1e-9), [
...     Assignment(dx, -expr/expr.diff(x)),
...     aug_assign(x, '+', dx)
... ])
```

`sympy.codegen.ast.`**`aug_assign`**(*lhs*, *op*, *rhs*)

Create 'lhs op= rhs'.

Parameters

lhs : Expr

SymPy object representing the lhs of the expression. These should be singular objects, such as one would use in writing code. Notable types include Symbol, MatrixSymbol, MatrixElement, and Indexed. Types that subclass these types are also supported.

op : str

Operator (+, -, /, *, %).

rhs : Expr

SymPy object representing the rhs of the expression. This can be any type, provided its shape corresponds to that of the lhs. For example, a Matrix type can be assigned to MatrixSymbol, but not to Symbol, as the dimensions will not align.

Explanation

Represents augmented variable assignment for code generation. This is a convenience function. You can also use the AugmentedAssignment classes directly, like AddAugmentedAssignment(x, y).

Examples

```
>>> from sympy import symbols
>>> from sympy.codegen.ast import aug_assign
>>> x, y = symbols('x, y')
>>> aug_assign(x, '+', y)
AddAugmentedAssignment(x, y)
```


Special C math functions (sympy.codegen.cfunctions)

This module contains SymPy functions mathcin corresponding to special math functions in the C standard library (since C99, also available in C++11).

The functions defined in this module allows the user to express functions such as `expm1` as a SymPy function for symbolic manipulation.

class sympy.codegen.cfunctions.**Cbrt**(**args*)

Represents the cube root function.

Explanation

The reason why one would use `Cbrt(x)` over `cbrt(x)` is that the latter is internally represented as `Pow(x, Rational(1, 3))` which may not be what one wants when doing code-generation.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import Cbrt
>>> Cbrt(x)
Cbrt(x)
>>> Cbrt(x).diff(x)
1/(3*x**(2/3))
```

See also:

Sqrt (page 1144)

fdiff(*argindex=1*)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**Sqrt**(**args*)

Represents the square root function.

Explanation

The reason why one would use `Sqrt(x)` over `sqrt(x)` is that the latter is internally represented as `Pow(x, S.Half)` which may not be what one wants when doing code-generation.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import Sqrt
>>> Sqrt(x)
Sqrt(x)
>>> Sqrt(x).diff(x)
1/(2*sqrt(x))
```

See also:

Cbrt (page 1143)

fdiff(*argindex=1*)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**exp2**(*arg*)

Represents the exponential function with base two.

Explanation

The benefit of using exp2(x) over 2**x is that the latter is not as efficient under finite precision arithmetic.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import exp2
>>> exp2(2).evalf() == 4
True
>>> exp2(x).diff(x)
log(2)*exp2(x)
```

See also:

log2 (page 1147)

fdiff(*argindex=1*)
Returns the first derivative of this function.

class sympy.codegen.cfunctions.**expm1**(*arg*)
Represents the exponential function minus one.

Explanation

The benefit of using expm1(x) over exp(x) - 1 is that the latter is prone to cancellation under finite precision arithmetic when x is close to zero.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import expm1
>>> '%.0e' % expm1(1e-99).evalf()
'1e-99'
>>> from math import exp
>>> exp(1e-99) - 1
0.0
>>> expm1(x).diff(x)
exp(x)
```

See also:

log1p (page 1147)

fdiff(*argindex=1*)
Returns the first derivative of this function.

class sympy.codegen.cfunctions.**fma**(**args*)
Represents "fused multiply add".

Explanation

The benefit of using `fma(x, y, z)` over `x*y + z` is that, under finite precision arithmetic, the former is supported by special instructions on some CPUs.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.codegen.cfunctions import fma
>>> fma(x, y, z).diff(x)
y
```

fdiff(*argindex=1*)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**hypot**(**args*)

Represents the hypotenuse function.

Explanation

The hypotenuse function is provided by e.g. the math library in the C99 standard, hence one may want to represent the function symbolically when doing code-generation.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.codegen.cfunctions import hypot
>>> hypot(3, 4).evalf() == 5
True
>>> hypot(x, y)
hypot(x, y)
>>> hypot(x, y).diff(x)
x/hypot(x, y)
```

fdiff(*argindex=1*)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log10**(*arg*)

Represents the logarithm function with base ten.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log10
>>> log10(100).evalf() == 2
True
>>> log10(x).diff(x)
1/(x*log(10))
```

See also:

log2 (page 1147)

fdiff(*argindex=1*)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log1p**(*arg*)

Represents the natural logarithm of a number plus one.

Explanation

The benefit of using `log1p(x)` over `log(x + 1)` is that the latter is prone to cancellation under finite precision arithmetic when x is close to zero.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log1p
>>> from sympy import expand_log
>>> '%.0e' % expand_log(log1p(1e-99)).evalf()
'1e-99'
>>> from math import log
>>> log(1 + 1e-99)
0.0
>>> log1p(x).diff(x)
1/(x + 1)
```

See also:

expm1 (page 1145)

fdiff(*argindex=1*)

Returns the first derivative of this function.

class sympy.codegen.cfunctions.**log2**(*arg*)

Represents the logarithm function with base two.

Explanation

The benefit of using log2(x) over log(x)/log(2) is that the latter is not as efficient under finite precision arithmetic.

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cfunctions import log2
>>> log2(4).evalf() == 2
True
>>> log2(x).diff(x)
1/(x*log(2))
```

See also:

exp2 (page 1144), *log10* (page 1146)

fdiff(*argindex=1*)

Returns the first derivative of this function.

C specific AST nodes (sympy.codegen.cnodes)

AST nodes specific to the C family of languages

class sympy.codegen.cnodes.**CommaOperator**(**args*)

Represents the comma operator in C

class sympy.codegen.cnodes.**Label**(**args*, ***kwargs*)

Label for use with e.g. goto statement.

Examples

```
>>> from sympy import ccode, Symbol
>>> from sympy.codegen.cnodes import Label, PreIncrement
>>> print(ccode(Label('foo')))
foo:
>>> print(ccode(Label('bar', [PreIncrement(Symbol('a'))])))
bar:
++(a);
```

class sympy.codegen.cnodes.**PostDecrement**(**args*)

Represents the post-decrement operator

class sympy.codegen.cnodes.**PostIncrement**(**args*)

Represents the post-increment operator

class sympy.codegen.cnodes.**PreDecrement**(**args*)

Represents the pre-decrement operator

Examples

```
>>> from sympy.abc import x
>>> from sympy.codegen.cnodes import PreDecrement
>>> from sympy import ccode
>>> ccode(PreDecrement(x))
'--(x)'
```

class sympy.codegen.cnodes.**PreIncrement**(**args*)

Represents the pre-increment operator

sympy.codegen.cnodes.**alignof**(*arg*)

Generate of FunctionCall instance for calling 'alignof'

class sympy.codegen.cnodes.**goto**(**args*, ***kwargs*)

Represents goto in C

sympy.codegen.cnodes.**sizeof**(*arg*)

Generate of FunctionCall instance for calling 'sizeof'

Examples

```
>>> from sympy.codegen.ast import real
>>> from sympy.codegen.cnodes import sizeof
>>> from sympy import ccode
>>> ccode(sizeof(real))
'sizeof(double)'
```


class sympy.codegen.cnodes.**struct**(**args*, ***kwargs*)

Represents a struct in C

class sympy.codegen.cnodes.**union**(**args*, ***kwargs*)

Represents a union in C

C++ specific AST nodes (sympy.codegen.cxxnodes)

AST nodes specific to C++.

class sympy.codegen.cxxnodes.**using**(**args*, ***kwargs*)

Represents a 'using' statement in C++

Fortran specific AST nodes (sympy.codegen.fnodes)

AST nodes specific to Fortran.

The functions defined in this module allows the user to express functions such as `dsign` as a SymPy function for symbolic manipulation.

class sympy.codegen.fnodes.**ArrayConstructor**(**args*, ***kwargs*)

Represents an array constructor.

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import ArrayConstructor
>>> ac = ArrayConstructor([1, 2, 3])
>>> fcode(ac, standard=95, source_format='free')
'(/1, 2, 3/)'
>>> fcode(ac, standard=2003, source_format='free')
'[1, 2, 3]'
```

class sympy.codegen.fnodes.**Do**(**args*, ***kwargs*)

Represents a Do loop in in Fortran.

Examples

```
>>> from sympy import fcode, symbols
>>> from sympy.codegen.ast import aug_assign, Print
>>> from sympy.codegen.fnodes import Do
>>> i, n = symbols('i n', integer=True)
>>> r = symbols('r', real=True)
>>> body = [aug_assign(r, '+', 1/i), Print([i, r])]
>>> do1 = Do(body, i, 1, n)
>>> print(fcode(do1, source_format='free'))
do i = 1, n
    r = r + 1d0/i
    print *, i, r
end do
>>> do2 = Do(body, i, 1, n, 2)
>>> print(fcode(do2, source_format='free'))
do i = 1, n, 2
    r = r + 1d0/i
    print *, i, r
end do
```

class sympy.codegen.fnodes.**Extent**(**args*)

Represents a dimension extent.

Examples

```
>>> from sympy.codegen.fnodes import Extent
>>> e = Extent(-3, 3)  # -3, -2, -1, 0, 1, 2, 3
>>> from sympy import fcode
>>> fcode(e, source_format='free')
'-3:3'
>>> from sympy.codegen.ast import Variable, real
>>> from sympy.codegen.fnodes import dimension, intent_out
>>> dim = dimension(e, e)
>>> arr = Variable('x', real, attrs=[dim, intent_out])
>>> fcode(arr.as_Declaration(), source_format='free', standard=2003)
'real*8, dimension(-3:3, -3:3), intent(out) :: x'
```

class sympy.codegen.fnodes.**FortranReturn**(**args*, ***kwargs*)

AST node explicitly mapped to a fortran "return".

Explanation

Because a return statement in fortran is different from C, and in order to aid reuse of our codegen ASTs the ordinary .codegen.ast.Return is interpreted as assignment to the result variable of the function. If one for some reason needs to generate a fortran RETURN statement, this node should be used.

Examples

```
>>> from sympy.codegen.fnodes import FortranReturn
>>> from sympy import fcode
>>> fcode(FortranReturn('x'))
'       return x'
```

class sympy.codegen.fnodes.**GoTo**(**args*, ***kwargs*)

Represents a goto statement in Fortran

Examples

```
>>> from sympy.codegen.fnodes import GoTo
>>> go = GoTo([10, 20, 30], 'i')
>>> from sympy import fcode
>>> fcode(go, source_format='free')
'go to (10, 20, 30), i'
```


class sympy.codegen.fnodes.**ImpliedDoLoop**(**args*, ***kwargs*)

Represents an implied do loop in Fortran.

Examples

```
>>> from sympy import Symbol, fcode
>>> from sympy.codegen.fnodes import ImpliedDoLoop, ArrayConstructor
>>> i = Symbol('i', integer=True)
>>> idl = ImpliedDoLoop(i**3, i, -3, 3, 2)  # -27, -1, 1, 27
>>> ac = ArrayConstructor([-28, idl, 28]) # -28, -27, -1, 1, 27, 28
>>> fcode(ac, standard=2003, source_format='free')
'[-28, (i**3, i = -3, 3, 2), 28]'
```

class sympy.codegen.fnodes.**Module**(**args*, ***kwargs*)

Represents a module in Fortran.

Examples

```
>>> from sympy.codegen.fnodes import Module
>>> from sympy import fcode
>>> print(fcode(Module('signallib', ['implicit none'], []), source_
↪format='free'))
module signallib
implicit none

contains


end module
```

class sympy.codegen.fnodes.**Program**(**args*, ***kwargs*)

Represents a 'program' block in Fortran.

Examples

```
>>> from sympy.codegen.ast import Print
>>> from sympy.codegen.fnodes import Program
>>> prog = Program('myprogram', [Print([42])])
>>> from sympy import fcode
>>> print(fcode(prog, source_format='free'))
program myprogram
    print *, 42
end program
```

class sympy.codegen.fnodes.**Subroutine**(**args*, ***kwargs*)

Represents a subroutine in Fortran.

Examples

```
>>> from sympy import fcode, symbols
>>> from sympy.codegen.ast import Print
>>> from sympy.codegen.fnodes import Subroutine
>>> x, y = symbols('x y', real=True)
>>> sub = Subroutine('mysub', [x, y], [Print([x**2 + y**2, x*y])])
>>> print(fcode(sub, source_format='free', standard=2003))
subroutine mysub(x, y)
real*8 :: x
real*8 :: y
print *, x**2 + y**2, x*y
end subroutine
```

class sympy.codegen.fnodes.**SubroutineCall**(**args*, ***kwargs*)

Represents a call to a subroutine in Fortran.

Examples

```
>>> from sympy.codegen.fnodes import SubroutineCall
>>> from sympy import fcode
>>> fcode(SubroutineCall('mysub', 'x y'.split()))
'       call mysub(x, y)'
```

sympy.codegen.fnodes.**allocated**(*array*)

Creates an AST node for a function call to Fortran's "allocated(…)"

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import allocated
>>> alloc = allocated('x')
>>> fcode(alloc, source_format='free')
'allocated(x)'
```

sympy.codegen.fnodes.**array**(*symbol, dim, intent=None, *, attrs=(), value=None, type=None*)

Convenience function for creating a Variable instance for a Fortran array.

Parameters

symbol : symbol

dim : Attribute or iterable

If dim is an `Attribute` it need to have the name 'dimension'. If it is not an `Attribute`, then it is passsed to *dimension()* (page 1154) as `*dim`

intent : str

One of: 'in', 'out', 'inout' or None

****kwargs:**

Keyword arguments for `Variable` ('type' & 'value')

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.ast import integer, real
>>> from sympy.codegen.fnodes import array
>>> arr = array('a', '*', 'in', type=integer)
>>> print(fcode(arr.as_Declaration(), source_format='free', standard=2003))
integer*4, dimension(*), intent(in) :: a
>>> x = array('x', [3, ':', ':'], intent='out', type=real)
>>> print(fcode(x.as_Declaration(value=1), source_format='free', standard=2003))
real*8, dimension(3, :, :), intent(out) :: x = 1
```

sympy.codegen.fnodes.**bind_C**(*name=None*)

Creates an Attribute bind_C with a name.

Parameters

name : str

Examples

```
>>> from sympy import fcode, Symbol
>>> from sympy.codegen.ast import FunctionDefinition, real, Return
>>> from sympy.codegen.fnodes import array, sum_, bind_C
>>> a = Symbol('a', real=True)
>>> s = Symbol('s', integer=True)
>>> arr = array(a, dim=[s], intent='in')
>>> body = [Return((sum_(a**2)/s)**.5)]
>>> fd = FunctionDefinition(real, 'rms', [arr, s], body, attrs=[bind_C(
↪'rms')])
>>> print(fcode(fd, source_format='free', standard=2003))
real*8 function rms(a, s) bind(C, name="rms")
real*8, dimension(s), intent(in) :: a
integer*4 :: s
rms = sqrt(sum(a**2)/s)
end function
```


class sympy.codegen.fnodes.**cmplx**(**args*)

Fortran complex conversion function.

sympy.codegen.fnodes.**dimension**(**args*)

Creates a 'dimension' Attribute with (up to 7) extents.

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import dimension, intent_in
>>> dim = dimension('2', ':')  # 2 rows, runtime determined number of␣
↪columns
>>> from sympy.codegen.ast import Variable, integer
>>> arr = Variable('a', integer, attrs=[dim, intent_in])
>>> fcode(arr.as_Declaration(), source_format='free', standard=2003)
'integer*4, dimension(2, :), intent(in) :: a'
```

class sympy.codegen.fnodes.**dsign**(**args*)

Fortran sign intrinsic for double precision arguments.

class sympy.codegen.fnodes.**isign**(**args*)

Fortran sign intrinsic for integer arguments.

class sympy.codegen.fnodes.**kind**(**args*)

Fortran kind function.

sympy.codegen.fnodes.**lbound**(*array, dim=None, kind=None*)

Creates an AST node for a function call to Fortran's "lbound(…)"

Parameters

array : Symbol or String

dim : expr

kind : expr

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import lbound
>>> lb = lbound('arr', dim=2)
>>> fcode(lb, source_format='free')
'lbound(arr, 2)'
```

class sympy.codegen.fnodes.**literal_dp**(*num*, *dps=None*, *precision=None*)

Fortran double precision real literal

class sympy.codegen.fnodes.**literal_sp**(*num*, *dps=None*, *precision=None*)

Fortran single precision real literal

class sympy.codegen.fnodes.**merge**(**args*)

Fortran merge function

sympy.codegen.fnodes.**reshape**(*source*, *shape*, *pad=None*, *order=None*)

Creates an AST node for a function call to Fortran's "reshape(…)"

Parameters

source : Symbol or String

shape : ArrayExpr

sympy.codegen.fnodes.**shape**(*source*, *kind=None*)

Creates an AST node for a function call to Fortran's "shape(…)"

Parameters

source : Symbol or String

kind : expr

Examples

```
>>> from sympy import fcode
>>> from sympy.codegen.fnodes import shape
>>> shp = shape('x')
>>> fcode(shp, source_format='free')
'shape(x)'
```

sympy.codegen.fnodes.**size**(*array*, *dim=None*, *kind=None*)

Creates an AST node for a function call to Fortran's "size(…)"

Examples

```
>>> from sympy import fcode, Symbol
>>> from sympy.codegen.ast import FunctionDefinition, real, Return
>>> from sympy.codegen.fnodes import array, sum_, size
>>> a = Symbol('a', real=True)
>>> body = [Return((sum_(a**2)/size(a))**.5)]
>>> arr = array(a, dim=[':'], intent='in')
>>> fd = FunctionDefinition(real, 'rms', [arr], body)
>>> print(fcode(fd, source_format='free', standard=2003))
real*8 function rms(a)
real*8, dimension(:), intent(in) :: a
rms = sqrt(sum(a**2)*1d0/size(a))
end function
```

class sympy.codegen.fnodes.**use**(**args, **kwargs*)

Represents a use statement in Fortran.

Examples

```
>>> from sympy.codegen.fnodes import use
>>> from sympy import fcode
>>> fcode(use('signallib'), source_format='free')
'use signallib'
>>> fcode(use('signallib', [('metric', 'snr')]), source_format='free')
'use signallib, metric => snr'
>>> fcode(use('signallib', only=['snr', 'convolution2d']), source_format=
↪'free')
'use signallib, only: snr, convolution2d'
```

class sympy.codegen.fnodes.**use_rename**(**args, **kwargs*)

Represents a renaming in a use statement in Fortran.

Examples

```
>>> from sympy.codegen.fnodes import use_rename, use
>>> from sympy import fcode
>>> ren = use_rename("thingy", "convolution2d")
>>> print(fcode(ren, source_format='free'))
thingy => convolution2d
>>> full = use('signallib', only=['snr', ren])
>>> print(fcode(full, source_format='free'))
use signallib, only: snr, thingy => convolution2d
```