sympy.interactive.session.**init_python_session**()

Construct new Python session.

sympy.interactive.session.**init_session**(*ipython=None*, *pretty_print=True*, *order=None*, *use_unicode=None*, *use_latex=None*, *quiet=False*, *auto_symbols=False*, *auto_int_to_Integer=False*, *str_printer=None*, *pretty_printer=None*, *latex_printer=None*, *argv=[]*)

Initialize an embedded IPython or Python session. The IPython session is initiated with the –pylab option, without the numpy imports, so that matplotlib plotting can be interactive.

Parameters

pretty_print: boolean

If True, use pretty_print to stringify; if False, use sstrrepr to stringify.

order: string or None

There are a few different settings for this parameter: lex (default), which is lexographic order; grlex, which is graded lexographic order; grevlex, which is reversed graded lexographic order; old, which is used for compatibility reasons and for long expressions; None, which sets it to lex.

use_unicode: boolean or None

If True, use unicode characters; if False, do not use unicode characters.

use_latex: boolean or None

If True, use latex rendering if IPython GUI's; if False, do not use latex rendering.

quiet: boolean

If True, init_session will not print messages regarding its status; if False, init_session will print messages regarding its status.

auto_symbols: boolean

If True, IPython will automatically create symbols for you. If False, it will not. The default is False.

auto_int_to_Integer: boolean

If True, IPython will automatically wrap int literals with Integer, so that things like 1/2 give Rational(1, 2). If False, it will not. The default is False.

ipython: boolean or None

If True, printing will initialize for an IPython console; if False, printing will initialize for a normal console; The default is None, which automatically determines whether we are in an ipython instance or not.

str_printer: function, optional, default=None

A custom string printer function. This should mimic sympy.printing.sstrrepr().

pretty_printer: function, optional, default=None

A custom pretty printer. This should mimic sympy.printing.pretty().

latex_printer: function, optional, default=None

A custom LaTeX printer. This should mimic sympy.printing.latex() This should mimic sympy.printing.latex().

argv: list of arguments for IPython

See sympy.bin.isympy for options that can be used to initialize IPython.

Examples

```
>>> from sympy import init_session, Symbol, sin, sqrt
>>> sin(x)
NameError: name 'x' is not defined
>>> init_session()
>>> sin(x)
sin(x)
>>> sqrt(5)
  ___
\/ 5
>>> init_session(pretty_print=False)
>>> sqrt(5)
sqrt(5)
>>> y + x + y**2 + x**2
x**2 + x + y**2 + y
>>> init_session(order='grlex')
>>> y + x + y**2 + x**2
x**2 + y**2 + x + y
>>> init_session(order='grevlex')
>>> y * x**2 + x * y**2
x**2*y + x*y**2
>>> init_session(order='old')
>>> x**2 + y**2 + x + y
x + y + x**2 + y**2
>>> theta = Symbol('theta')
>>> theta
theta
>>> init_session(use_unicode=True)
>>> theta
θ
```

See also:

sympy.interactive.printing.init_printing **(page 2119)**
for examples and the rest of the parameters.

sympy.interactive.session.**int_to_Integer**(*s*)

Wrap integer literals with Integer.

This is based on the decistmt example from http://docs.python.org/library/tokenize.html.

Only integer literals are converted. Float literals are left alone.

Examples

```
>>> from sympy import Integer # noqa: F401
>>> from sympy.interactive.session import int_to_Integer
>>> s = '1.2 + 1/2 - 0x12 + a1'
>>> int_to_Integer(s)
'1.2 +Integer (1 )/Integer (2 )-Integer (0x12 )+a1 '
>>> s = 'print (1/2)'
>>> int_to_Integer(s)
'print (Integer (1 )/Integer (2 ))'
>>> exec(s)
0.5
>>> exec(int_to_Integer(s))
1/2
```

Printing

Tools for setting up printing in interactive sessions.

sympy.interactive.printing.**init_printing**(*pretty_print=True*, *order=None*, *use_unicode=None*, *use_latex=None*, *wrap_line=None*, *num_columns=None*, *no_global=False*, *ip=None*, *euler=False*, *forecolor=None*, *backcolor='Transparent'*, *fontsize='10pt'*, *latex_mode='plain'*, *print_builtin=True*, *str_printer=None*, *pretty_printer=None*, *latex_printer=None*, *scale=1.0*, ***settings*)

Initializes pretty-printer depending on the environment.

Parameters

pretty_print : bool, default=True

If `True`, use *pretty_print()* (page 2139) to stringify or the provided pretty printer; if `False`, use *sstrrepr()* (page 2178) to stringify or the provided string printer.

order : string or None, default='lex'

There are a few different settings for this parameter: `'lex'` (default), which is lexographic order; `'grlex'`, which is graded lexographic order; `'grevlex'`, which is reversed graded lexographic order; `'old'`, which is used for compatibility reasons and for long expressions; `None`, which sets it to lex.

use_unicode : bool or None, default=None

If `True`, use unicode characters; if `False`, do not use unicode characters; if `None`, make a guess based on the environment.

use_latex : string, bool, or None, default=None

If `True`, use default LaTeX rendering in GUI interfaces (png and mathjax); if `False`, do not use LaTeX rendering; if `None`, make a guess based on the environment; if `'png'`, enable LaTeX rendering with an external LaTeX compiler, falling back to matplotlib if external compilation fails; if `'matplotlib'`, enable LaTeX rendering with matplotlib; if `'mathjax'`, enable LaTeX text generation, for example MathJax rendering in IPython notebook or text rendering in LaTeX documents; if `'svg'`, enable LaTeX rendering with an external latex compiler, no fallback

wrap_line : bool

If True, lines will wrap at the end; if False, they will not wrap but continue as one line. This is only relevant if `pretty_print` is True.

num_columns : int or None, default=None

If `int`, number of columns before wrapping is set to num_columns; if `None`, number of columns before wrapping is set to terminal width. This is only relevant if `pretty_print` is `True`.

no_global : bool, default=False

If `True`, the settings become system wide; if `False`, use just for this console/session.

ip : An interactive console

This can either be an instance of IPython, or a class that derives from code.InteractiveConsole.

euler : bool, optional, default=False

Loads the euler package in the LaTeX preamble for handwritten style fonts (http://www.ctan.org/pkg/euler).

forecolor : string or None, optional, default=None

DVI setting for foreground color. `None` means that either `'Black'`, `'White'`, or `'Gray'` will be selected based on a guess of the IPython terminal color setting. See notes.

backcolor : string, optional, default='Transparent'

DVI setting for background color. See notes.

fontsize : string or int, optional, default='10pt'

A font size to pass to the LaTeX documentclass function in the preamble. Note that the options are limited by the documentclass. Consider using scale instead.

latex_mode : string, optional, default='plain'

The mode used in the LaTeX printer. Can be one of: `{'inline'|'plain'|'equation'|'equation*'}`.

print_builtin : boolean, optional, default=True

If `True` then floats and integers will be printed. If `False` the printer will only print SymPy types.

str_printer : function, optional, default=None

A custom string printer function. This should mimic *sstrrepr()* (page 2178).

pretty_printer : function, optional, default=None

A custom pretty printer. This should mimic *pretty()* (page 2139).

latex_printer : function, optional, default=None

A custom LaTeX printer. This should mimic *latex()* (page 2170).

scale : float, optional, default=1.0

Scale the LaTeX output when using the `'png'` or `'svg'` backends. Useful for high dpi screens.

settings :

Any additional settings for the `latex` and `pretty` commands can be used to fine-tune the output.

Examples

```
>>> from sympy.interactive import init_printing
>>> from sympy import Symbol, sqrt
>>> from sympy.abc import x, y
>>> sqrt(5)
sqrt(5)
>>> init_printing(pretty_print=True)
>>> sqrt(5)
  ___
\/ 5
>>> theta = Symbol('theta')
>>> init_printing(use_unicode=True)
>>> theta
\u03b8
>>> init_printing(use_unicode=False)
>>> theta
theta
>>> init_printing(order='lex')
>>> str(y + x + y**2 + x**2)
x**2 + x + y**2 + y
>>> init_printing(order='grlex')
>>> str(y + x + y**2 + x**2)
x**2 + x + y**2 + y
>>> init_printing(order='grevlex')
>>> str(y * x**2 + x * y**2)
x**2*y + x*y**2
>>> init_printing(order='old')
>>> str(x**2 + y**2 + x + y)
x**2 + x + y**2 + y
>>> init_printing(num_columns=10)
```

(continues on next page)

(continued from previous page)

```
>>> x**2 + x + y**2 + y
x + y +
x**2 + y**2
```

Notes

The foreground and background colors can be selected when using `'png'` or `'svg'` LaTeX rendering. Note that before the `init_printing` command is executed, the LaTeX rendering is handled by the IPython console and not SymPy.

The colors can be selected among the 68 standard colors known to `dvips`, for a list see [R553]. In addition, the background color can be set to `'Transparent'` (which is the default value).

When using the `'Auto'` foreground color, the guess is based on the `colors` variable in the IPython console, see [R554]. Hence, if that variable is set correctly in your IPython console, there is a high chance that the output will be readable, although manual settings may be needed.

See also:

sympy.printing.latex (page 2170), *sympy.printing.pretty* (page 2139)

References

[R553], [R554]

Parsing

Parsing Functions Reference

sympy.parsing.sympy_parser.**parse_expr**(*s: str, local_dict: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, transformations: ~typing.Union[~typing.Tuple[~typing.Callable[[~typing.List[~typ str]], ~typing.Dict[str, ~typing.Any], ~typing.Dict[str, ~typing.Any]], ~typing.List[~typing.Tuple[int, str]]], ...], str] = (<function lambda_notation>, <function auto_symbol>, <function repeated_decimals>, <function auto_number>, <function factorial_notation>), global_dict: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, evaluate=True*)

Converts the string `s` to a SymPy expression, in `local_dict`

Parameters

s : str

The string to parse.

local_dict : dict, optional

A dictionary of local variables to use when parsing.

global_dict : dict, optional

A dictionary of global variables. By default, this is initialized with `from sympy import *`; provide this parameter to override this behavior (for instance, to parse `"Q & S"`).

transformations : tuple or str

A tuple of transformation functions used to modify the tokens of the parsed expression before evaluation. The default transformations convert numeric literals into their SymPy equivalents, convert undefined variables into SymPy symbols, and allow the use of standard mathematical factorial notation (e.g. `x!`). Selection via string is available (see below).

evaluate : bool, optional

When False, the order of the arguments will remain as they were in the string and automatic simplification that would normally occur is suppressed. (see examples)

Examples

```
>>> from sympy.parsing.sympy_parser import parse_expr
>>> parse_expr("1/2")
1/2
>>> type(_)
<class 'sympy.core.numbers.Half'>
>>> from sympy.parsing.sympy_parser import standard_transformations,\
... implicit_multiplication_application
>>> transformations = (standard_transformations +
...     (implicit_multiplication_application,))
>>> parse_expr("2x", transformations=transformations)
2*x
```

When evaluate=False, some automatic simplifications will not occur:

```
>>> parse_expr("2**3"), parse_expr("2**3", evaluate=False)
(8, 2**3)
```

In addition the order of the arguments will not be made canonical. This feature allows one to tell exactly how the expression was entered:

```
>>> a = parse_expr('1 + x', evaluate=False)
>>> b = parse_expr('x + 1', evaluate=0)
>>> a == b
False
>>> a.args
(1, x)
>>> b.args
(x, 1)
```

Note, however, that when these expressions are printed they will appear the same:

```
>>> assert str(a) == str(b)
```

As a convenience, transformations can be seen by printing `transformations`:

```
>>> from sympy.parsing.sympy_parser import transformations
```

```
>>> print(transformations)
0: lambda_notation
1: auto_symbol
2: repeated_decimals
3: auto_number
4: factorial_notation
5: implicit_multiplication_application
6: convert_xor
7: implicit_application
8: implicit_multiplication
9: convert_equals_signs
10: function_exponentiation
11: rationalize
```

The `T` object provides a way to select these transformations:

```
>>> from sympy.parsing.sympy_parser import T
```

If you print it, you will see the same list as shown above.

```
>>> str(T) == str(transformations)
True
```

Standard slicing will return a tuple of transformations:

```
>>> T[:5] == standard_transformations
True
```

So `T` can be used to specify the parsing transformations:

```
>>> parse_expr("2x", transformations=T[:5])
Traceback (most recent call last):
...
SyntaxError: invalid syntax
>>> parse_expr("2x", transformations=T[:6])
2*x
>>> parse_expr('.3', transformations=T[3, 11])
3/10
>>> parse_expr('.3x', transformations=T[:])
3*x/10
```

As a further convenience, strings 'implicit' and 'all' can be used to select 0-5 and all the transformations, respectively.

```
>>> parse_expr('.3x', transformations='all')
3*x/10
```

See also:

stringify_expr (page 2125), *eval_expr* (page 2125), *standard_transformations* (page 2126), *implicit_multiplication_application* (page 2128)

sympy.parsing.sympy_parser.**stringify_expr**(*s: str, local_dict: Dict[str, Any], global_dict: Dict[str, Any], transformations: Tuple[Callable[[List[Tuple[int, str]], Dict[str, Any], Dict[str, Any]], List[Tuple[int, str]]], ...]*) → str

Converts the string `s` to Python code, in `local_dict`

Generally, `parse_expr` should be used.

sympy.parsing.sympy_parser.**eval_expr**(*code*, *local_dict: Dict[str, Any]*, *global_dict: Dict[str, Any]*)

Evaluate Python code generated by `stringify_expr`.

Generally, `parse_expr` should be used.

sympy.parsing.maxima.**parse_maxima**(*str*, *globals=None*, *name_dict={}*)

sympy.parsing.mathematica.**parse_mathematica**(*s*)

Translate a string containing a Wolfram Mathematica expression to a SymPy expression.

If the translator is unable to find a suitable SymPy expression, the `FullForm` of the Mathematica expression will be output, using SymPy `Function` objects as nodes of the syntax tree.

Examples

```
>>> from sympy.parsing.mathematica import parse_mathematica
>>> parse_mathematica("Sin[x]^2 Tan[y]")
sin(x)**2*tan(y)
>>> e = parse_mathematica("F[7,5,3]")
>>> e
F(7, 5, 3)
>>> from sympy import Function, Max, Min
>>> e.replace(Function("F"), lambda *x: Max(*x)*Min(*x))
21
```

Both standard input form and Mathematica full form are supported:

```
>>> parse_mathematica("x*(a + b)")
x*(a + b)
>>> parse_mathematica("Times[x, Plus[a, b]]")
x*(a + b)
```

To get a matrix from Wolfram's code:

```
>>> m = parse_mathematica("{{a, b}, {c, d}}")
>>> m
((a, b), (c, d))
>>> from sympy import Matrix
```

(continues on next page)

(continued from previous page)

```
>>> Matrix(m)
Matrix([
[a, b],
[c, d]])
```

If the translation into equivalent SymPy expressions fails, an SymPy expression equivalent to Wolfram Mathematica's "FullForm" will be created:

```
>>> parse_mathematica("x_.")
Optional(Pattern(x, Blank()))
>>> parse_mathematica("Plus @@ {x, y, z}")
Apply(Plus, (x, y, z))
>>> parse_mathematica("f[x_, 3] := x^3 /; x > 0")
SetDelayed(f(Pattern(x, Blank()), 3), Condition(x**3, x > 0))
```

Parsing Transformations Reference

A transformation is a function that accepts the arguments `tokens`, `local_dict`, `global_dict` and returns a list of transformed tokens. They can be used by passing a list of functions to *parse_expr()* (page 2122) and are applied in the order given.

sympy.parsing.sympy_parser.**standard_transformations: Tuple[Callable[[List[Tuple[int, str]], Dict[str, Any], Dict[str, Any]], List[Tuple[int, str]]], ...] = (<function lambda_notation>, <function auto_symbol>, <function repeated_decimals>, <function auto_number>, <function factorial_notation>)**

Standard transformations for *parse_expr()* (page 2122). Inserts calls to *Symbol* (page 976), *Integer* (page 987), and other SymPy datatypes and allows the use of standard factorial notation (e.g. `x!`).

sympy.parsing.sympy_parser.**split_symbols**(*tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*)

Splits symbol names for implicit multiplication.

Intended to let expressions like `xyz` be parsed as `x*y*z`. Does not split Greek character names, so `theta` will *not* become `t*h*e*t*a`. Generally this should be used with `implicit_multiplication`.

sympy.parsing.sympy_parser.**split_symbols_custom**(*predicate: Callable[[str], bool]*)

Creates a transformation that splits symbol names.

`predicate` should return True if the symbol name is to be split.

For instance, to retain the default behavior but avoid splitting certain symbol names, a predicate like this would work:

```
>>> from sympy.parsing.sympy_parser import (parse_expr, _token_
↪splittable,
... standard_transformations, implicit_multiplication,
... split_symbols_custom)
>>> def can_split(symbol):
...     if symbol not in ('list', 'of', 'unsplittable', 'names'):
...             return _token_splittable(symbol)
```

(continues on next page)

(continued from previous page)

```
...     return False
...
>>> transformation = split_symbols_custom(can_split)
>>> parse_expr('unsplittable', transformations=standard_transformations +
... (transformation, implicit_multiplication))
unsplittable
```

sympy.parsing.sympy_parser.**implicit_multiplication**(*tokens: List[Tuple[int, str]]*, *local_dict: Dict[str, Any]*, *global_dict: Dict[str, Any]*) → List[Tuple[int, str]]

Makes the multiplication operator optional in most cases.

Use this before *implicit_application()* (page 2127), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication)
>>> transformations = standard_transformations + (implicit_
↪multiplication,)
>>> parse_expr('3 x y', transformations=transformations)
3*x*y
```

sympy.parsing.sympy_parser.**implicit_application**(*tokens: List[Tuple[int, str]]*, *local_dict: Dict[str, Any]*, *global_dict: Dict[str, Any]*) → List[Tuple[int, str]]

Makes parentheses optional in some cases for function calls.

Use this after *implicit_multiplication()* (page 2127), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_application)
>>> transformations = standard_transformations + (implicit_application,)
>>> parse_expr('cot z + csc z', transformations=transformations)
cot(z) + csc(z)
```

sympy.parsing.sympy_parser.**function_exponentiation**(*tokens: List[Tuple[int, str]]*, *local_dict: Dict[str, Any]*, *global_dict: Dict[str, Any]*)

Allows functions to be exponentiated, e.g. `cos**2(x)`.

Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, function_exponentiation)
>>> transformations = standard_transformations + (function_
↪exponentiation,)
>>> parse_expr('sin**4(x)', transformations=transformations)
sin(x)**4
```

sympy.parsing.sympy_parser.**implicit_multiplication_application**(*result: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*) → List[Tuple[int, str]]

Allows a slightly relaxed syntax.

- Parentheses for single-argument method calls are optional.
- Multiplication is implicit.
- Symbol names can be split (i.e. spaces are not needed between symbols).
- Functions can be exponentiated.

Examples

```
>>> from sympy.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication_application)
>>> parse_expr("10sin**2 x**2 + 3xyz + tan theta",
... transformations=(standard_transformations +
... (implicit_multiplication_application,)))
3*x*y*z + 10*sin(x**2)**2 + tan(theta)
```

sympy.parsing.sympy_parser.**rationalize**(*tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*)

Converts floats into `Rational`. Run AFTER `auto_number`.

sympy.parsing.sympy_parser.**convert_xor**(*tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*)

Treats XOR, `^`, as exponentiation, `**`.

These are included in *sympy.parsing.sympy_parser.standard_transformations* (page 2126) and generally don't need to be manually added by the user.

sympy.parsing.sympy_parser.**lambda_notation**(*tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*)

Substitutes "lambda" with its SymPy equivalent Lambda(). However, the conversion does not take place if only "lambda" is passed because that is a syntax error.

sympy.parsing.sympy_parser.**auto_symbol**(*tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*)

Inserts calls to `Symbol`/`Function` for undefined variables.

sympy.parsing.sympy_parser.**repeated_decimals**(*tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*)

Allows 0.2[1] notation to represent the repeated decimal 0.2111… (19/90)

Run this before auto_number.

sympy.parsing.sympy_parser.**auto_number**(*tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*)

Converts numeric literals to use SymPy equivalents.

Complex numbers use `I`, integer literals use `Integer`, and float literals use `Float`.

sympy.parsing.sympy_parser.**factorial_notation**(*tokens: List[Tuple[int, str]], local_dict: Dict[str, Any], global_dict: Dict[str, Any]*)

Allows standard notation for factorial.

Experimental LaTeX Parsing

LaTeX parsing was ported from latex2sympy. While functional and its API should remain stable, the parsing behavior or backend may change in future releases.

LaTeX Parsing Caveats

The current implementation is experimental. The behavior, parser backend and API might change in the future. Unlike some of the other parsers, LaTeX is designed as a *type-setting* language, not a *computer algebra system* and so can contain typographical conventions that might be interpreted multiple ways.

In its current definition, the parser will at times will fail to fully parse the expression, but not throw a warning:

```
parse_latex(r'x -')
```

Will simply find `x`. What is covered by this behavior will almost certainly change between releases, and become stricter, more relaxed, or some mix.

LaTeX Parsing Functions Reference

sympy.parsing.latex.**parse_latex**(*s*)

Converts the string `s` to a SymPy `Expr`

Parameters

s : str

The LaTeX string to parse. In Python source containing LaTeX, *raw strings* (denoted with `r"`, like this one) are preferred, as LaTeX makes liberal use of the \ character, which would trigger escaping in normal Python strings.

Examples

```
>>> from sympy.parsing.latex import parse_latex
>>> expr = parse_latex(r"\frac {1 + \sqrt {\a}} {\b}")
>>> expr
(sqrt(a) + 1)/b
>>> expr.evalf(4, subs=dict(a=5, b=2))
1.618
```

LaTeX Parsing Exceptions Reference

class sympy.parsing.latex.**LaTeXParsingError**

SymPy Expression Reference

class sympy.parsing.sym_expr.**SymPyExpression**(*source_code=None*, *mode=None*)

Class to store and handle SymPy expressions

This class will hold SymPy Expressions and handle the API for the conversion to and from different languages.

It works with the C and the Fortran Parser to generate SymPy expressions which are stored here and which can be converted to multiple language's source code.

Notes

The module and its API are currently under development and experimental and can be changed during development.

The Fortran parser does not support numeric assignments, so all the variables have been Initialized to zero.

The module also depends on external dependencies:

- LFortran which is required to use the Fortran parser
- Clang which is required for the C parser

Examples

Example of parsing C code:

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src = '''
... int a,b;
... float c = 2, d =4;
... '''
>>> a = SymPyExpression(src, 'c')
>>> a.return_expr()
[Declaration(Variable(a, type=intc)),
```

(continues on next page)

(continued from previous page)

```
Declaration(Variable(b, type=intc)),
Declaration(Variable(c, type=float32, value=2.0)),
Declaration(Variable(d, type=float32, value=4.0))]
```

An example of variable definiton:

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src2, 'f')
>>> p.convert_to_c()
['int a = 0', 'int b = 0', 'int c = 0', 'int d = 0', 'double p = 0.0',
 ↪'double q = 0.0', 'double r = 0.0', 'double s = 0.0']
```

An example of Assignment:

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src3 = '''
... integer :: a, b, c, d, e
... d = a + b - c
... e = b * d + c * e / a
... '''
>>> p = SymPyExpression(src3, 'f')
>>> p.convert_to_python()
['a = 0', 'b = 0', 'c = 0', 'd = 0', 'e = 0', 'd = a + b - c', 'e = b*d␣
 ↪+ c*e/a']
```

An example of function definition:

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src = '''
... integer function f(a,b)
... integer, intent(in) :: a, b
... integer :: r
... end function
... '''
>>> a = SymPyExpression(src, 'f')
>>> a.convert_to_python()
['def f(a, b):\n    f = 0\n    r = 0\n    return f']
```

convert_to_c()

Returns a list with the c source code for the SymPy expressions

Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src2, 'f')
>>> p.convert_to_c()
['int a = 0', 'int b = 0', 'int c = 0', 'int d = 0', 'double p = 0.0',
 ↪ 'double q = 0.0', 'double r = 0.0', 'double s = 0.0', 'c = a/b;',
 ↪'d = c/a;', 's = p/q;', 'r = q/p;']
```

convert_to_expr(*src_code*, *mode*)

Converts the given source code to SymPy Expressions

Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src3 = '''
... integer function f(a,b) result(r)
... integer, intent(in) :: a, b
... integer :: x
... r = a + b -x
... end function
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src3, 'f')
>>> p.return_expr()
[FunctionDefinition(integer, name=f, parameters=(Variable(a),␣
 ↪Variable(b)), body=CodeBlock(
Declaration(Variable(r, type=integer, value=0)),
Declaration(Variable(x, type=integer, value=0)),
Assignment(Variable(r), a + b - x),
Return(Variable(r))
))]
```

Attributes

src_code	(String) the source code or filename of the source code that is to be converted
mode: String	the mode to determine which parser is to be used according to the language of the source code f or F for Fortran c or C for C/C++

convert_to_fortran()

Returns a list with the fortran source code for the SymPy expressions

Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
... '''
>>> p = SymPyExpression(src2, 'f')
>>> p.convert_to_fortran()
['      integer*4 a', '      integer*4 b', '      integer*4 c', '
 ↪ integer*4 d', '      real*8 p', '      real*8 q', '      real*8 r',
 ↪ '      real*8 s', '      c = a/b', '      d = c/a', '      s = p/q
 ↪', '      r = q/p']
```

convert_to_python()

Returns a list with Python code for the SymPy expressions

Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src2 = '''
... integer :: a, b, c, d
... real :: p, q, r, s
... c = a/b
... d = c/a
... s = p/q
... r = q/p
... '''
>>> p = SymPyExpression(src2, 'f')
>>> p.convert_to_python()
['a = 0', 'b = 0', 'c = 0', 'd = 0', 'p = 0.0', 'q = 0.0', 'r = 0.0',
 ↪'s = 0.0', 'c = a/b', 'd = c/a', 's = p/q', 'r = q/p']
```

return_expr()

Returns the expression list

Examples

```
>>> from sympy.parsing.sym_expr import SymPyExpression
>>> src3 = '''
... integer function f(a,b)
... integer, intent(in) :: a, b
... integer :: r
... r = a+b
... f = r
... end function
... '''
>>> p = SymPyExpression()
>>> p.convert_to_expr(src3, 'f')
>>> p.return_expr()
[FunctionDefinition(integer, name=f, parameters=(Variable(a),␣
↪Variable(b)), body=CodeBlock(
Declaration(Variable(f, type=integer, value=0)),
Declaration(Variable(r, type=integer, value=0)),
Assignment(Variable(f), Variable(r)),
Return(Variable(f))
))]
```

Runtime Installation

The currently-packaged LaTeX parser backend is partially generated with ANTLR4, but to use the parser, you only need the `antlr4` Python package available.

Depending on your package manager, you can install the right package with, for example, `pip`:

```
$ pip install antlr4-python3-runtime==4.10
```

or `conda`:

```
$ conda install -c conda-forge antlr-python-runtime==4.10
```

The C parser depends on `clang` and the Fortran parser depends on `LFortran`. You can install these packages using:

```
$ conda install -c conda-forge lfortran clang
```

Printing

See the *Printing* (page 19) section in tutorial for introduction into printing.

This guide documents the printing system in SymPy and how it works internally.

Printer Class

Printing subsystem driver

SymPy's printing system works the following way: Any expression can be passed to a designated Printer who then is responsible to return an adequate representation of that expression.

The basic concept is the following:

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.

Which Method is Responsible for Printing?

The whole printing process is started by calling `.doprint(expr)` on the printer which you want to use. This method looks for an appropriate method which can print the given expression in the given style that the printer defines. While looking for the method, it follows these steps:

1. **Let the object print itself if it knows how.**

   The printer looks for a specific method in every object. The name of that method depends on the specific printer and is defined under `Printer.printmethod`. For example, StrPrinter calls `_sympystr` and LatexPrinter calls `_latex`. Look at the documentation of the printer that you want to use. The name of the method is specified there.

   This was the original way of doing printing in sympy. Every class had its own latex, mathml, str and repr methods, but it turned out that it is hard to produce a high quality printer, if all the methods are spread out that far. Therefore all printing code was combined into the different printers, which works great for built-in SymPy objects, but not that good for user defined classes where it is inconvenient to patch the printers.

2. **Take the best fitting method defined in the printer.**

   The printer loops through expr classes (class + its bases), and tries to dispatch the work to `_print_<EXPR_CLASS>`

   e.g., suppose we have the following class hierarchy:

```
    Basic
    |
    Atom
    |
    Number
    |
Rational
```

   then, for `expr=Rational(...)`, the Printer will try to call printer methods in the order as shown in the figure below:

```
p._print(expr)
|
|-- p._print_Rational(expr)
|
```

(continues on next page)

(continued from previous page)

```
|-- p._print_Number(expr)
|
|-- p._print_Atom(expr)
|
`-- p._print_Basic(expr)
```

if `._print_Rational` method exists in the printer, then it is called, and the result is returned back. Otherwise, the printer tries to call `._print_Number` and so on.

3. **As a fall-back use the emptyPrinter method for the printer.**

   As fall-back `self.emptyPrinter` will be called with the expression. If not defined in the Printer subclass this will be the same as `str(expr)`.

Example of Custom Printer

In the example below, we have a printer which prints the derivative of a function in a shorter form.

```
from sympy.core.symbol import Symbol
from sympy.printing.latex import LatexPrinter, print_latex
from sympy.core.function import UndefinedFunction, Function


class MyLatexPrinter(LatexPrinter):
    """Print derivative of a function of symbols in a shorter form.
    """
    def _print_Derivative(self, expr):
        function, *vars = expr.args
        if not isinstance(type(function), UndefinedFunction) or \
           not all(isinstance(i, Symbol) for i in vars):
            return super()._print_Derivative(expr)

        # If you want the printer to work correctly for nested
        # expressions then use self._print() instead of str() or latex().
        # See the example of nested modulo below in the custom printing
        # method section.
        return "{}_{{{}}}".format(
            self._print(Symbol(function.func.__name__)),
                        ''.join(self._print(i) for i in vars))


def print_my_latex(expr):
    """ Most of the printers define their own wrappers for print().
    These wrappers usually take printer settings. Our printer does not have
    any settings.
    """
    print(MyLatexPrinter().doprint(expr))


y = Symbol("y")
x = Symbol("x")
```

(continues on next page)

(continued from previous page)

```
f = Function("f")
expr = f(x, y).diff(x, y)

# Print the expression using the normal latex printer and our custom
# printer.
print_latex(expr)
print_my_latex(expr)
```

The output of the code above is:

```
\frac{\partial^{2}}{\partial x\partial y}  f{\left(x,y \right)}
f_{xy}
```

Example of Custom Printing Method

In the example below, the latex printing of the modulo operator is modified. This is done by overriding the method `_latex` of `Mod`.

```
>>> from sympy import Symbol, Mod, Integer, print_latex
```

```
>>> # Always use printer._print()
>>> class ModOp(Mod):
...     def _latex(self, printer):
...         a, b = [printer._print(i) for i in self.args]
...         return r"\operatorname{Mod}{\left(%s, %s\right)}" % (a, b)
```

Comparing the output of our custom operator to the builtin one:

```
>>> x = Symbol('x')
>>> m = Symbol('m')
>>> print_latex(Mod(x, m))
x \bmod m
>>> print_latex(ModOp(x, m))
\operatorname{Mod}{\left(x, m\right)}
```

Common mistakes

It's important to always use `self._print(obj)` to print subcomponents of an expression when customizing a printer. Mistakes include:

1. Using `self.doprint(obj)` instead:

```
>>> # This example does not work properly, as only the outermost call␣
 ↪may use
>>> # doprint.
>>> class ModOpModeWrong(Mod):
...     def _latex(self, printer):
...         a, b = [printer.doprint(i) for i in self.args]
...         return r"\operatorname{Mod}{\left(%s, %s\right)}" % (a, b)
```

This fails when the mode argument is passed to the printer:

```
>>> print_latex(ModOp(x, m), mode='inline')  # ok
$\operatorname{Mod}{\left(x, m\right)}$
>>> print_latex(ModOpModeWrong(x, m), mode='inline')  # bad
$\operatorname{Mod}{\left($x$, $m$\right)}$
```

2. Using `str(obj)` instead:

```
>>> class ModOpNestedWrong(Mod):
...     def _latex(self, printer):
...         a, b = [str(i) for i in self.args]
...         return r"\operatorname{Mod}{\left(%s, %s\right)}" % (a, b)
```

This fails on nested objects:

```
>>> # Nested modulo.
>>> print_latex(ModOp(ModOp(x, m), Integer(7)))  # ok
\operatorname{Mod}{\left(\operatorname{Mod}{\left(x, m\right)}, 7\right)}
>>> print_latex(ModOpNestedWrong(ModOpNestedWrong(x, m), Integer(7)))  #␣
↪bad
\operatorname{Mod}{\left(ModOpNestedWrong(x, m), 7\right)}
```

3. Using `LatexPrinter()._print(obj)` instead.

```
>>> from sympy.printing.latex import LatexPrinter
>>> class ModOpSettingsWrong(Mod):
...     def _latex(self, printer):
...         a, b = [LatexPrinter()._print(i) for i in self.args]
...         return r"\operatorname{Mod}{\left(%s, %s\right)}" % (a, b)
```

This causes all the settings to be discarded in the subobjects. As an example, the `full_prec` setting which shows floats to full precision is ignored:

```
>>> from sympy import Float
>>> print_latex(ModOp(Float(1) * x, m), full_prec=True)  # ok
\operatorname{Mod}{\left(1.00000000000000 x, m\right)}
>>> print_latex(ModOpSettingsWrong(Float(1) * x, m), full_prec=True)  #␣
↪bad
\operatorname{Mod}{\left(1.0 x, m\right)}
```

The main class responsible for printing is `Printer` (see also its source code):

class sympy.printing.printer.**Printer**(*settings=None*)

Generic printer

Its job is to provide infrastructure for implementing new printers easily.

If you want to define your custom Printer or your custom printing method for your custom class then see the example above: *printer_example* (page 2136) .

printmethod: str = None

_print(*expr*, ***kwargs*) → str

Internal dispatcher

Tries the following concepts to print an expression:

1. Let the object print itself if it knows how.
2. Take the best fitting method defined in the printer.
3. As fall-back use the emptyPrinter method for the printer.

doprint(*expr*)

Returns printer's representation for expr (as a string)

classmethod set_global_settings(***settings*)

Set system-wide printing settings.

PrettyPrinter Class

The pretty printing subsystem is implemented in `sympy.printing.pretty.pretty` by the `PrettyPrinter` class deriving from `Printer`. It relies on the modules `sympy.printing.pretty.stringPict`, and `sympy.printing.pretty.pretty_symbology` for rendering nice-looking formulas.

The module `stringPict` provides a base class `stringPict` and a derived class `prettyForm` that ease the creation and manipulation of formulas that span across multiple lines.

The module `pretty_symbology` provides primitives to construct 2D shapes (hline, vline, etc) together with a technique to use unicode automatically when possible.

class sympy.printing.pretty.pretty.**PrettyPrinter**(*settings=None*)

Printer, which converts an expression into 2D ASCII-art figure.

printmethod: str = '_pretty'

sympy.printing.pretty.pretty.**pretty**(*expr*, ***, *order=None*, *full_prec='auto'*, *use_unicode=True*, *wrap_line=False*, *num_columns=None*, *use_unicode_sqrt_char=True*, *root_notation=True*, *mat_symbol_style='plain'*, *imaginary_unit='i'*, *perm_cyclic=True*)

Returns a string containing the prettified form of expr.

For information on keyword arguments see pretty_print function.

sympy.printing.pretty.pretty.**pretty_print**(*expr*, ***kwargs*)

Prints expr in pretty form.

pprint is just a shortcut for this function.

Parameters

expr : expression

The expression to print.

wrap_line : bool, optional (default=True)

Line wrapping enabled/disabled.

num_columns : int or None, optional (default=None)

Number of columns before line breaking (default to None which reads the terminal width), useful when using SymPy without terminal.

use_unicode : bool or None, optional (default=None)

Use unicode characters, such as the Greek letter pi instead of the string pi.

full_prec : bool or string, optional (default="auto")

Use full precision.

order : bool or string, optional (default=None)

Set to 'none' for long expressions if slow; default is None.

use_unicode_sqrt_char : bool, optional (default=True)

Use compact single-character square root symbol (when unambiguous).

root_notation : bool, optional (default=True)

Set to 'False' for printing exponents of the form 1/n in fractional form. By default exponent is printed in root form.

mat_symbol_style : string, optional (default="plain")

Set to "bold" for printing MatrixSymbols using a bold mathematical symbol face. By default the standard face is used.

imaginary_unit : string, optional (default="i")

Letter to use for imaginary unit when use_unicode is True. Can be "i" (default) or "j".

C code printers

This class implements C code printing, i.e. it converts Python expressions to strings of C code (see also `C89CodePrinter`).

Usage:

```
>>> from sympy.printing import print_ccode
>>> from sympy.functions import sin, cos, Abs, gamma
>>> from sympy.abc import x
>>> print_ccode(sin(x)**2 + cos(x)**2, standard='C89')
pow(sin(x), 2) + pow(cos(x), 2)
>>> print_ccode(2*x + cos(x), assign_to="result", standard='C89')
result = 2*x + cos(x);
>>> print_ccode(Abs(x**2), standard='C89')
fabs(pow(x, 2))
>>> print_ccode(gamma(x**2), standard='C99')
tgamma(pow(x, 2))
```

sympy.printing.c.known_functions_C89 = {'Abs': [(<function <lambda>>, 'fabs'), (<function <lambda>>, 'abs')], 'acos': 'acos', 'asin': 'asin', 'atan': 'atan', 'atan2': 'atan2', 'ceiling': 'ceil', 'cos': 'cos', 'cosh': 'cosh', 'exp': 'exp', 'floor': 'floor', 'log': 'log', 'sin': 'sin', 'sinh': 'sinh', 'sqrt': 'sqrt', 'tan': 'tan', 'tanh': 'tanh'}

sympy.printing.c.**known_functions_C99 = {'Abs': [(<function <lambda>>, 'fabs'), (<function <lambda>>, 'abs')], 'Cbrt': 'cbrt', 'Max': 'fmax', 'Min': 'fmin', 'acos': 'acos', 'acosh': 'acosh', 'asin': 'asin', 'asinh': 'asinh', 'atan': 'atan', 'atan2': 'atan2', 'atanh': 'atanh', 'ceiling': 'ceil', 'cos': 'cos', 'cosh': 'cosh', 'erf': 'erf', 'erfc': 'erfc', 'exp': 'exp', 'exp2': 'exp2', 'expm1': 'expm1', 'floor': 'floor', 'fma': 'fma', 'gamma': 'tgamma', 'hypot': 'hypot', 'log': 'log', 'log10': 'log10', 'log1p': 'log1p', 'log2': 'log2', 'loggamma': 'lgamma', 'sin': 'sin', 'sinh': 'sinh', 'sqrt': 'sqrt', 'tan': 'tan', 'tanh': 'tanh'}**

class sympy.printing.c.**C89CodePrinter**(*settings=None*)

A printer to convert Python expressions to strings of C code

printmethod: str = '_ccode'

indent_code(*code*)

Accepts a string of code or a list of code lines

class sympy.printing.c.**C99CodePrinter**(*settings=None*)

printmethod: str = '_ccode'

sympy.printing.c.**ccode**(*expr*, *assign_to=None*, *standard='c99'*, ***settings*)

Converts an expr to a string of c code

Parameters

expr : Expr

A SymPy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

standard : str, optional

String specifying the standard. If your compiler supports a more modern standard you may set this to 'c99' to allow the printer to use more math functions. [default='c89'].

precision : integer, optional

The precision for numbers such as pi [default=17].

user_functions : dict, optional

A dictionary where the keys are string representations of either `FunctionClass` or `UndefinedFunction` instances and the values are their desired C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)] or [(argument_test, cfunction_formater)]. See below for examples.

dereference : iterable, optional

An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if `dereference=[a]`, the resulting code would print `(*a)` instead of `a`.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> from sympy import ccode, symbols, Rational, sin, ceiling, Abs,␣
↪Function
>>> x, tau = symbols("x, tau")
>>> expr = (2*tau)**Rational(7, 2)
>>> ccode(expr)
'8*M_SQRT2*pow(tau, 7.0/2.0)'
>>> ccode(expr, math_macros={})
'8*sqrt(2)*pow(tau, 7.0/2.0)'
>>> ccode(sin(x), assign_to="s")
's = sin(x);'
>>> from sympy.codegen.ast import real, float80
>>> ccode(expr, type_aliases={real: float80})
'8*M_SQRT2l*powl(tau, 7.0L/2.0L)'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type" : "function"} to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...   "ceiling": "CEIL",
...   "Abs": [(lambda x: not x.is_integer, "fabs"),
...           (lambda x: x.is_integer, "ABS")],
...   "func": "f"
... }
>>> func = Function('func')
>>> ccode(func(Abs(x) + ceiling(x)), standard='C89', user_
↪functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

or if the C-function takes a subset of the original arguments:

```
>>> ccode(2**x + 3**x, standard='C99', user_functions={'Pow': [
...   (lambda b, e: b == 2, lambda b, e: 'exp2(%s)' % e),
...   (lambda b, e: b != 2, 'pow')]})
'exp2(x) + pow(3, x)'
```

`Piecewise` expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if

the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(ccode(expr, tau, standard='C89'))
if (x > 0) {
tau = x + 1;
}
else {
tau = x;
}
```

Support for loops is provided through Indexed types. With contract=True these expressions will be turned into loops, whereas contract=False will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> ccode(e.rhs, assign_to=e.lhs, contract=False, standard='C89')
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a MatrixSymbol of the same dimensions must be provided to assign_to. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(ccode(mat, A, standard='C89'))
A[0] = pow(x, 2);
if (x > 0) {
   A[1] = x + 1;
}
else {
   A[1] = x;
}
A[2] = sin(x);
```

sympy.printing.c.**print_ccode**(*expr*, ***settings*)

Prints C representation of the given expression.

C++ code printers

This module contains printers for C++ code, i.e. functions to convert SymPy expressions to strings of C++ code.

Usage:

```
>>> from sympy.printing import cxxcode
>>> from sympy.functions import Min, gamma
>>> from sympy.abc import x
>>> print(cxxcode(Min(gamma(x) - 1, x), standard='C++11'))
std::min(x, std::tgamma(x) - 1)
```

class sympy.printing.cxx.**CXX98CodePrinter**(*settings=None*)

printmethod: str = '_cxxcode'

class sympy.printing.cxx.**CXX11CodePrinter**(*settings=None*)

printmethod: str = '_cxxcode'

sympy.printing.codeprinter.**cxxcode**(*expr*, *assign_to=None*, *standard='c++11'*, ***settings*)

C++ equivalent of *ccode()* (page 2141).

RCodePrinter

This class implements R code printing (i.e. it converts Python expressions to strings of R code).

Usage:

```
>>> from sympy.printing import print_rcode
>>> from sympy.functions import sin, cos, Abs
>>> from sympy.abc import x
>>> print_rcode(sin(x)**2 + cos(x)**2)
sin(x)^2 + cos(x)^2
>>> print_rcode(2*x + cos(x), assign_to="result")
result = 2*x + cos(x);
>>> print_rcode(Abs(x**2))
abs(x^2)
```

sympy.printing.rcode.**known_functions = {'Abs': 'abs', 'Max': 'max', 'Min': 'min', 'acos': 'acos', 'acosh': 'acosh', 'asin': 'asin', 'asinh': 'asinh', 'atan': 'atan', 'atan2': 'atan2', 'atanh': 'atanh', 'beta': 'beta', 'ceiling': 'ceiling', 'cos': 'cos', 'cosh': 'cosh', 'digamma': 'digamma', 'erf': 'erf', 'exp': 'exp', 'factorial': 'factorial', 'floor': 'floor', 'gamma': 'gamma', 'log': 'log', 'sign': 'sign', 'sin': 'sin', 'sinh': 'sinh', 'sqrt': 'sqrt', 'tan': 'tan', 'tanh': 'tanh', 'trigamma': 'trigamma'}**

class sympy.printing.rcode.**RCodePrinter**(*settings={}*)

A printer to convert SymPy expressions to strings of R code

printmethod: str = '_rcode'

indent_code(*code*)

Accepts a string of code or a list of code lines

sympy.printing.rcode.**rcode**(*expr*, *assign_to=None*, ***settings*)

Converts an expr to a string of r code

Parameters

expr : Expr

A SymPy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision : integer, optional

The precision for numbers such as pi [default=15].

user_functions : dict, optional

A dictionary where the keys are string representations of either `FunctionClass` or `UndefinedFunction` instances and the values are their desired R string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, rfunction_string)] or [(argument_test, rfunction_formater)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> from sympy import rcode, symbols, Rational, sin, ceiling, Abs,␣
↪Function
>>> x, tau = symbols("x, tau")
>>> rcode((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau^(7.0/2.0)'
>>> rcode(sin(x), assign_to="s")
's = sin(x);'
```

Simple custom printing can be defined for certain types by passing a dictionary of {"type" : "function"} to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...   "ceiling": "CEIL",
...   "Abs": [(lambda x: not x.is_integer, "fabs"),
...           (lambda x: x.is_integer, "ABS")],
...   "func": "f"
... }
>>> func = Function('func')
>>> rcode(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

or if the R-function takes a subset of the original arguments:

```
>>> rcode(2**x + 3**x, user_functions={'Pow': [
...   (lambda b, e: b == 2, lambda b, e: 'exp2(%s)' % e),
...   (lambda b, e: b != 2, 'pow')]})
'exp2(x) + pow(3, x)'
```

`Piecewise` expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the `Piecewise` lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(rcode(expr, assign_to=tau))
tau = ifelse(x > 0,x + 1,x);
```

Support for loops is provided through `Indexed` types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> rcode(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(rcode(mat, A))
A[0] = x^2;
A[1] = ifelse(x > 0,x + 1,x);
A[2] = sin(x);
```

sympy.printing.rcode.**print_rcode**(*expr*, ***settings*)

Prints R representation of the given expression.

Fortran Printing

The `fcode` function translates a sympy expression into Fortran code. The main purpose is to take away the burden of manually translating long mathematical expressions. Therefore the resulting expression should also require no (or very little) manual tweaking to make it compilable. The optional arguments of `fcode` can be used to fine-tune the behavior of `fcode` in such a way that manual changes in the result are no longer needed.

`sympy.printing.fortran.`**`fcode`**(*expr*, *assign_to=None*, ***settings*)

Converts an expr to a string of fortran code

Parameters

expr : Expr

A SymPy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision : integer, optional

DEPRECATED. Use type_mappings instead. The precision for numbers such as pi [default=17].

user_functions : dict, optional

A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

source_format : optional

The source format can be either ‘fixed’ or ‘free’. [default=’fixed’]

standard : integer, optional

The Fortran standard to be followed. This is specified as an integer. Acceptable standards are 66, 77, 90, 95, 2003, and 2008. Default

is 77. Note that currently the only distinction internally is between standards before 95, and those 95 and after. This may change later as more features are added.

name_mangling : bool, optional

If True, then the variables that would become identical in case-insensitive Fortran are mangled by appending different number of _ at the end. If False, SymPy Will not interfere with naming of variables. [default=True]

Examples

```
>>> from sympy import fcode, symbols, Rational, sin, ceiling, floor
>>> x, tau = symbols("x, tau")
>>> fcode((2*tau)**Rational(7, 2))
'      8*sqrt(2.0d0)*tau**(7.0d0/2.0d0)'
>>> fcode(sin(x), assign_to="s")
'      s = sin(x)'
```

Custom printing can be defined for certain types by passing a dictionary of "type" : "function" to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...   "ceiling": "CEIL",
...   "floor": [(lambda x: not x.is_integer, "FLOOR1"),
...             (lambda x: x.is_integer, "FLOOR2")]
... }
>>> fcode(floor(x) + ceiling(x), user_functions=custom_functions)
'      CEIL(x) + FLOOR1(x)'
```

`Piecewise` expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the `Piecewise` lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
      if (x > 0) then
         tau = x + 1
      else
         tau = x
      end if
```

Support for loops is provided through `Indexed` types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from sympy import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
```

(continues on next page)

(continued from previous page)

```
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> fcode(e.rhs, assign_to=e.lhs, contract=False)
'      Dy(i) = (y(i + 1) - y(i))/(t(i + 1) - t(i))'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(fcode(mat, A))
      A(1, 1) = x**2
         if (x > 0) then
      A(2, 1) = x + 1
         else
      A(2, 1) = x
         end if
      A(3, 1) = sin(x)
```

sympy.printing.fortran.**print_fcode**(*expr*, ***settings*)

Prints the Fortran representation of the given expression.

See fcode for the meaning of the optional arguments.

class sympy.printing.fortran.**FCodePrinter**(*settings=None*)

A printer to convert SymPy expressions to strings of Fortran code

printmethod: str = '_fcode'

indent_code(*code*)

Accepts a string of code or a list of code lines

Two basic examples:

```
>>> from sympy import *
>>> x = symbols("x")
>>> fcode(sqrt(1-x**2))
'      sqrt(1 - x**2)'
>>> fcode((3 + 4*I)/(1 - conjugate(x)))
'      (cmplx(3,4))/(1 - conjg(x))'
```

An example where line wrapping is required:

```
>>> expr = sqrt(1-x**2).series(x,n=20).removeO()
>>> print(fcode(expr))
    -715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
    @ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
    @ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
    @ /2.0d0*x**2 + 1
```

In case of line wrapping, it is handy to include the assignment so that lines are wrapped properly when the assignment part is added.

```
>>> print(fcode(expr, assign_to="var"))
      var = -715.0d0/65536.0d0*x**18 - 429.0d0/32768.0d0*x**16 - 33.0d0/
     @ 2048.0d0*x**14 - 21.0d0/1024.0d0*x**12 - 7.0d0/256.0d0*x**10 -
     @ 5.0d0/128.0d0*x**8 - 1.0d0/16.0d0*x**6 - 1.0d0/8.0d0*x**4 - 1.0d0
     @ /2.0d0*x**2 + 1
```

For piecewise functions, the `assign_to` option is mandatory:

```
>>> print(fcode(Piecewise((x,x<1),(x**2,True)), assign_to="var"))
      if (x < 1) then
        var = x
      else
        var = x**2
      end if
```

Note that by default only top-level piecewise functions are supported due to the lack of a conditional operator in Fortran 77. Inline conditionals can be supported using the `merge` function introduced in Fortran 95 by setting of the kwarg `standard=95`:

```
>>> print(fcode(Piecewise((x,x<1),(x**2,True)), standard=95))
      merge(x, x**2, x < 1)
```

Loops are generated if there are Indexed objects in the expression. This also requires use of the assign_to option.

```
>>> A, B = map(IndexedBase, ['A', 'B'])
>>> m = Symbol('m', integer=True)
>>> i = Idx('i', m)
>>> print(fcode(2*B[i], assign_to=A[i]))
    do i = 1, m
        A(i) = 2*B(i)
    end do
```

Repeated indices in an expression with Indexed objects are interpreted as summation. For instance, code for the trace of a matrix can be generated with

```
>>> print(fcode(A[i, i], assign_to=x))
      x = 0
      do i = 1, m
          x = x + A(i, i)
      end do
```

By default, number symbols such as `pi` and `E` are detected and defined as Fortran parameters. The precision of the constants can be tuned with the precision argument. Parameter definitions are easily avoided using the `N` function.

```
>>> print(fcode(x - pi**2 - E))
      parameter (E = 2.7182818284590452d0)
      parameter (pi = 3.1415926535897932d0)
      x - pi**2 - E
>>> print(fcode(x - pi**2 - E, precision=25))
```

(continues on next page)

(continued from previous page)

```
      parameter (E = 2.71828182845904523536028 7d0)
      parameter (pi = 3.14159265358979323846264 3d0)
      x - pi**2 - E
>>> print(fcode(N(x - pi**2, 25)))
      x - 9.8696044010893586188344910d0
```

When some functions are not part of the Fortran standard, it might be desirable to introduce the names of user-defined functions in the Fortran expression.

```
>>> print(fcode(1 - gamma(x)**2, user_functions={'gamma': 'mygamma'}))
      1 - mygamma(x)**2
```

However, when the user_functions argument is not provided, `fcode` will generate code which assumes that a function of the same name will be provided by the user. A comment will be added to inform the user of the issue:

```
>>> print(fcode(1 - gamma(x)**2))
C     Not supported in Fortran:
C     gamma
      1 - gamma(x)**2
```

The printer can be configured to omit these comments:

```
>>> print(fcode(1 - gamma(x)**2, allow_unknown_functions=True))
      1 - gamma(x)**2
```

By default the output is human readable code, ready for copy and paste. With the option `human=False`, the return value is suitable for post-processing with source code generators that write routines with multiple instructions. The return value is a three-tuple containing: (i) a set of number symbols that must be defined as 'Fortran parameters', (ii) a list functions that cannot be translated in pure Fortran and (iii) a string of Fortran code. A few examples:

```
>>> fcode(1 - gamma(x)**2, human=False)
(set(), {gamma(x)}, '      1 - gamma(x)**2')
>>> fcode(1 - sin(x)**2, human=False)
(set(), set(), '      1 - sin(x)**2')
>>> fcode(x - pi**2, human=False)
({(pi, '3.1415926535897932d0')}, set(), '      x - pi**2')
```

Mathematica code printing

sympy.printing.mathematica.known_functions = {'Chi': [(<function <lambda>>, 'CoshIntegral')], 'Ci': [(<function <lambda>>, 'CosIntegral')], 'DiracDelta': [(<function <lambda>>, 'DiracDelta')], 'Ei': [(<function <lambda>>, 'ExpIntegralEi')], 'FallingFactorial': [(<function <lambda>>, 'FactorialPower')], 'Heaviside': [(<function <lambda>>, 'HeavisideTheta')], 'KroneckerDelta': [(<function <lambda>>, 'KroneckerDelta')], 'Max': [(<function <lambda>>, 'Max')], 'Min': [(<function <lambda>>, 'Min')], 'RisingFactorial': [(<function <lambda>>, 'Pochhammer')], 'Shi': [(<function <lambda>>, 'SinhIntegral')], 'Si': [(<function <lambda>>, 'SinIntegral')], 'acos': [(<function <lambda>>, 'ArcCos')], 'acosh': [(<function <lambda>>, 'ArcCosh')], 'acot': [(<function <lambda>>, 'ArcCot')], 'acoth': [(<function <lambda>>, 'ArcCoth')], 'acsc': [(<function <lambda>>, 'ArcCsc')], 'acsch': [(<function <lambda>>, 'ArcCsch')], 'airyai': [(<function <lambda>>, 'AiryAi')], 'airyaiprime': [(<function <lambda>>, 'AiryAiPrime')], 'airybi': [(<function <lambda>>, 'AiryBi')], 'airybiprime': [(<function <lambda>>, 'AiryBiPrime')], 'appellf1': [(<function <lambda>>, 'AppellF1')], 'asec': [(<function <lambda>>, 'ArcSec')], 'asech': [(<function <lambda>>, 'ArcSech')], 'asin': [(<function <lambda>>, 'ArcSin')], 'asinh': [(<function <lambda>>, 'ArcSinh')], 'assoc_laguerre': [(<function <lambda>>, 'LaguerreL')], 'assoc_legendre': [(<function <lambda>>, 'LegendreP')], 'atan': [(<function <lambda>>, 'ArcTan')], 'atan2': [(<function <lambda>>, 'ArcTan')], 'atanh': [(<function <lambda>>, 'ArcTanh')], 'besseli': [(<function <lambda>>, 'BesselI')], 'besselj': [(<function <lambda>>, 'BesselJ')], 'besselk': [(<function <lambda>>, 'BesselK')], 'bessely': [(<function <lambda>>, 'BesselY')], 'beta': [(<function <lambda>>, 'Beta')], 'catalan': [(<function <lambda>>, 'CatalanNumber')], 'chebyshevt': [(<function <lambda>>, 'ChebyshevT')], 'chebyshevu': [(<function <lambda>>, 'ChebyshevU')], 'conjugate': [(<function <lambda>>, 'Conjugate')], 'cos': [(<function <lambda>>, 'Cos')], 'cosh': [(<function <lambda>>, 'Cosh')], 'cot': [(<function <lambda>>, 'Cot')], 'coth': [(<function <lambda>>, 'Coth')], 'csc': [(<function <lambda>>, 'Csc')], 'csch': [(<function <lambda>>, 'Csch')], 'dirichlet_eta': [(<function <lambda>>, 'DirichletEta')], 'elliptic_e': [(<function <lambda>>, 'EllipticE')], 'elliptic_f': [(<function <lambda>>, 'EllipticE')], 'elliptic_k': [(<function <lambda>>, 'EllipticK')], 'elliptic_pi': [(<function <lambda>>, 'EllipticPi')], 'erf': [(<function <lambda>>, 'Erf')], 'erf2': [(<function <lambda>>, 'Erf')], 'erf2inv': [(<function <lambda>>, 'InverseErf')], 'erfc': [(<function <lambda>>, 'Erfc')], 'erfcinv': [(<function <lambda>>, 'InverseErfc')], 'erfi': [(<function <lambda>>, 'Erfi')], 'erfinv': [(<function <lambda>>, 'InverseErf')], 'exp': [(<function <lambda>>, 'Exp')], 'expint': [(<function <lambda>>, 'ExpIntegralE')], 'factorial': [(<function <lambda>>, 'Factorial')], 'factorial2': [(<function <lambda>>, 'Factorial2')], 'fresnelc': [(<function <lambda>>, 'FresnelC')], 'fresnels': [(<function <lambda>>, 'FresnelS')], 'gamma': [(<function <lambda>>, 'Gamma')], 'gcd': [(<function <lambda>>, 'GCD')], 'gegenbauer': [(<function <lambda>>, 'GegenbauerC')], 'hankel1': [(<function <lambda>>, 'HankelH1')], 'hankel2': [(<function <lambda>>, 'HankelH2')], 'harmonic': [(<function <lambda>>, 'HarmonicNumber')], 'hermite': [(<function <lambda>>, 'HermiteH')], 'hyper': [(<function <lambda>>, 'HypergeometricPFQ')], 'jacobi': [(<function <lambda>>, 'JacobiP')], 'jn': [(<function <lambda>>, 'SphericalBesselJ')], 'laguerre': [(<function <lambda>>, 'LaguerreL')], 'lcm': [(<function <lambda>>, 'LCM')], 'legendre': [(<function <lambda>>, 'LegendreP')], 'lerchphi': [(<function <lambda>>, 'LerchPhi')], 'li': [(<function <lambda>>, 'LogIntegral')], 'log': [(<function <lambda>>, 'Log')], 'loggamma': [(<function <lambda>>, 'LogGamma')], 'lucas': [(<function <lambda>>, 'LucasL')], 'mathieuc': [(<function <lambda>>, 'MathieuC')], 'mathieucprime': [(<function <lambda>>, 'MathieuCPrime')], 'mathieus': [(<function <lambda>>, 'MathieuS')], 'mathieusprime': [(<function <lambda>>, 'MathieuSPrime')], 'meijerg': [(<function <lambda>>, 'MeijerG')], 'polygamma': [(<function <lambda>>, 'PolyGamma')], 'polylog': [(<function <lambda>>, 'PolyLog')], 'riemann_xi':

class sympy.printing.mathematica.**MCodePrinter**(*settings={}*)

A printer to convert Python expressions to strings of the Wolfram's Mathematica code

printmethod: str = '_mcode'

sympy.printing.mathematica.**mathematica_code**(*expr*, ***settings*)

Converts an expr to a string of the Wolfram Mathematica code

Examples

```
>>> from sympy import mathematica_code as mcode, symbols, sin
>>> x = symbols('x')
>>> mcode(sin(x).series(x).removeO())
'(1/120)*x^5 - 1/6*x^3 + x'
```

Maple code printing

class sympy.printing.maple.**MapleCodePrinter**(*settings=None*)

Printer which converts a SymPy expression into a maple code.

printmethod: str = '_maple'

sympy.printing.maple.**maple_code**(*expr*, *assign_to=None*, ***settings*)

Converts `expr` to a string of Maple code.

Parameters

expr : Expr

A SymPy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`, `MatrixSymbol`, or `Indexed` type. This can be helpful for expressions that generate multi-line statements.

precision : integer, optional

The precision for numbers such as pi [default=16].

user_functions : dict, optional

A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

inline: bool, optional

If True, we try to create single-statement code instead of multiple statements. [default=True].

sympy.printing.maple.**print_maple_code**(*expr*, ***settings*)

Prints the Maple representation of the given expression.

See *maple_code()* (page 2154) for the meaning of the optional arguments.

Examples

```
>>> from sympy import print_maple_code, symbols
>>> x, y = symbols('x y')
>>> print_maple_code(x, assign_to=y)
y := x
```

Javascript Code printing

sympy.printing.jscode.**known_functions = {'Abs': 'Math.abs', 'Max': 'Math.max', 'Min': 'Math.min', 'acos': 'Math.acos', 'acosh': 'Math.acosh', 'asin': 'Math.asin', 'asinh': 'Math.asinh', 'atan': 'Math.atan', 'atan2': 'Math.atan2', 'atanh': 'Math.atanh', 'ceiling': 'Math.ceil', 'cos': 'Math.cos', 'cosh': 'Math.cosh', 'exp': 'Math.exp', 'floor': 'Math.floor', 'log': 'Math.log', 'sign': 'Math.sign', 'sin': 'Math.sin', 'sinh': 'Math.sinh', 'tan': 'Math.tan', 'tanh': 'Math.tanh'}**

class sympy.printing.jscode.**JavascriptCodePrinter**(*settings={}*)

"A Printer to convert Python expressions to strings of JavaScript code

printmethod: str = '_javascript'

indent_code(*code*)

Accepts a string of code or a list of code lines

sympy.printing.jscode.**jscode**(*expr*, *assign_to=None*, ***settings*)

Converts an expr to a string of javascript code

Parameters

expr : Expr

A SymPy expression to be converted.

assign_to : optional

When given, the argument is used as the name of the variable to which the expression is assigned. Can be a string, `Symbol`,

`MatrixSymbol`, or `Indexed` type. This is helpful in case of line-wrapping, or for expressions that generate multi-line statements.

precision : integer, optional

The precision for numbers such as pi [default=15].

user_functions : dict, optional

A dictionary where keys are `FunctionClass` instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, js_function_string)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, `Indexed` instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> from sympy import jscode, symbols, Rational, sin, ceiling, Abs
>>> x, tau = symbols("x, tau")
>>> jscode((2*tau)**Rational(7, 2))
'8*Math.sqrt(2)*Math.pow(tau, 7/2)'
>>> jscode(sin(x), assign_to="s")
's = Math.sin(x);'
```

Custom printing can be defined for certain types by passing a dictionary of "type" : "function" to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, js_function_string)].

```
>>> custom_functions = {
...   "ceiling": "CEIL",
...   "Abs": [(lambda x: not x.is_integer, "fabs"),
...           (lambda x: x.is_integer, "ABS")]
... }
>>> jscode(Abs(x) + ceiling(x), user_functions=custom_functions)
'fabs(x) + CEIL(x)'
```

`Piecewise` expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the `Piecewise` lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.