```
>>> from sympy import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(jscode(expr, tau))
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]))
>>> jscode(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(jscode(mat, A))
A[0] = Math.pow(x, 2);
if (x > 0) {
   A[1] = x + 1;
}
else {
   A[1] = x;
}
A[2] = Math.sin(x);
```

Julia code printing

sympy.printing.julia.**known_fcns_src1 = ['sin', 'cos', 'tan', 'cot', 'sec', 'csc', 'asin', 'acos', 'atan', 'acot', 'asec', 'acsc', 'sinh', 'cosh', 'tanh', 'coth', 'sech', 'csch', 'asinh', 'acosh', 'atanh', 'acoth', 'asech', 'acsch', 'sinc', 'atan2', 'sign', 'floor', 'log', 'exp', 'cbrt', 'sqrt', 'erf', 'erfc', 'erfi', 'factorial', 'gamma', 'digamma', 'trigamma', 'polygamma', 'beta', 'airyai', 'airyaiprime', 'airybi', 'airybiprime', 'besselj', 'bessely', 'besseli', 'besselk', 'erfinv', 'erfcinv']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

sympy.printing.julia.**known_fcns_src2 = {'Abs': 'abs', 'ceiling': 'ceil', 'conjugate': 'conj', 'hankel1': 'hankelh1', 'hankel2': 'hankelh2', 'im': 'imag', 're': 'real'}**

class sympy.printing.julia.**JuliaCodePrinter**(*settings={}*)

A printer to convert expressions to strings of Julia code.

printmethod: str = '_julia'

indent_code(*code*)

Accepts a string of code or a list of code lines

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

Converts *expr* to a string of Julia 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].

Examples

```
>>> from sympy import julia_code, symbols, sin, pi
>>> x = symbols('x')
>>> julia_code(sin(x).series(x).removeO())
'x .^ 5 / 120 - x .^ 3 / 6 + x'
```

```
>>> from sympy import Rational, ceiling
>>> x, y, tau = symbols("x, y, tau")
>>> julia_code((2*tau)**Rational(7, 2))
'8 * sqrt(2) * tau .^ (7 // 2)'
```

Note that element-wise (Hadamard) operations are used by default between symbols. This is because its possible in Julia to write "vectorized" code. It is harmless if the values are scalars.

```
>>> julia_code(sin(pi*x*y), assign_to="s")
's = sin(pi * x .* y)'
```

If you need a matrix product "*" or matrix power "^", you can specify the symbol as a `MatrixSymbol`.

```
>>> from sympy import Symbol, MatrixSymbol
>>> n = Symbol('n', integer=True, positive=True)
>>> A = MatrixSymbol('A', n, n)
>>> julia_code(3*pi*A**3)
'(3 * pi) * A ^ 3'
```

This class uses several rules to decide which symbol to use a product. Pure numbers use "*", Symbols use ".*" and MatrixSymbols use "*". A HadamardProduct can be used to specify componentwise multiplication ".*" of two MatrixSymbols. There is currently there is no easy way to specify scalar symbols, so sometimes the code might have some minor cosmetic issues. For example, suppose x and y are scalars and A is a Matrix, then while a human programmer might write "(x^2*y)*A^3", we generate:

```
>>> julia_code(x**2*y*A**3)
'(x .^ 2 .* y) * A ^ 3'
```

Matrices are supported using Julia inline notation. When using `assign_to` with matrices, the name can be specified either as a string or as a `MatrixSymbol`. The dimensions must align in the latter case.

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([[x**2, sin(x), ceiling(x)]])
>>> julia_code(mat, assign_to='A')
'A = [x .^ 2 sin(x) ceil(x)]'
```

`Piecewise` expressions are implemented with logical masking by default. Alternatively, you can pass "inline=False" to use if-else conditionals. 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
>>> pw = Piecewise((x + 1, x > 0), (x, True))
```

(continues on next page)

(continued from previous page)

```
>>> julia_code(pw, assign_to=tau)
'tau = ((x > 0) ? (x + 1) : (x))'
```

Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> mat = Matrix([[x**2, pw, sin(x)]])
>>> julia_code(mat, assign_to='A')
'A = [x .^ 2 ((x > 0) ? (x + 1) : (x)) 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)]. This can be used to call a custom Julia function.

```
>>> from sympy import Function
>>> f = Function('f')
>>> g = Function('g')
>>> custom_functions = {
...   "f": "existing_julia_fcn",
...   "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...         (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> julia_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
'existing_julia_fcn(x) + my_fcn(x) + my_mat_fcn([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]))
>>> julia_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i]) ./ (t[i + 1] - t[i])'
```

Octave (and Matlab) Code printing

`sympy.printing.octave.`**`known_fcns_src1 = ['sin', 'cos', 'tan', 'cot', 'sec', 'csc', 'asin', 'acos', 'acot', 'atan', 'atan2', 'asec', 'acsc', 'sinh', 'cosh', 'tanh', 'coth', 'csch', 'sech', 'asinh', 'acosh', 'atanh', 'acoth', 'asech', 'acsch', 'erfc', 'erfi', 'erf', 'erfinv', 'erfcinv', 'besseli', 'besselj', 'besselk', 'bessely', 'bernoulli', 'beta', 'euler', 'exp', 'factorial', 'floor', 'fresnelc', 'fresnels', 'gamma', 'harmonic', 'log', 'polylog', 'sign', 'zeta', 'legendre']`**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

sympy.printing.octave.**known_fcns_src2 = {'Abs': 'abs', 'Chi': 'coshint', 'Ci': 'cosint', 'DiracDelta': 'dirac', 'Heaviside': 'heaviside', 'LambertW': 'lambertw', 'Max': 'max', 'Min': 'min', 'Mod': 'mod', 'RisingFactorial': 'pochhammer', 'Shi': 'sinhint', 'Si': 'sinint', 'arg': 'angle', 'binomial': 'bincoeff', 'ceiling': 'ceil', 'chebyshevt': 'chebyshevT', 'chebyshevu': 'chebyshevU', 'conjugate': 'conj', 'im': 'imag', 'laguerre': 'laguerreL', 'li': 'logint', 'loggamma': 'gammaln', 'polygamma': 'psi', 're': 'real'}**

class sympy.printing.octave.**OctaveCodePrinter**(*settings={}*)

A printer to convert expressions to strings of Octave/Matlab code.

printmethod: str = '_octave'

indent_code(*code*)

Accepts a string of code or a list of code lines

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

Converts *expr* to a string of Octave (or Matlab) code.

The string uses a subset of the Octave language for Matlab compatibility.

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].

Examples

```
>>> from sympy import octave_code, symbols, sin, pi
>>> x = symbols('x')
>>> octave_code(sin(x).series(x).removeO())
'x.^5/120 - x.^3/6 + x'
```

```
>>> from sympy import Rational, ceiling
>>> x, y, tau = symbols("x, y, tau")
>>> octave_code((2*tau)**Rational(7, 2))
'8*sqrt(2)*tau.^(7/2)'
```

Note that element-wise (Hadamard) operations are used by default between symbols. This is because its very common in Octave to write "vectorized" code. It is harmless if the values are scalars.

```
>>> octave_code(sin(pi*x*y), assign_to="s")
's = sin(pi*x.*y);'
```

If you need a matrix product "*" or matrix power "^", you can specify the symbol as a `MatrixSymbol`.

```
>>> from sympy import Symbol, MatrixSymbol
>>> n = Symbol('n', integer=True, positive=True)
>>> A = MatrixSymbol('A', n, n)
>>> octave_code(3*pi*A**3)
'(3*pi)*A^3'
```

This class uses several rules to decide which symbol to use a product. Pure numbers use "*", Symbols use ".*" and MatrixSymbols use "*". A HadamardProduct can be used to specify componentwise multiplication ".*" of two MatrixSymbols. There is currently there is no easy way to specify scalar symbols, so sometimes the code might have some minor cosmetic issues. For example, suppose x and y are scalars and A is a Matrix, then while a human programmer might write "(x^2*y)*A^3", we generate:

```
>>> octave_code(x**2*y*A**3)
'(x.^2.*y)*A^3'
```

Matrices are supported using Octave inline notation. When using `assign_to` with matrices, the name can be specified either as a string or as a `MatrixSymbol`. The dimensions must align in the latter case.

```
>>> from sympy import Matrix, MatrixSymbol
>>> mat = Matrix([[x**2, sin(x), ceiling(x)]])
>>> octave_code(mat, assign_to='A')
'A = [x.^2 sin(x) ceil(x)];'
```

`Piecewise` expressions are implemented with logical masking by default. Alternatively, you can pass "inline=False" to use if-else conditionals. 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
>>> pw = Piecewise((x + 1, x > 0), (x, True))
>>> octave_code(pw, assign_to=tau)
'tau = ((x > 0).*(x + 1) + (~(x > 0)).*(x));'
```

Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> mat = Matrix([[x**2, pw, sin(x)]])
>>> octave_code(mat, assign_to='A')
'A = [x.^2 ((x > 0).*(x + 1) + (~(x > 0)).*(x)) 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)]. This can be used to call a custom Octave function.

```
>>> from sympy import Function
>>> f = Function('f')
>>> g = Function('g')
>>> custom_functions = {
...   "f": "existing_octave_fcn",
...   "g": [(lambda x: x.is_Matrix, "my_mat_fcn"),
...         (lambda x: not x.is_Matrix, "my_fcn")]
... }
>>> mat = Matrix([[1, x]])
>>> octave_code(f(x) + g(x) + g(mat), user_functions=custom_functions)
'existing_octave_fcn(x) + my_fcn(x) + my_mat_fcn([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]))
>>> octave_code(e.rhs, assign_to=e.lhs, contract=False)
'Dy(i) = (y(i + 1) - y(i))./(t(i + 1) - t(i));'
```

Rust code printing

sympy.printing.rust.**known_functions = {'Abs': 'abs', 'Max': 'max', 'Min': 'min', 'Pow': [(<function <lambda>>, 'recip', 2), (<function <lambda>>, 'sqrt', 2), (<function <lambda>>, 'sqrt().recip', 2), (<function <lambda>>, 'cbrt', 2), (<function <lambda>>, 'exp2', 3), (<function <lambda>>, 'powi', 1), (<function <lambda>>, 'powf', 1)], 'acos': 'acos', 'acosh': 'acosh', 'asin': 'asin', 'asinh': 'asinh', 'atan': 'atan', 'atan2': 'atan2', 'atanh': 'atanh', 'ceiling': 'ceil', 'cos': 'cos', 'cosh': 'cosh', 'exp': [(<function <lambda>>, 'exp', 2)], 'floor': 'floor', 'log': 'ln', 'sign': 'signum', 'sin': 'sin', 'sinh': 'sinh', 'sqrt': 'sqrt', 'tan': 'tan', 'tanh': 'tanh'}**

class sympy.printing.rust.**RustCodePrinter**(*settings={}*)

A printer to convert SymPy expressions to strings of Rust code

printmethod: str = '_rust_code'

indent_code(*code*)

Accepts a string of code or a list of code lines

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

Converts an expr to a string of Rust 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 C string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. 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 rust_code, symbols, Rational, sin, ceiling, Abs,␣
↪Function
>>> x, tau = symbols("x, tau")
>>> rust_code((2*tau)**Rational(7, 2))
'8*1.4142135623731*tau.powf(7_f64/2.0)'
>>> rust_code(sin(x), assign_to="s")
's = x.sin();'
```

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", 4),
...           (lambda x: x.is_integer, "ABS", 4)],
...   "func": "f"
... }
>>> func = Function('func')
>>> rust_code(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'(fabs(x) + x.CEIL()).f()'
```

`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(rust_code(expr, tau))
tau = if (x > 0) {
    x + 1
} else {
    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,))
```

(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]))
>>> rust_code(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(rust_code(mat, A))
A = [x.powi(2), if (x > 0) {
    x + 1
} else {
    x
}, x.sin()];
```

Aesara Code printing

class sympy.printing.aesaracode.**AesaraPrinter**(**args*, ***kwargs*)

Code printer which creates Aesara symbolic expression graphs.

Parameters

cache : dict

Cache dictionary to use. If None (default) will use the global cache. To create a printer which does not depend on or alter global state pass an empty dictionary. Note: the dictionary is not copied on initialization of the printer and will be updated in-place, so using the same dict object when creating multiple printers or making multiple calls to *aesara_code()* (page 2167) or *aesara_function()* (page 2167) means the cache is shared between all these applications.

Attributes

cache	(dict) A cache of Aesara variables which have been created for SymPy symbol-like objects (e.g. *sympy.core.symbol.Symbol* (page 976) or *sympy.matrices.expressions.MatrixSymbol* (page 1372)). This is used to ensure that all references to a given symbol in an expression (or multiple expressions) are printed as the same Aesara variable, which is created only once. Symbols are differentiated only by name and type. The format of the cache's contents should be considered opaque to the user.

printmethod: str = '_aesara'

doprint(*expr*, *dtypes=None*, *broadcastables=None*)

Convert a SymPy expression to a Aesara graph variable.

The `dtypes` and `broadcastables` arguments are used to specify the data type, dimension, and broadcasting behavior of the Aesara variables corresponding to the free symbols in `expr`. Each is a mapping from SymPy symbols to the value of the corresponding argument to `aesara.tensor.var.TensorVariable`.

See the corresponding documentation page for more information on broadcasting in Aesara.

Parameters

expr : sympy.core.expr.Expr

SymPy expression to print.

dtypes : dict

Mapping from SymPy symbols to Aesara datatypes to use when creating new Aesara variables for those symbols. Corresponds to the `dtype` argument to `aesara.tensor.var.TensorVariable`. Defaults to `'floatX'` for symbols not included in the mapping.

broadcastables : dict

Mapping from SymPy symbols to the value of the `broadcastable` argument to `aesara.tensor.var.TensorVariable` to use when creating Aesara variables for those symbols. Defaults to the empty tuple for symbols not included in the mapping (resulting in a scalar).

Returns

aesara.graph.basic.Variable

A variable corresponding to the expression's value in a Aesara symbolic expression graph.

`sympy.printing.aesaracode.`**aesara_code**(*expr*, *cache=None*, ***kwargs*)

Convert a SymPy expression into a Aesara graph variable.

Parameters

expr : sympy.core.expr.Expr

SymPy expression object to convert.

cache : dict

Cached Aesara variables (see *AesaraPrinter.cache* (page 2166)). Defaults to the module-level global cache.

dtypes : dict

Passed to *AesaraPrinter.doprint()* (page 2166).

broadcastables : dict

Passed to *AesaraPrinter.doprint()* (page 2166).

Returns

aesara.graph.basic.Variable

A variable corresponding to the expression's value in a Aesara symbolic expression graph.

sympy.printing.aesaracode.**aesara_function**(*inputs*, *outputs*, *scalar=False*, ***, *dim=None*, *dims=None*, *broadcastables=None*, ***kwargs*)

Create a Aesara function from SymPy expressions.

The inputs and outputs are converted to Aesara variables using *aesara_code()* (page 2167) and then passed to `aesara.function`.

Parameters

inputs

Sequence of symbols which constitute the inputs of the function.

outputs

Sequence of expressions which constitute the outputs(s) of the function. The free symbols of each expression must be a subset of `inputs`.

scalar : bool

Convert 0-dimensional arrays in output to scalars. This will return a Python wrapper function around the Aesara function object.

cache : dict

Cached Aesara variables (see *AesaraPrinter.cache* (page 2166)). Defaults to the module-level global cache.

dtypes : dict

Passed to *AesaraPrinter.doprint()* (page 2166).

broadcastables : dict

Passed to *AesaraPrinter.doprint()* (page 2166).

dims : dict

Alternative to `broadcastables` argument. Mapping from elements of `inputs` to integers indicating the dimension of their associated arrays/tensors. Overrides `broadcastables` argument if given.

dim : int

Another alternative to the `broadcastables` argument. Common number of dimensions to use for all arrays/tensors. `aesara_function([x, y], [...], dim=2)` is equivalent to using `broadcastables={x: (False, False), y: (False, False)}`.

Returns

callable

A callable object which takes values of `inputs` as positional arguments and returns an output array for each of the expressions in `outputs`. If `outputs` is a single expression the function will return a Numpy array, if it is a list of multiple expressions the function will return a list of arrays. See description of the `squeeze` argument above for the behavior when a single output is passed in a list. The returned object will either be an instance of `aesara.compile.function.types.Function` or a Python wrapper function around one. In both cases, the returned value will have a `aesara_function` attribute which points to the return value of `aesara.function`.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.printing.aesaracode import aesara_function
```

A simple function with one input and one output:

```
>>> f1 = aesara_function([x], [x**2 - 1], scalar=True)
>>> f1(3)
8.0
```

A function with multiple inputs and one output:

```
>>> f2 = aesara_function([x, y, z], [(x**z + y**z)**(1/z)], scalar=True)
>>> f2(3, 4, 2)
5.0
```

A function with multiple inputs and multiple outputs:

```
>>> f3 = aesara_function([x, y], [x**2 + y**2, x**2 - y**2], scalar=True)
>>> f3(2, 3)
[13.0, -5.0]
```

See also:

dim_handling (page 2169)

sympy.printing.aesaracode.**dim_handling**(*inputs*, *dim=None*, *dims=None*, *broadcastables=None*)

Get value of `broadcastables` argument to *aesara_code()* (page 2167) from keyword arguments to *aesara_function()* (page 2167).

Included for backwards compatibility.

Parameters

inputs

Sequence of input symbols.

dim : int

Common number of dimensions for all inputs. Overrides other arguments if given.

dims : dict

Mapping from input symbols to number of dimensions. Overrides `broadcastables` argument if given.

broadcastables : dict

Explicit value of `broadcastables` argument to *AesaraPrinter.doprint()* (page 2166). If not None function will return this value unchanged.

Returns

dict

Dictionary mapping elements of `inputs` to their "broadcastable" values (tuple of `bools`).

Gtk

You can print to a gtkmathview widget using the function `print_gtk` located in `sympy.printing.gtk` (it requires to have installed gtkmathview and libgtkmathview-bin in some systems).

GtkMathView accepts MathML, so this rendering depends on the MathML representation of the expression.

Usage:

```
from sympy import *
print_gtk(x**2 + 2*exp(x**3))
```

sympy.printing.gtk.**print_gtk**(*x*, *start_viewer=True*)

Print to Gtkmathview, a gtk widget capable of rendering MathML.

Needs libgtkmathview-bin

LambdaPrinter

This classes implements printing to strings that can be used by the *sympy.utilities.lambdify.lambdify()* (page 2100) function.

class sympy.printing.lambdarepr.**LambdaPrinter**(*settings=None*)

This printer converts expressions into strings that can be used by lambdify.

printmethod: str = '_lambdacode'

sympy.printing.lambdarepr.**lambdarepr**(*expr*, ***settings*)

Returns a string usable for lambdifying.

LatexPrinter

This class implements LaTeX printing. See `sympy.printing.latex`.

sympy.printing.latex.**accepted_latex_functions = ['arcsin', 'arccos', 'arctan', 'sin', 'cos', 'tan', 'sinh', 'cosh', 'tanh', 'sqrt', 'ln', 'log', 'sec', 'csc', 'cot', 'coth', 're', 'im', 'frac', 'root', 'arg']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

class sympy.printing.latex.**LatexPrinter**(*settings=None*)

printmethod: str = '_latex'

parenthesize_super(*s*)

Protect superscripts in s

If the parenthesize_super option is set, protect with parentheses, else wrap in braces.

sympy.printing.latex.**latex**(*expr*, *, *full_prec=False*, *fold_frac_powers=False*, *fold_func_brackets=False*, *fold_short_frac=None*, *inv_trig_style='abbreviated'*, *itex=False*, *ln_notation=False*, *long_frac_ratio=None*, *mat_delim='['*, *mat_str=None*, *mode='plain'*, *mul_symbol=None*, *order=None*, *symbol_names={}*, *root_notation=True*, *mat_symbol_style='plain'*, *imaginary_unit='i'*, *gothic_re_im=False*, *decimal_separator='period'*, *perm_cyclic=True*, *parenthesize_super=True*, *min=None*, *max=None*, *diff_operator='d'*)

Convert the given expression to LaTeX string representation.

Parameters

full_prec: boolean, optional

If set to True, a floating point number is printed with full precision.

fold_frac_powers : boolean, optional

Emit `^{p/q}` instead of `^{\frac{p}{q}}` for fractional powers.

fold_func_brackets : boolean, optional

Fold function brackets where applicable.

fold_short_frac : boolean, optional

Emit `p / q` instead of `\frac{p}{q}` when the denominator is simple enough (at most two terms and no powers). The default value is `True` for inline mode, `False` otherwise.

inv_trig_style : string, optional

How inverse trig functions should be displayed. Can be one of `'abbreviated'`, `'full'`, or `'power'`. Defaults to `'abbreviated'`.

itex : boolean, optional

Specifies if itex-specific syntax is used, including emitting `$$...$$`.

ln_notation : boolean, optional

If set to `True`, `\ln` is used instead of default `\log`.

long_frac_ratio : float or None, optional

The allowed ratio of the width of the numerator to the width of the denominator before the printer breaks off long fractions. If `None` (the default value), long fractions are not broken up.

mat_delim : string, optional

The delimiter to wrap around matrices. Can be one of `'['`, `'('`, or the empty string `''`. Defaults to `'['`.

mat_str : string, optional

Which matrix environment string to emit. `'smallmatrix'`, `'matrix'`, `'array'`, etc. Defaults to `'smallmatrix'` for inline mode, `'matrix'` for matrices of no more than 10 columns, and `'array'` otherwise.

mode: string, optional

Specifies how the generated code will be delimited. `mode` can be one of `'plain'`, `'inline'`, `'equation'` or `'equation*'`. If `mode` is set to `'plain'`, then the resulting code will not be delimited at all (this is the default). If `mode` is set to `'inline'` then inline LaTeX `$...$` will be used. If `mode` is set to `'equation'` or `'equation*'`, the resulting code will be enclosed in the `equation` or `equation*` environment (remember to import `amsmath` for `equation*`), unless the `itex` option is set. In the latter case, the `$$...$$` syntax is used.

mul_symbol : string or None, optional

The symbol to use for multiplication. Can be one of `None`, `'ldot'`, `'dot'`, or `'times'`.

order: string, optional

Any of the supported monomial orderings (currently `'lex'`, `'grlex'`, or `'grevlex'`), `'old'`, and `'none'`. This parameter does nothing for $.Mul$ objects. Setting order to `'old'` uses the compatibility ordering for `~.Add` defined in Printer. For very large expressions, set the `order` keyword to `'none'` if speed is a concern.

symbol_names : dictionary of strings mapped to symbols, optional

Dictionary of symbols and the custom strings they should be emitted as.

root_notation : boolean, optional

If set to `False`, exponents of the form 1/n are printed in fractonal form. Default is `True`, to print exponent in root form.

mat_symbol_style : string, optional

Can be either `'plain'` (default) or `'bold'`. If set to `'bold'`, a $.MatrixSymbol$ A will be printed as `\mathbf{A}`, otherwise as `A`.

imaginary_unit : string, optional

String to use for the imaginary unit. Defined options are `'i'` (default) and `'j'`. Adding `r` or `t` in front gives `\mathrm` or `\text`, so `'ri'` leads to `\mathrm{i}` which gives i.

gothic_re_im : boolean, optional

If set to `True`, $\Re$ and $\Im$ is used for `re` and `im`, respectively. The default is `False` leading to re and im.

decimal_separator : string, optional

Specifies what separator to use to separate the whole and fractional parts of a floating point number as in 2.5 for the default, `period` or $2{,}5$ when `comma` is specified. Lists, sets, and tuple are printed with semicolon separating the elements when `comma` is chosen. For example, [1; 2; 3] when `comma` is chosen and [1,2,3] for when `period` is chosen.

parenthesize_super : boolean, optional

If set to `False`, superscripted expressions will not be parenthesized when powered. Default is `True`, which parenthesizes the expression when powered.

min: Integer or None, optional

Sets the lower bound for the exponent to print floating point numbers in fixed-point format.

max: Integer or None, optional

Sets the upper bound for the exponent to print floating point numbers in fixed-point format.

diff_operator: string, optional

String to use for differential operator. Default is `'d'`, to print in italic form. `'rd'`, `'td'` are shortcuts for `\mathrm{d}` and `\text{d}`.

Notes

Not using a print statement for printing, results in double backslashes for latex commands since that's the way Python escapes backslashes in strings.

```
>>> from sympy import latex, Rational
>>> from sympy.abc import tau
>>> latex((2*tau)**Rational(7,2))
'8 \\sqrt{2} \\tau^{\\frac{7}{2}}'
>>> print(latex((2*tau)**Rational(7,2)))
8 \sqrt{2} \tau^{\frac{7}{2}}
```

Examples

```
>>> from sympy import latex, pi, sin, asin, Integral, Matrix, Rational,␣
↪log
>>> from sympy.abc import x, y, mu, r, tau
```

Basic usage:

```
>>> print(latex((2*tau)**Rational(7,2)))
8 \sqrt{2} \tau^{\frac{7}{2}}
```

`mode` and `itex` options:

```
>>> print(latex((2*mu)**Rational(7,2), mode='plain'))
8 \sqrt{2} \mu^{\frac{7}{2}}
>>> print(latex((2*tau)**Rational(7,2), mode='inline'))
$8 \sqrt{2} \tau^{7 / 2}$
>>> print(latex((2*mu)**Rational(7,2), mode='equation*'))
\begin{equation*}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation*}
>>> print(latex((2*mu)**Rational(7,2), mode='equation'))
\begin{equation}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation}
>>> print(latex((2*mu)**Rational(7,2), mode='equation', itex=True))
$$8 \sqrt{2} \mu^{\frac{7}{2}}$$
>>> print(latex((2*mu)**Rational(7,2), mode='plain'))
8 \sqrt{2} \mu^{\frac{7}{2}}
>>> print(latex((2*tau)**Rational(7,2), mode='inline'))
$8 \sqrt{2} \tau^{7 / 2}$
>>> print(latex((2*mu)**Rational(7,2), mode='equation*'))
```

(continues on next page)

(continued from previous page)

```
\begin{equation*}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation*}
>>> print(latex((2*mu)**Rational(7,2), mode='equation'))
\begin{equation}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation}
>>> print(latex((2*mu)**Rational(7,2), mode='equation', itex=True))
$$8 \sqrt{2} \mu^{\frac{7}{2}}$$
```

Fraction options:

```
>>> print(latex((2*tau)**Rational(7,2), fold_frac_powers=True))
8 \sqrt{2} \tau^{7/2}
>>> print(latex((2*tau)**sin(Rational(7,2))))
\left(2 \tau\right)^{\sin{\left(\frac{7}{2} \right)}}
>>> print(latex((2*tau)**sin(Rational(7,2)), fold_func_brackets=True))
\left(2 \tau\right)^{\sin {\frac{7}{2}}}
>>> print(latex(3*x**2/y))
\frac{3 x^{2}}{y}
>>> print(latex(3*x**2/y, fold_short_frac=True))
3 x^{2} / y
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=2))
\frac{\int r\, dr}{2 \pi}
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=0))
\frac{1}{2 \pi} \int r\, dr
```

Multiplication options:

```
>>> print(latex((2*tau)**sin(Rational(7,2)), mul_symbol="times"))
\left(2 \times \tau\right)^{\sin{\left(\frac{7}{2} \right)}}
```

Trig options:

```
>>> print(latex(asin(Rational(7,2))))
\operatorname{asin}{\left(\frac{7}{2} \right)}
>>> print(latex(asin(Rational(7,2)), inv_trig_style="full"))
\arcsin{\left(\frac{7}{2} \right)}
>>> print(latex(asin(Rational(7,2)), inv_trig_style="power"))
\sin^{-1}{\left(\frac{7}{2} \right)}
```

Matrix options:

```
>>> print(latex(Matrix(2, 1, [x, y])))
\left[\begin{matrix}x\\y\end{matrix}\right]
>>> print(latex(Matrix(2, 1, [x, y]), mat_str = "array"))
\left[\begin{array}{c}x\\y\end{array}\right]
>>> print(latex(Matrix(2, 1, [x, y]), mat_delim="("))
\left(\begin{matrix}x\\y\end{matrix}\right)
```

Custom printing of symbols:

```
>>> print(latex(x**2, symbol_names={x: 'x_i'}))
x_i^{2}
```

Logarithms:

```
>>> print(latex(log(10)))
\log{\left(10 \right)}
>>> print(latex(log(10), ln_notation=True))
\ln{\left(10 \right)}
```

`latex()` also supports the builtin container types `list`, `tuple`, and `dict`:

```
>>> print(latex([2/x, y], mode='inline'))
$\left[ 2 / x, \  y\right]$
```

Unsupported types are rendered as monospaced plaintext:

```
>>> print(latex(int))
\mathtt{\text{<class 'int'>}}
>>> print(latex("plain % text"))
\mathtt{\text{plain \% text}}
```

See *Example of Custom Printing Method* (page 2137) for an example of how to override this behavior for your own types by implementing `_latex`.

Changed in version 1.7.0: Unsupported types no longer have their `str` representation treated as valid latex.

`sympy.printing.latex.`**`print_latex`**(*expr*, ***settings*)

Prints LaTeX representation of the given expression. Takes the same settings as `latex()`.

MathMLPrinter

This class is responsible for MathML printing. See `sympy.printing.mathml`.

More info on mathml : http://www.w3.org/TR/MathML2

class `sympy.printing.mathml.`**`MathMLPrinterBase`**(*settings=None*)

Contains common code required for MathMLContentPrinter and MathMLPresentationPrinter.

doprint(*expr*)

Prints the expression as MathML.

class `sympy.printing.mathml.`**`MathMLContentPrinter`**(*settings=None*)

Prints an expression to the Content MathML markup language.

References: https://www.w3.org/TR/MathML2/chapter4.html

printmethod: str = '_mathml_content'

mathml_tag(*e*)

Returns the MathML tag for an expression.

class `sympy.printing.mathml.`**`MathMLPresentationPrinter`**(*settings=None*)

Prints an expression to the Presentation MathML markup language.

References: https://www.w3.org/TR/MathML2/chapter3.html

printmethod: str = '_mathml_presentation'

mathml_tag(*e*)

Returns the MathML tag for an expression.

sympy.printing.mathml.**mathml**(*expr*, *printer='content'*, ***, *order=None*, *encoding='utf-8'*, *fold_frac_powers=False*, *fold_func_brackets=False*, *fold_short_frac=None*, *inv_trig_style='abbreviated'*, *ln_notation=False*, *long_frac_ratio=None*, *mat_delim='['*, *mat_symbol_style='plain'*, *mul_symbol=None*, *root_notation=True*, *symbol_names={}*, *mul_symbol_mathml_numbers='·'*)

Returns the MathML representation of expr. If printer is presentation then prints Presentation MathML else prints content MathML.

sympy.printing.mathml.**print_mathml**(*expr*, *printer='content'*, ***settings*)

Prints a pretty representation of the MathML code for expr. If printer is presentation then prints Presentation MathML else prints content MathML.

Examples

```
>>> ##
>>> from sympy import print_mathml
>>> from sympy.abc import x
>>> print_mathml(x+1)
<apply>
    <plus/>
    <ci>x</ci>
    <cn>1</cn>
</apply>
>>> print_mathml(x+1, printer='presentation')
<mrow>
    <mi>x</mi>
    <mo>+</mo>
    <mn>1</mn>
</mrow>
```

PythonCodePrinter

Python code printers

This module contains Python code printers for plain Python as well as NumPy & SciPy enabled code.

class sympy.printing.pycode.**MpmathPrinter**(*settings=None*)

Lambda printer for mpmath which maintains precision for floats

sympy.printing.pycode.**pycode**(*expr*, ***settings*)

Converts an expr to a string of Python code

Parameters

expr : Expr

A SymPy expression.

fully_qualified_modules : bool

Whether or not to write out full module names of functions (`math.sin` vs. `sin`). default: `True`.

standard : str or None, optional

Only 'python3' (default) is supported. This parameter may be removed in the future.

Examples

```
>>> from sympy import pycode, tan, Symbol
>>> pycode(tan(Symbol('x')) + 1)
'math.tan(x) + 1'
```

PythonPrinter

This class implements Python printing. Usage:

```
>>> from sympy import print_python, sin
>>> from sympy.abc import x

>>> print_python(5*x**3 + sin(x))
x = Symbol('x')
e = 5*x**3 + sin(x)
```

srepr

This printer generates executable code. This code satisfies the identity `eval(srepr(expr)) == expr`.

`srepr()` gives more low level textual output than `repr()`

Example:

```
>>> repr(5*x**3 + sin(x))
'5*x**3 + sin(x)'

>>> srepr(5*x**3 + sin(x))
"Add(Mul(Integer(5), Pow(Symbol('x'), Integer(3))), sin(Symbol('x')))"
```

`srepr()` gives the `repr` form, which is what `repr()` would normally give but for SymPy we don't actually use `srepr()` for `__repr__` because it's is so verbose, it is unlikely that anyone would want it called by default. Another reason is that lists call repr on their elements, like `print([a, b, c])` calls `repr(a)`, `repr(b)`, `repr(c)`. So if we used srepr for `` __repr__`` any list with SymPy objects would include the srepr form, even if we used `str()` or `print()`.

class sympy.printing.repr.**ReprPrinter**(*settings=None*)

printmethod: str = '_sympyrepr'

emptyPrinter(*expr*)

The fallback printer.

reprify(*args*, *sep*)

Prints each item in $args$ and joins them with sep.

sympy.printing.repr.**srepr**(*expr*, ***, *order=None*, *perm_cyclic=True*)

return expr in repr form

StrPrinter

This module generates readable representations of SymPy expressions.

class sympy.printing.str.**StrPrinter**(*settings=None*)

printmethod: str = '_sympystr'

sympy.printing.str.**sstr**(*expr*, ***, *order=None*, *full_prec='auto'*, *sympy_integers=False*, *abbrev=False*, *perm_cyclic=True*, *min=None*, *max=None*)

Returns the expression as a string.

For large expressions where speed is a concern, use the setting order='none'. If abbrev=True setting is used then units are printed in abbreviated form.

Examples

```
>>> from sympy import symbols, Eq, sstr
>>> a, b = symbols('a b')
>>> sstr(Eq(a + b, 0))
'Eq(a + b, 0)'
```

sympy.printing.str.**sstrrepr**(*expr*, ***, *order=None*, *full_prec='auto'*, *sympy_integers=False*, *abbrev=False*, *perm_cyclic=True*, *min=None*, *max=None*)

return expr in mixed str/repr form

i.e. strings are returned in repr form with quotes, and everything else is returned in str form.

This function could be useful for hooking into sys.displayhook

Tree Printing

The functions in this module create a representation of an expression as a tree.

sympy.printing.tree.**pprint_nodes**(*subtrees*)

Prettyprints systems of nodes.

Examples

```
>>> from sympy.printing.tree import pprint_nodes
>>> print(pprint_nodes(["a", "b1\nb2", "c"]))
+-a
+-b1
| b2
+-c
```

sympy.printing.tree.**print_node**(*node*, *assumptions=True*)

Returns information about the "node".

This includes class name, string representation and assumptions.

Parameters

assumptions : bool, optional

See the `assumptions` keyword in `tree`

sympy.printing.tree.**tree**(*node*, *assumptions=True*)

Returns a tree representation of "node" as a string.

It uses print_node() together with pprint_nodes() on node.args recursively.

Parameters

asssumptions : bool, optional

The flag to decide whether to print out all the assumption data (such as `is_integer`, ``is_real`) associated with the expression or not.

Enabling the flag makes the result verbose, and the printed result may not be deterministic because of the randomness used in back-tracing the assumptions.

See also:

print_tree (page 2179)

sympy.printing.tree.**print_tree**(*node*, *assumptions=True*)

Prints a tree representation of "node".

Parameters

asssumptions : bool, optional

The flag to decide whether to print out all the assumption data (such as `is_integer`, ``is_real`) associated with the expression or not.

Enabling the flag makes the result verbose, and the printed result may not be deterministic because of the randomness used in back-tracing the assumptions.

Examples

```
>>> from sympy.printing import print_tree
>>> from sympy import Symbol
>>> x = Symbol('x', odd=True)
>>> y = Symbol('y', even=True)
```

Printing with full assumptions information:

```
>>> print_tree(y**x)
Pow: y**x
+-Symbol: y
| algebraic: True
| commutative: True
| complex: True
| even: True
| extended_real: True
| finite: True
| hermitian: True
| imaginary: False
| infinite: False
| integer: True
| irrational: False
| noninteger: False
| odd: False
| rational: True
| real: True
| transcendental: False
+-Symbol: x
  algebraic: True
  commutative: True
  complex: True
  even: False
  extended_nonzero: True
  extended_real: True
  finite: True
  hermitian: True
  imaginary: False
  infinite: False
  integer: True
  irrational: False
  noninteger: False
  nonzero: True
  odd: True
  rational: True
  real: True
  transcendental: False
  zero: False
```

Hiding the assumptions:

```
>>> print_tree(y**x, assumptions=False)
Pow: y**x
```

(continues on next page)

(continued from previous page)

```
+-Symbol: y
+-Symbol: x
```

See also:

tree (page 2179)

Preview

A useful function is `preview`:

`sympy.printing.preview.`**`preview`**(*expr*, *output='png'*, *viewer=None*, *euler=True*, *packages=()*, *filename=None*, *outputbuffer=None*, *preamble=None*, *dvioptions=None*, *outputTexFile=None*, *extra_preamble=None*, *fontsize=None*, ***latex_settings*)

View expression or LaTeX markup in PNG, DVI, PostScript or PDF form.

If the expr argument is an expression, it will be exported to LaTeX and then compiled using the available TeX distribution. The first argument, 'expr', may also be a LaTeX string. The function will then run the appropriate viewer for the given output format or use the user defined one. By default png output is generated.

By default pretty Euler fonts are used for typesetting (they were used to typeset the well known "Concrete Mathematics" book). For that to work, you need the 'eulervm.sty' LaTeX style (in Debian/Ubuntu, install the texlive-fonts-extra package). If you prefer default AMS fonts or your system lacks 'eulervm' LaTeX package then unset the 'euler' keyword argument.

To use viewer auto-detection, lets say for 'png' output, issue

```
>>> from sympy import symbols, preview, Symbol
>>> x, y = symbols("x,y")
```

```
>>> preview(x + y, output='png')
```

This will choose 'pyglet' by default. To select a different one, do

```
>>> preview(x + y, output='png', viewer='gimp')
```

The 'png' format is considered special. For all other formats the rules are slightly different. As an example we will take 'dvi' output format. If you would run

```
>>> preview(x + y, output='dvi')
```

then 'view' will look for available 'dvi' viewers on your system (predefined in the function, so it will try evince, first, then kdvi and xdvi). If nothing is found, it will fall back to using a system file association (via `open` and `xdg-open`). To always use your system file association without searching for the above readers, use

```
>>> from sympy.printing.preview import system_default_viewer
>>> preview(x + y, output='dvi', viewer=system_default_viewer)
```

If this still does not find the viewer you want, it can be set explicitly.

```
>>> preview(x + y, output='dvi', viewer='superior-dvi-viewer')
```

This will skip auto-detection and will run user specified 'superior-dvi-viewer'. If `view` fails to find it on your system it will gracefully raise an exception.

You may also enter `'file'` for the viewer argument. Doing so will cause this function to return a file object in read-only mode, if `filename` is unset. However, if it was set, then 'preview' writes the genereted file to this filename instead.

There is also support for writing to a `io.BytesIO` like object, which needs to be passed to the `outputbuffer` argument.

```
>>> from io import BytesIO
>>> obj = BytesIO()
>>> preview(x + y, output='png', viewer='BytesIO',
...         outputbuffer=obj)
```

The LaTeX preamble can be customized by setting the 'preamble' keyword argument. This can be used, e.g., to set a different font size, use a custom documentclass or import certain set of LaTeX packages.

```
>>> preamble = "\\documentclass[10pt]{article}\n" \
...            "\\usepackage{amsmath,amsfonts}\\begin{document}"
>>> preview(x + y, output='png', preamble=preamble)
```

It is also possible to use the standard preamble and provide additional information to the preamble using the `extra_preamble` keyword argument.

```
>>> from sympy import sin
>>> extra_preamble = "\\renewcommand{\\sin}{\\cos}"
>>> preview(sin(x), output='png', extra_preamble=extra_preamble)
```

If the value of 'output' is different from 'dvi' then command line options can be set ('dvioptions' argument) for the execution of the 'dvi'+output conversion tool. These options have to be in the form of a list of strings (see `subprocess.Popen`).

Additional keyword args will be passed to the *latex()* (page 2170) call, e.g., the `symbol_names` flag.

```
>>> phidd = Symbol('phidd')
>>> preview(phidd, symbol_names={phidd: r'\ddot{\varphi}'})
```

For post-processing the generated TeX File can be written to a file by passing the desired filename to the 'outputTexFile' keyword argument. To write the TeX code to a file named `"sample.tex"` and run the default png viewer to display the resulting bitmap, do

```
>>> preview(x + y, outputTexFile="sample.tex")
```

Implementation - Helper Classes/Functions

sympy.printing.conventions.**split_super_sub**(*text*)

Split a symbol name into a name, superscripts and subscripts

The first part of the symbol name is considered to be its actual 'name', followed by super- and subscripts. Each superscript is preceded with a "^" character or by "__". Each subscript is preceded by a "_" character. The three return values are the actual name, a list with superscripts and a list with subscripts.

Examples

```
>>> from sympy.printing.conventions import split_super_sub
>>> split_super_sub('a_x^1')
('a', ['1'], ['x'])
>>> split_super_sub('var_sub1__sup_sub2')
('var', ['sup'], ['sub1', 'sub2'])
```

CodePrinter

This class is a base class for other classes that implement code-printing functionality, and additionally lists a number of functions that cannot be easily translated to C or Fortran.

class sympy.printing.codeprinter.**CodePrinter**(*settings=None*)

The base class for code-printing subclasses.

printmethod: str = '_sympystr'

doprint(*expr*, *assign_to=None*)

Print the expression as code.

Parameters

expr : Expression

The expression to be printed.

assign_to : Symbol, string, MatrixSymbol, list of strings or Symbols (optional)

If provided, the printed code will set the expression to a variable or multiple variables with the name or names given in `assign_to`.

exception sympy.printing.codeprinter.**AssignmentError**

Raised if an assignment variable for a loop is missing.

Precedence

sympy.printing.precedence.**PRECEDENCE = {'Add': 40, 'And': 30, 'Atom': 1000, 'BitwiseAnd': 38, 'BitwiseOr': 36, 'BitwiseXor': 37, 'Func': 70, 'Lambda': 1, 'Mul': 50, 'Not': 100, 'Or': 20, 'Pow': 60, 'Relational': 35, 'Xor': 10}**

Default precedence values for some basic types.

sympy.printing.precedence.**PRECEDENCE_VALUES = {'Add': 40, 'And': 30, 'Equality': 50, 'Equivalent': 10, 'Function': 70, 'HadamardPower': 60, 'HadamardProduct': 50, 'Implies': 10, 'KroneckerProduct': 50, 'MatAdd': 40, 'MatPow': 60, 'MatrixSolve': 50, 'Mod': 50, 'NegativeInfinity': 40, 'Not': 100, 'Or': 20, 'Pow': 60, 'Relational': 35, 'Sub': 40, 'TensAdd': 40, 'TensMul': 50, 'Unequality': 50, 'Xor': 10}**

A dictionary assigning precedence values to certain classes. These values are treated like they were inherited, so not every single class has to be named here.

sympy.printing.precedence.**PRECEDENCE_FUNCTIONS = {'Float': <function precedence_Float>, 'FracElement': <function precedence_FracElement>, 'Integer': <function precedence_Integer>, 'Mul': <function precedence_Mul>, 'PolyElement': <function precedence_PolyElement>, 'Rational': <function precedence_Rational>, 'UnevaluatedExpr': <function precedence_UnevaluatedExpr>}**

Sometimes it's not enough to assign a fixed precedence value to a class. Then a function can be inserted in this dictionary that takes an instance of this class as argument and returns the appropriate precedence value.

sympy.printing.precedence.**precedence**(*item*)

Returns the precedence of a given object.

This is the precedence for StrPrinter.

Pretty-Printing Implementation Helpers

sympy.printing.pretty.pretty_symbology.**U**(*name*)

Get a unicode character by name or, None if not found.

This exists because older versions of Python use older unicode databases.

sympy.printing.pretty.pretty_symbology.**pretty_use_unicode**(*flag=None*)

Set whether pretty-printer should use unicode by default

sympy.printing.pretty.pretty_symbology.**pretty_try_use_unicode**()

See if unicode output is available and leverage it if possible

sympy.printing.pretty.pretty_symbology.**xstr**(**args*)

The following two functions return the Unicode version of the inputted Greek letter.

sympy.printing.pretty.pretty_symbology.**g**(*l*)

sympy.printing.pretty.pretty_symbology.**G**(*l*)

sympy.printing.pretty.pretty_symbology.**greek_letters = ['alpha', 'beta', 'gamma', 'delta', 'epsilon', 'zeta', 'eta', 'theta', 'iota', 'kappa', 'lamda', 'mu', 'nu', 'xi', 'omicron', 'pi', 'rho', 'sigma', 'tau', 'upsilon', 'phi', 'chi', 'psi', 'omega']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

sympy.printing.pretty.pretty_symbology.**digit_2txt = {'0': 'ZERO', '1': 'ONE', '2': 'TWO', '3': 'THREE', '4': 'FOUR', '5': 'FIVE', '6': 'SIX', '7': 'SEVEN', '8': 'EIGHT', '9': 'NINE'}**

sympy.printing.pretty.pretty_symbology.**symb_2txt = {'(': 'LEFT PARENTHESIS', ')': 'RIGHT PARENTHESIS', '+': 'PLUS SIGN', '-': 'MINUS', '=': 'EQUALS SIGN', '[': 'LEFT SQUARE BRACKET', ']': 'RIGHT SQUARE BRACKET', 'int': 'INTEGRAL', 'sum': 'SUMMATION', '{': 'LEFT CURLY BRACKET', '{}': 'CURLY BRACKET', '}': 'RIGHT CURLY BRACKET'}**

The following functions return the Unicode subscript/superscript version of the character.

sympy.printing.pretty.pretty_symbology.**sub = {'(': '₍', ')': '₎', '+': '₊', '-': '₋', '0': '₀', '1': '₁', '2': '₂', '3': '₃', '4': '₄', '5': '₅', '6': '₆', '7': '₇', '8': '₈', '9': '₉', '=': '₌', 'a': 'ₐ', 'beta': 'ᵦ', 'chi': 'ᵪ', 'e': 'ₑ', 'gamma': 'ᵧ', 'h': 'ₕ', 'i': 'ᵢ', 'k': 'ₖ', 'l': 'ₗ', 'm': 'ₘ', 'n': 'ₙ', 'o': 'ₒ', 'p': 'ₚ', 'phi': 'ᵩ', 'r': 'ᵣ', 'rho': 'ᵨ', 's': 'ₛ', 't': 'ₜ', 'u': 'ᵤ', 'v': 'ᵥ', 'x': 'ₓ'}**

sympy.printing.pretty.pretty_symbology.**sup = {'(': '⁽', ')': '⁾', '+': '⁺', '-': '⁻', '0': '⁰', '1': '¹', '2': '²', '3': '³', '4': '⁴', '5': '⁵', '6': '⁶', '7': '⁷', '8': '⁸', '9': '⁹', '=': '⁼', 'i': 'ⁱ', 'n': 'ⁿ'}**

The following functions return Unicode vertical objects.

sympy.printing.pretty.pretty_symbology.**xobj**(*symb*, *length*)

Construct spatial object of given length.

return: [] of equal-length strings

sympy.printing.pretty.pretty_symbology.**vobj**(*symb*, *height*)

Construct vertical object of a given height

see: xobj

sympy.printing.pretty.pretty_symbology.**hobj**(*symb*, *width*)

Construct horizontal object of a given width

see: xobj

The following constants are for rendering roots and fractions.

sympy.printing.pretty.pretty_symbology.**root = {2: '√', 3: '∛', 4: '∜'}**

sympy.printing.pretty.pretty_symbology.**VF**(*txt*)

sympy.printing.pretty.pretty_symbology.**frac = {(1, 2): '½', (1, 3): '⅓', (1, 4): '¼', (1, 5): '⅕', (1, 6): '⅙', (1, 8): '⅛', (2, 3): '⅔', (2, 5): '⅖', (3, 4): '¾', (3, 5): '⅗', (3, 8): '⅜', (4, 5): '⅘', (5, 6): '⅚', (5, 8): '⅝', (7, 8): '⅞'}**

The following constants/functions are for rendering atoms and symbols.

sympy.printing.pretty.pretty_symbology.**xsym**(*sym*)

get symbology for a 'character'

sympy.printing.pretty.pretty_symbology.**atoms_table = {'Complexes': 'ℂ', 'EmptySequence': 'EmptySequence', 'EmptySet': 'ø', 'Exp1': 'e', 'ImaginaryUnit': 'i', 'Infinity': '∞', 'Integers': 'ℤ', 'Intersection': 'n', 'Modifier Letter Low Ring': '˳', 'Naturals': 'ℕ', 'Naturals0': 'ℕ₀', 'NegativeInfinity': '-∞', 'Pi': 'π', 'Rationals': 'ℚ', 'Reals': 'ℝ', 'Ring': '∘', 'SymmetricDifference': 'Δ', 'Union': 'u'}**

sympy.printing.pretty.pretty_symbology.**pretty_atom**(*atom_name*, *default=None*, *printer=None*)

return pretty representation of an atom

sympy.printing.pretty.pretty_symbology.**pretty_symbol**(*symb_name*, *bold_name=False*)

return pretty representation of a symbol

sympy.printing.pretty.pretty_symbology.**annotated**(*letter*)

Return a stylised drawing of the letter `letter`, together with information on how to put annotations (super- and subscripts to the left and to the right) on it.

See pretty.py functions _print_meijerg, _print_hyper on how to use this information.

Prettyprinter by Jurjen Bos. (I hate spammers: mail me at pietjepuk314 at the reverse of ku.oc.oohay). All objects have a method that create a "stringPict", that can be used in the str method for pretty printing.

Updates by Jason Gedge (email <my last name> at cs mun ca)

- terminal_string() method
- minor fixes and changes (mostly to prettyForm)

TODO:

- Allow left/center/right alignment options for above/below and top/center/bottom alignment options for left/right

class sympy.printing.pretty.stringpict.**stringPict**(*s*, *baseline=0*)

An ASCII picture. The pictures are represented as a list of equal length strings.

above(**args*)

Put pictures above this picture. Returns string, baseline arguments for stringPict. Baseline is baseline of bottom picture.

below(**args*)

Put pictures under this picture. Returns string, baseline arguments for stringPict. Baseline is baseline of top picture

Examples

```
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("x+3").below(
...       stringPict.LINE, '3')[0])
x+3
---
 3
```

height()

The height of the picture in characters.

left(**args*)

Put pictures (left to right) at left. Returns string, baseline arguments for stringPict.

leftslash()

Precede object by a slash of the proper size.

static next(**args*)

Put a string of stringPicts next to each other. Returns string, baseline arguments for stringPict.

parens(*left='(', right=')', ifascii_nougly=False*)

Put parentheses around self. Returns string, baseline arguments for stringPict.

left or right can be None or empty string which means 'no paren from that side'

render(**args*, ***kwargs*)

Return the string form of self.

Unless the argument line_break is set to False, it will break the expression in a form that can be printed on the terminal without being broken up.

right(**args*)

Put pictures next to this one. Returns string, baseline arguments for stringPict. (Multiline) strings are allowed, and are given a baseline of 0.

Examples

```
>>> from sympy.printing.pretty.stringpict import stringPict
>>> print(stringPict("10").right(" + ",stringPict("1\r-\r2",1))[0])
     1
10 + -
     2
```

root(*n=None*)

Produce a nice root symbol. Produces ugly results for big n inserts.

static stack(**args*)

Put pictures on top of each other, from top to bottom. Returns string, baseline arguments for stringPict. The baseline is the baseline of the second picture. Everything is centered. Baseline is the baseline of the second picture. Strings are allowed. The special value stringPict.LINE is a row of '-' extended to the width.

terminal_width()

Return the terminal width if possible, otherwise return 0.

width()

The width of the picture in characters.

class sympy.printing.pretty.stringpict.**prettyForm**(*s*, *baseline=0*, *binding=0*, *unicode=None*)

Extension of the stringPict class that knows about basic math applications, optimizing double minus signs.

"Binding" is interpreted as follows:

```
ATOM this is an atom: never needs to be parenthesized
FUNC this is a function application: parenthesize if added (?)
DIV  this is a division: make wider division if divided
POW  this is a power: only parenthesize if exponent
MUL  this is a multiplication: parenthesize if powered
ADD  this is an addition: parenthesize if multiplied or powered
NEG  this is a negative number: optimize if added, parenthesize if
     multiplied or powered
OPEN this is an open object: parenthesize if added, multiplied, or
     powered (example: Piecewise)
```

static apply(*function*, **args*)

Functions of one or more variables.

dotprint

sympy.printing.dot.**dotprint**(*expr*, *styles=((<class 'sympy.core.basic.Basic'>, {'color': 'blue', 'shape': 'ellipse'}), (<class 'sympy.core.expr.Expr'>, {'color': 'black'}))*, *atom=<function <lambda>>*, *maxdepth=None*, *repeat=True*, *labelfunc=<class 'str'>*, ***kwargs*)

DOT description of a SymPy expression tree

Parameters

styles : list of lists composed of (Class, mapping), optional

Styles for different classes.

The default is

```
(
    (Basic, {'color': 'blue', 'shape': 'ellipse'}),
    (Expr,  {'color': 'black'})
)
```

atom : function, optional

Function used to determine if an arg is an atom.

A good choice is `lambda x: not x.args`.

The default is `lambda x: not isinstance(x, Basic)`.

maxdepth : integer, optional

The maximum depth.

The default is `None`, meaning no limit.

repeat : boolean, optional

Whether to use different nodes for common subexpressions.

The default is `True`.

For example, for `x + x*y` with `repeat=True`, it will have two nodes for `x`; with `repeat=False`, it will have one node.

> **Warning:** Even if a node appears twice in the same object like `x` in `Pow(x, x)`, it will still only appear once. Hence, with `repeat=False`, the number of arrows out of an object might not equal the number of args it has.

labelfunc : function, optional

A function to create a label for a given leaf node.

The default is `str`.

Another good option is `srepr`.

For example with `str`, the leaf nodes of `x + 1` are labeled, `x` and `1`. With `srepr`, they are labeled `Symbol('x')` and `Integer(1)`.

****kwargs** : optional

Additional keyword arguments are included as styles for the graph.

Examples

```
>>> from sympy import dotprint
>>> from sympy.abc import x
>>> print(dotprint(x+2))
digraph{

# Graph style
"ordering"="out"
"rankdir"="TD"

#########
# Nodes #
#########

"Add(Integer(2), Symbol('x'))_()" ["color"="black", "label"="Add", "shape
↪"="ellipse"];
"Integer(2)_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol('x')_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];

#########
# Edges #
#########
```

(continues on next page)

(continued from previous page)

```
"Add(Integer(2), Symbol('x'))_()" -> "Integer(2)_(0,)";
"Add(Integer(2), Symbol('x'))_()" -> "Symbol('x')_(1,)";
}
```

5.8.8 Topics

Contents

Geometry

Introduction

The geometry module for SymPy allows one to create two-dimensional geometrical entities, such as lines and circles, and query for information about these entities. This could include asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines. The primary use case of the module involves entities with numerical values, but it is possible to also use symbolic representations.

Available Entities

The following entities are currently available in the geometry module:

- *Point* (page 2202)
- *Line* (page 2227), *Segment* (page 2233), *Ray* (page 2230)
- *Ellipse* (page 2253), *Circle* (page 2269)
- *Polygon* (page 2273), *RegularPolygon* (page 2284), *Triangle* (page 2293)

Most of the work one will do will be through the properties and methods of these entities, but several global methods exist:

- `intersection(entity1, entity2)`
- `are_similar(entity1, entity2)`
- `convex_hull(points)`

For a full API listing and an explanation of the methods and their return values please see the list of classes at the end of this document.

Example Usage

The following Python session gives one an idea of how to work with some of the geometry module.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>> x = Point(0, 0)
>>> y = Point(1, 1)
>>> z = Point(2, 2)
>>> zp = Point(1, 0)
>>> Point.is_collinear(x, y, z)
True
>>> Point.is_collinear(x, y, zp)
False
>>> t = Triangle(zp, y, x)
>>> t.area
1/2
>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(1, 1/2))
>>> m = t.medians
>>> intersection(m[x], m[y], m[zp])
[Point2D(2/3, 1/3)]
>>> c = Circle(x, 5)
>>> l = Line(Point(5, -5), Point(5, 5))
>>> c.is_tangent(l) # is l tangent to c?
True
>>> l = Line(x, y)
>>> c.is_tangent(l) # is l tangent to c?
False
>>> intersection(c, l)
[Point2D(-5*sqrt(2)/2, -5*sqrt(2)/2), Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
```

Intersection of medians

```
>>> from sympy import symbols
>>> from sympy.geometry import Point, Triangle, intersection

>>> a, b = symbols("a,b", positive=True)

>>> x = Point(0, 0)
>>> y = Point(a, 0)
>>> z = Point(2*a, b)
>>> t = Triangle(x, y, z)

>>> t.area
a*b/2

>>> t.medians[x]
Segment2D(Point2D(0, 0), Point2D(3*a/2, b/2))
```

(continues on next page)

(continued from previous page)

```
>>> intersection(t.medians[x], t.medians[y], t.medians[z])
[Point2D(a, b/3)]
```

An in-depth example: Pappus' Hexagon Theorem

From Wikipedia ([WikiPappus]):

> Given one set of collinear points A, B, C, and another set of collinear points a, b, c, then the intersection points X, Y, Z of line pairs Ab and aB, Ac and aC, Bc and bC are collinear.

```
>>> from sympy import *
>>> from sympy.geometry import *
>>>
>>> l1 = Line(Point(0, 0), Point(5, 6))
>>> l2 = Line(Point(0, 0), Point(2, -2))
>>>
>>> def subs_point(l, val):
...     """Take an arbitrary point and make it a fixed point."""
...     t = Symbol('t', real=True)
...     ap = l.arbitrary_point()
...     return Point(ap.x.subs(t, val), ap.y.subs(t, val))
...
>>> p11 = subs_point(l1, 5)
>>> p12 = subs_point(l1, 6)
>>> p13 = subs_point(l1, 11)
>>>
>>> p21 = subs_point(l2, -1)
>>> p22 = subs_point(l2, 2)
>>> p23 = subs_point(l2, 13)
>>>
>>> ll1 = Line(p11, p22)
>>> ll2 = Line(p11, p23)
>>> ll3 = Line(p12, p21)
>>> ll4 = Line(p12, p23)
>>> ll5 = Line(p13, p21)
>>> ll6 = Line(p13, p22)
>>>
>>> pp1 = intersection(ll1, ll3)[0]
>>> pp2 = intersection(ll2, ll5)[0]
>>> pp3 = intersection(ll4, ll6)[0]
>>>
>>> Point.is_collinear(pp1, pp2, pp3)
True
```

References

Miscellaneous Notes

- The area property of `Polygon` and `Triangle` may return a positive or negative value, depending on whether or not the points are oriented counter-clockwise or clockwise, respectively. If you always want a positive value be sure to use the `abs` function.
- Although `Polygon` can refer to any type of polygon, the code has been written for simple polygons. Hence, expect potential problems if dealing with complex polygons (overlapping sides).
- Since SymPy is still in its infancy some things may not simplify properly and hence some things that should return `True` (e.g., `Point.is_collinear`) may not actually do so. Similarly, attempting to find the intersection of entities that do intersect may result in an empty result.

Future Work

Truth Setting Expressions

When one deals with symbolic entities, it often happens that an assertion cannot be guaranteed. For example, consider the following code:

```
>>> from sympy import *
>>> from sympy.geometry import *
>>> x,y,z = map(Symbol, 'xyz')
>>> p1,p2,p3 = Point(x, y), Point(y, z), Point(2*x*y, y)
>>> Point.is_collinear(p1, p2, p3)
False
```

Even though the result is currently `False`, this is not *always* true. If the quantity $z - y - 2 * y * z + 2 * y * *2 == 0$ then the points will be collinear. It would be really nice to inform the user of this because such a quantity may be useful to a user for further calculation and, at the very least, being nice to know. This could be potentially done by returning an object (e.g., GeometryResult) that the user could use. This actually would not involve an extensive amount of work.

Three Dimensions and Beyond

Currently a limited subset of the geometry module has been extended to three dimensions, but it certainly would be a good addition to extend more. This would probably involve a fair amount of work since many of the algorithms used are specific to two dimensions.

Geometry Visualization

The plotting module is capable of plotting geometric entities. See *Plotting Geometric Entities* (page 2872) in the plotting module entry.

Submodules

Entities

class sympy.geometry.entity.**GeometryEntity**(**args*, ***kwargs*)

The base class for all geometrical entities.

This class does not represent any particular geometric entity, it only provides the implementation of some methods common to all subclasses.

property ambient_dimension

What is the dimension of the space that the object is contained in?

property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

encloses(*o*)

Return True if o is inside (not on or outside) the boundaries of self.

The object will be decomposed into Points and individual Entities need only define an encloses_point method for their class.

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t  = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t2 = Polygon(*RegularPolygon(Point(0, 0), 2, 3).vertices)
>>> t2.encloses(t)
True
>>> t.encloses(t2)
False
```

See also:

sympy.geometry.ellipse.Ellipse.encloses_point (page 2257), *sympy.geometry.polygon.Polygon.encloses_point* (page 2278)

intersection(*o*)

Returns a list of all of the intersections of self with o.

Notes

An entity is not required to implement this method.

If two different types of entities can intersect, the item with higher index in ordering_of_classes should implement intersections with anything having a lower index.

See also:

sympy.geometry.util.intersection (page 2197)

is_similar(*other*)

Is this geometrical entity similar to another geometrical entity?

Two entities are similar if a uniform scaling (enlarging or shrinking) of one of the entities will allow one to obtain the other.

Notes

This method is not intended to be used directly but rather through the $are_similar$ function found in util.py. An entity is not required to implement this method. If two different types of entities can be similar, it is only required that one of them be able to determine this.

See also:

scale (page 2196)

parameter_value(*other*, *t*)

Return the parameter corresponding to the given point. Evaluating an arbitrary point of the entity at this parameter value will return the given point.

Examples

```
>>> from sympy import Line, Point
>>> from sympy.abc import t
>>> a = Point(0, 0)
>>> b = Point(2, 2)
>>> Line(a, b).parameter_value((1, 1), t)
{t: 1/2}
>>> Line(a, b).arbitrary_point(t).subs(_)
Point2D(1, 1)
```

reflect(*line*)

Reflects an object across a line.

Parameters
line: Line

Examples

```
>>> from sympy import pi, sqrt, Line, RegularPolygon
>>> l = Line((0, pi), slope=sqrt(2))
>>> pent = RegularPolygon((1, 2), 1, 5)
>>> rpent = pent.reflect(l)
>>> rpent
RegularPolygon(Point2D(-2*sqrt(2)*pi/3 - 1/3 + 4*sqrt(2)/3, 2/3 +␣
↪2*sqrt(2)/3 + 2*pi/3), -1, 5, -atan(2*sqrt(2)) + 3*pi/5)
```

```
>>> from sympy import pi, Line, Circle, Point
>>> l = Line((0, pi), slope=1)
>>> circ = Circle(Point(0, 0), 5)
>>> rcirc = circ.reflect(l)
>>> rcirc
Circle(Point2D(-pi, pi), -5)
```

rotate(*angle*, *pt=None*)

Rotate `angle` radians counterclockwise about Point `pt`.

The default pt is the origin, Point(0, 0)

Examples

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t # vertex on x axis
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -
↪sqrt(3)/2))
>>> t.rotate(pi/2) # vertex on y axis now
Triangle(Point2D(0, 1), Point2D(-sqrt(3)/2, -1/2), Point2D(sqrt(3)/2,␣
↪-1/2))
```

See also:

scale (page 2196), *translate* (page 2197)

scale(*x=1*, *y=1*, *pt=None*)

Scale the object by multiplying the x,y-coordinates by x and y.

If pt is given, the scaling is done relative to that point; the object is shifted by -pt, scaled, and shifted by pt.