(continued from previous page)

```
>>> P(X(t) < 1).simplify()
lowergamma(2*t, 1)/gamma(2*t)
>>> P(Not((X(t) < 5) & (X(d) > 3)), Contains(t, Interval.Ropen(2, 4)) &
... Contains(d, Interval.Lopen(7, 8))).simplify()
-4*exp(-3) + 472*exp(-8)/3 + 1
>>> E(X(2) + x*E(X(5)))
10*x + 4
```

References

[R943]

Matrix Distributions

sympy.stats.**MatrixGamma**(*symbol, alpha, beta, scale_matrix*)

Creates a random variable with Matrix Gamma Distribution.

The density of the said distribution can be found at [1].

Parameters

alpha: Positive Real number

Shape Parameter

beta: Positive Real number

Scale Parameter

scale_matrix: Positive definite real square matrix

Scale Matrix

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density, MatrixGamma
>>> from sympy import MatrixSymbol, symbols
>>> a, b = symbols('a b', positive=True)
>>> M = MatrixGamma('M', a, b, [[2, 1], [1, 2]])
>>> X = MatrixSymbol('X', 2, 2)
>>> density(M)(X).doit()
exp(Trace(Matrix([
[-2/3,  1/3],
[ 1/3, -2/3]])*X)/b)*Determinant(X)**(a - 3/2)/
↪(3**a*sqrt(pi)*b**(2*a)*gamma(a)*gamma(a - 1/2))
>>> density(M)([[1, 0], [0, 1]]).doit()
exp(-4/(3*b))/(3**a*sqrt(pi)*b**(2*a)*gamma(a)*gamma(a - 1/2))
```

References

[R944]

sympy.stats.**Wishart**(*symbol, n, scale_matrix*)

Creates a random variable with Wishart Distribution.

The density of the said distribution can be found at [1].

Parameters

n: Positive Real number

Represents degrees of freedom

scale_matrix: Positive definite real square matrix

Scale Matrix

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import density, Wishart
>>> from sympy import MatrixSymbol, symbols
>>> n = symbols('n', positive=True)
>>> W = Wishart('W', n, [[2, 1], [1, 2]])
>>> X = MatrixSymbol('X', 2, 2)
>>> density(W)(X).doit()
exp(Trace(Matrix([
[-1/3,  1/6],
[ 1/6, -1/3]])*X))*Determinant(X)**(n/2 - 3/2)/(2**n*3**(n/
↪2)*sqrt(pi)*gamma(n/2)*gamma(n/2 - 1/2))
>>> density(W)([[1, 0], [0, 1]]).doit()
exp(-2/3)/(2**n*3**(n/2)*sqrt(pi)*gamma(n/2)*gamma(n/2 - 1/2))
```

References

[R945]

sympy.stats.**MatrixNormal**(*symbol, location_matrix, scale_matrix_1, scale_matrix_2*)

Creates a random variable with Matrix Normal Distribution.

The density of the said distribution can be found at [1].

Parameters

location_matrix: Real ``n x p`` matrix

Represents degrees of freedom

scale_matrix_1: Positive definite matrix

Scale Matrix of shape `n x n`

scale_matrix_2: Positive definite matrix

Scale Matrix of shape `p x p`

Returns
RandomSymbol

Examples

```
>>> from sympy import MatrixSymbol
>>> from sympy.stats import density, MatrixNormal
>>> M = MatrixNormal('M', [[1, 2]], [1], [[1, 0], [0, 1]])
>>> X = MatrixSymbol('X', 1, 2)
>>> density(M)(X).doit()
2*exp(-Trace((Matrix([
[-1],
[-2]]) + X.T)*(Matrix([[-1, -2]]) + X))/2)/pi
>>> density(M)([[3, 4]]).doit()
2*exp(-4)/pi
```

References

[R946]

Compound Distribution

class sympy.stats.compound_rv.**CompoundDistribution**(*dist*)
Class for Compound Distributions.

Parameters
dist : Distribution

Distribution must contain a random parameter

Examples

```
>>> from sympy.stats.compound_rv import CompoundDistribution
>>> from sympy.stats.crv_types import NormalDistribution
>>> from sympy.stats import Normal
>>> from sympy.abc import x
>>> X = Normal('X', 2, 4)
>>> N = NormalDistribution(X, 4)
>>> C = CompoundDistribution(N)
>>> C.set
Interval(-oo, oo)
>>> C.pdf(x, evaluate=True).simplify()
exp(-x**2/64 + x/16 - 1/16)/(8*sqrt(pi))
```

References

[R947]

Interface

sympy.stats.**P**(*condition*, *given_condition=None*, *numsamples=None*, *evaluate=True*, ***kwargs*)

Probability that a condition is true, optionally given a second condition.

Parameters

condition : Combination of Relationals containing RandomSymbols

The condition of which you want to compute the probability

given_condition : Combination of Relationals containing RandomSymbols

A conditional expression. P(X > 1, X > 0) is expectation of X > 1 given X > 0

numsamples : int

Enables sampling and approximates the probability with this many samples

evaluate : Bool (defaults to True)

In case of continuous systems return unevaluated integral

Examples

```
>>> from sympy.stats import P, Die
>>> from sympy import Eq
>>> X, Y = Die('X', 6), Die('Y', 6)
>>> P(X > 3)
1/2
>>> P(Eq(X, 5), X > 2) # Probability that X == 5 given that X > 2
1/4
>>> P(X > Y)
5/12
```

class sympy.stats.**Probability**(*prob*, *condition=None*, ***kwargs*)

Symbolic expression for the probability.

Examples

```
>>> from sympy.stats import Probability, Normal
>>> from sympy import Integral
>>> X = Normal(”X”, 0, 1)
>>> prob = Probability(X > 1)
>>> prob
Probability(X > 1)
```

Integral representation:

```
>>> prob.rewrite(Integral)
Integral(sqrt(2)*exp(-_z**2/2)/(2*sqrt(pi)), (_z, 1, oo))
```

Evaluation of the integral:

```
>>> prob.evaluate_integral()
sqrt(2)*(-sqrt(2)*sqrt(pi)*erf(sqrt(2)/2) + sqrt(2)*sqrt(pi))/
↪(4*sqrt(pi))
```

sympy.stats.**E**(*expr*, *condition=None*, *numsamples=None*, *evaluate=True*, ***kwargs*)

Returns the expected value of a random expression.

Parameters

expr : Expr containing RandomSymbols

The expression of which you want to compute the expectation value

given : Expr containing RandomSymbols

A conditional expression. E(X, X>0) is expectation of X given X > 0

numsamples : int

Enables sampling and approximates the expectation with this many samples

evalf : Bool (defaults to True)

If sampling return a number rather than a complex expression

evaluate : Bool (defaults to True)

In case of continuous systems return unevaluated integral

Examples

```
>>> from sympy.stats import E, Die
>>> X = Die('X', 6)
>>> E(X)
7/2
>>> E(2*X + 1)
8
```

```
>>> E(X, X > 3) # Expectation of X given that it is above 3
5
```

class sympy.stats.**Expectation**(*expr*, *condition=None*, ***kwargs*)

Symbolic expression for the expectation.

Examples

```
>>> from sympy.stats import Expectation, Normal, Probability, Poisson
>>> from sympy import symbols, Integral, Sum
>>> mu = symbols("mu")
>>> sigma = symbols("sigma", positive=True)
>>> X = Normal("X", mu, sigma)
>>> Expectation(X)
Expectation(X)
>>> Expectation(X).evaluate_integral().simplify()
mu
```

To get the integral expression of the expectation:

```
>>> Expectation(X).rewrite(Integral)
Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/(2*sqrt(pi)*sigma), (X,
↪ -oo, oo))
```

The same integral expression, in more abstract terms:

```
>>> Expectation(X).rewrite(Probability)
Integral(x*Probability(Eq(X, x)), (x, -oo, oo))
```

To get the Summation expression of the expectation for discrete random variables:

```
>>> lamda = symbols('lamda', positive=True)
>>> Z = Poisson('Z', lamda)
>>> Expectation(Z).rewrite(Sum)
Sum(Z*lamda**Z*exp(-lamda)/factorial(Z), (Z, 0, oo))
```

This class is aware of some properties of the expectation:

```
>>> from sympy.abc import a
>>> Expectation(a*X)
Expectation(a*X)
>>> Y = Normal("Y", 1, 2)
>>> Expectation(X + Y)
Expectation(X + Y)
```

To expand the `Expectation` into its expression, use `expand()`:

```
>>> Expectation(X + Y).expand()
Expectation(X) + Expectation(Y)
>>> Expectation(a*X + Y).expand()
a*Expectation(X) + Expectation(Y)
>>> Expectation(a*X + Y)
Expectation(a*X + Y)
>>> Expectation((X + Y)*(X - Y)).expand()
Expectation(X**2) - Expectation(Y**2)
```

To evaluate the `Expectation`, use `doit()`:

```
>>> Expectation(X + Y).doit()
mu + 1
```

(continues on next page)

(continued from previous page)

```
>>> Expectation(X + Expectation(Y + Expectation(2*X))).doit()
3*mu + 1
```

To prevent evaluating nested Expectation, use doit(deep=False)

```
>>> Expectation(X + Expectation(Y)).doit(deep=False)
mu + Expectation(Expectation(Y))
>>> Expectation(X + Expectation(Y + Expectation(2*X))).doit(deep=False)
mu + Expectation(Expectation(Y + Expectation(2*X)))
```

sympy.stats.**density**(*expr*, *condition=None*, *evaluate=True*, *numsamples=None*, ***kwargs*)

Probability density of a random expression, optionally given a second condition.

Parameters

expr : Expr containing RandomSymbols

The expression of which you want to compute the density value

condition : Relational containing RandomSymbols

A conditional expression. density(X > 1, X > 0) is density of X > 1 given X > 0

numsamples : int

Enables sampling and approximates the density with this many samples

Explanation

This density will take on different forms for different types of probability spaces. Discrete variables produce Dicts. Continuous variables produce Lambdas.

Examples

```
>>> from sympy.stats import density, Die, Normal
>>> from sympy import Symbol
```

```
>>> x = Symbol('x')
>>> D = Die('D', 6)
>>> X = Normal(x, 0, 1)
```

```
>>> density(D).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
>>> density(2*D).dict
{2: 1/6, 4: 1/6, 6: 1/6, 8: 1/6, 10: 1/6, 12: 1/6}
>>> density(X)(x)
sqrt(2)*exp(-x**2/2)/(2*sqrt(pi))
```

sympy.stats.**entropy**(*expr*, *condition=None*, ***kwargs*)

Calculuates entropy of a probability distribution.

Parameters

expression : the random expression whose entropy is to be calculated

condition : optional, to specify conditions on random expression

b: base of the logarithm, optional

By default, it is taken as Euler's number

Returns

result : Entropy of the expression, a constant

Examples

```
>>> from sympy.stats import Normal, Die, entropy
>>> X = Normal('X', 0, 1)
>>> entropy(X)
log(2)/2 + 1/2 + log(pi)/2
```

```
>>> D = Die('D', 4)
>>> entropy(D)
log(4)
```

References

[R948], [R949], [R950]

sympy.stats.**given**(*expr*, *condition=None*, ***kwargs*)

Conditional Random Expression.

Explanation

From a random expression and a condition on that expression creates a new probability space from the condition and returns the same expression on that conditional probability space.

Examples

```
>>> from sympy.stats import given, density, Die
>>> X = Die('X', 6)
>>> Y = given(X, X > 3)
>>> density(Y).dict
{4: 1/3, 5: 1/3, 6: 1/3}
```

Following convention, if the condition is a random symbol then that symbol is considered fixed.

```
>>> from sympy.stats import Normal
>>> from sympy import pprint
>>> from sympy.abc import z
>>> X = Normal('X', 0, 1)
>>> Y = Normal('Y', 0, 1)
>>> pprint(density(X + Y, Y)(z), use_unicode=False)
                2
       -(-Y + z)
       -----------
              2
  ___
\/ 2 *e
------------------
         ____
     2*\/ pi
```

sympy.stats.**where**(*condition*, *given_condition=None*, ***kwargs*)

Returns the domain where a condition is True.

Examples

```
>>> from sympy.stats import where, Die, Normal
>>> from sympy import And
```

```
>>> D1, D2 = Die('a', 6), Die('b', 6)
>>> a, b = D1.symbol, D2.symbol
>>> X = Normal('x', 0, 1)
```

```
>>> where(X**2<1)
Domain: (-1 < x) & (x < 1)
```

```
>>> where(X**2<1).set
Interval.open(-1, 1)
```

```
>>> where(And(D1<=D2, D2<3))
Domain: (Eq(a, 1) & Eq(b, 1)) | (Eq(a, 1) & Eq(b, 2)) | (Eq(a, 2) & Eq(b,
↪ 2))
```

sympy.stats.**variance**(*X*, *condition=None*, ***kwargs*)

Variance of a random expression.

$$variance(X) = E((X - E(X))^2)$$

Examples

```
>>> from sympy.stats import Die, Bernoulli, variance
>>> from sympy import simplify, Symbol
```

```
>>> X = Die('X', 6)
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)
>>> variance(2*X)
35/3
```

```
>>> simplify(variance(B))
p*(1 - p)
```

class sympy.stats.**Variance**(*arg*, *condition=None*, ***kwargs*)

Symbolic expression for the variance.

Examples

```
>>> from sympy import symbols, Integral
>>> from sympy.stats import Normal, Expectation, Variance, Probability
>>> mu = symbols("mu", positive=True)
>>> sigma = symbols("sigma", positive=True)
>>> X = Normal("X", mu, sigma)
>>> Variance(X)
Variance(X)
>>> Variance(X).evaluate_integral()
sigma**2
```

Integral representation of the underlying calculations:

```
>>> Variance(X).rewrite(Integral)
Integral(sqrt(2)*(X - Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/
↪(2*sqrt(pi)*sigma), (X, -oo, oo)))**2*exp(-(X - mu)**2/(2*sigma**2))/
↪(2*sqrt(pi)*sigma), (X, -oo, oo))
```

Integral representation, without expanding the PDF:

```
>>> Variance(X).rewrite(Probability)
-Integral(x*Probability(Eq(X, x)), (x, -oo, oo))**2 +␣
↪Integral(x**2*Probability(Eq(X, x)), (x, -oo, oo))
```

Rewrite the variance in terms of the expectation

```
>>> Variance(X).rewrite(Expectation)
-Expectation(X)**2 + Expectation(X**2)
```

Some transformations based on the properties of the variance may happen:

```
>>> from sympy.abc import a
>>> Y = Normal("Y", 0, 1)
>>> Variance(a*X)
Variance(a*X)
```

To expand the variance in its expression, use expand():

```
>>> Variance(a*X).expand()
a**2*Variance(X)
>>> Variance(X + Y)
Variance(X + Y)
```

(continues on next page)

(continued from previous page)

```
>>> Variance(X + Y).expand()
2*Covariance(X, Y) + Variance(X) + Variance(Y)
```

sympy.stats.**covariance**(*X*, *Y*, *condition=None*, ***kwargs*)

Covariance of two random expressions.

Explanation

The expectation that the two variables will rise and fall together

$$covariance(X, Y) = E((X - E(X))(Y - E(Y)))$$

Examples

```
>>> from sympy.stats import Exponential, covariance
>>> from sympy import Symbol
```

```
>>> rate = Symbol('lambda', positive=True, real=True)
>>> X = Exponential('X', rate)
>>> Y = Exponential('Y', rate)
```

```
>>> covariance(X, X)
lambda**(-2)
>>> covariance(X, Y)
0
>>> covariance(X, Y + rate*X)
1/lambda
```

class sympy.stats.**Covariance**(*arg1*, *arg2*, *condition=None*, ***kwargs*)

Symbolic expression for the covariance.

Examples

```
>>> from sympy.stats import Covariance
>>> from sympy.stats import Normal
>>> X = Normal("X", 3, 2)
>>> Y = Normal("Y", 0, 1)
>>> Z = Normal("Z", 0, 1)
>>> W = Normal("W", 0, 1)
>>> cexpr = Covariance(X, Y)
>>> cexpr
Covariance(X, Y)
```

Evaluate the covariance, X and Y are independent, therefore zero is the result:

```
>>> cexpr.evaluate_integral()
0
```

Rewrite the covariance expression in terms of expectations:

```
>>> from sympy.stats import Expectation
>>> cexpr.rewrite(Expectation)
Expectation(X*Y) - Expectation(X)*Expectation(Y)
```

In order to expand the argument, use `expand()`:

```
>>> from sympy.abc import a, b, c, d
>>> Covariance(a*X + b*Y, c*Z + d*W)
Covariance(a*X + b*Y, c*Z + d*W)
>>> Covariance(a*X + b*Y, c*Z + d*W).expand()
a*c*Covariance(X, Z) + a*d*Covariance(W, X) + b*c*Covariance(Y, Z) +␣
↪b*d*Covariance(W, Y)
```

This class is aware of some properties of the covariance:

```
>>> Covariance(X, X).expand()
Variance(X)
>>> Covariance(a*X, b*Y).expand()
a*b*Covariance(X, Y)
```

`sympy.stats.`**`coskewness`**(*X, Y, Z, condition=None, **kwargs*)

Calculates the co-skewness of three random variables.

Parameters

X : RandomSymbol

Random Variable used to calculate coskewness

Y : RandomSymbol

Random Variable used to calculate coskewness

Z : RandomSymbol

Random Variable used to calculate coskewness

condition : Expr containing RandomSymbols

A conditional expression

Returns

coskewness : The coskewness of the three random variables

Explanation

Mathematically Coskewness is defined as

$$coskewness(X,Y,Z)=\frac{E[(X-E[X])*(Y-E[Y])*(Z-E[Z])]}{\sigma_{X}\sigma_{Y}\sigma_{Z}}$$

Examples

```
>>> from sympy.stats import coskewness, Exponential, skewness
>>> from sympy import symbols
>>> p = symbols('p', positive=True)
>>> X = Exponential('X', p)
>>> Y = Exponential('Y', 2*p)
>>> coskewness(X, Y, Y)
0
>>> coskewness(X, Y + X, Y + 2*X)
16*sqrt(85)/85
>>> coskewness(X + 2*Y, Y + X, Y + 2*X, X > 3)
9*sqrt(170)/85
>>> coskewness(Y, Y, Y) == skewness(Y)
True
>>> coskewness(X, Y + p*X, Y + 2*p*X)
4/(sqrt(1 + 1/(4*p**2))*sqrt(4 + 1/(4*p**2)))
```

References

[R951]

sympy.stats.**median**(*X*, *evaluate=True*, ***kwargs*)

Calculuates the median of the probability distribution.

Parameters

X: The random expression whose median is to be calculated.

Returns

The FiniteSet or an Interval which contains the median of the random expression.

Explanation

Mathematically, median of Probability distribution is defined as all those values of m for which the following condition is satisfied

$$P(X \leq m) \geq \frac{1}{2} \text{ and } P(X \geq m) \geq \frac{1}{2}$$

Examples

```
>>> from sympy.stats import Normal, Die, median
>>> N = Normal('N', 3, 1)
>>> median(N)
{3}
>>> D = Die('D')
>>> median(D)
{3, 4}
```

References

[R952]

`sympy.stats.std(X, condition=None, **kwargs)`

Standard Deviation of a random expression

$$std(X) = \sqrt(E((X - E(X))^2))$$

Examples

```
>>> from sympy.stats import Bernoulli, std
>>> from sympy import Symbol, simplify
```

```
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)
```

```
>>> simplify(std(B))
sqrt(p*(1 - p))
```

`sympy.stats.quantile(expr, evaluate=True, **kwargs)`

Return the p^{th} order quantile of a probability distribution.

Explanation

Quantile is defined as the value at which the probability of the random variable is less than or equal to the given probability.

$$Q(p) = \inf\{x \in (-\infty, \infty) : p \le F(x)\}$$

Examples

```
>>> from sympy.stats import quantile, Die, Exponential
>>> from sympy import Symbol, pprint
>>> p = Symbol("p")
```

```
>>> l = Symbol("lambda", positive=True)
>>> X = Exponential("x", l)
>>> quantile(X)(p)
-log(1 - p)/lambda
```

```
>>> D = Die("d", 6)
>>> pprint(quantile(D)(p), use_unicode=False)
/nan  for Or(p > 1, p < 0)
|
| 1       for p <= 1/6
|
| 2       for p <= 1/3
```

(continues on next page)

(continued from previous page)

```
|
< 3       for p <= 1/2
|
| 4       for p <= 2/3
|
| 5       for p <= 5/6
|
\ 6        for p <= 1
```

sympy.stats.**sample**(*expr*, *condition=None*, *size=()*, *library='scipy'*, *numsamples=1*, *seed=None*, ***kwargs*)

A realization of the random expression.

Parameters

expr : Expression of random variables

Expression from which sample is extracted

condition : Expr containing RandomSymbols

A conditional expression

size : int, tuple

Represents size of each sample in numsamples

library : str

- 'scipy' : Sample using scipy
- 'numpy' : Sample using numpy
- 'pymc' : Sample using PyMC

Choose any of the available options to sample from as string, by default is 'scipy'

numsamples : int

Number of samples, each with size as `size`.

Deprecated since version 1.9.

The `numsamples` parameter is deprecated and is only provided for compatibility with v1.8. Use a list comprehension or an additional dimension in `size` instead. See *sympy.stats.sample(numsamples=n)* (page 168) for details.

seed :

An object to be used as seed by the given external library for sampling $expr$. Following is the list of possible types of object for the supported libraries,

- 'scipy': int, numpy.random.RandomState, numpy.random.Generator
- 'numpy': int, numpy.random.RandomState, numpy.random.Generator
- 'pymc': int

Optional, by default None, in which case seed settings related to the given library will be used. No modifications to environment's global seed settings are done by this argument.

Returns

sample: float/list/numpy.ndarray

one sample or a collection of samples of the random expression.

- sample(X) returns float/numpy.float64/numpy.int64 object.
- sample(X, size=int/tuple) returns numpy.ndarray object.

Examples

```
>>> from sympy.stats import Die, sample, Normal, Geometric
>>> X, Y, Z = Die('X', 6), Die('Y', 6), Die('Z', 6) # Finite Random␣
↪Variable
>>> die_roll = sample(X + Y + Z)
>>> die_roll
3
>>> N = Normal('N', 3, 4) # Continuous Random Variable
>>> samp = sample(N)
>>> samp in N.pspace.domain.set
True
>>> samp = sample(N, N>0)
>>> samp > 0
True
>>> samp_list = sample(N, size=4)
>>> [sam in N.pspace.domain.set for sam in samp_list]
[True, True, True, True]
>>> sample(N, size = (2,3))
array([[5.42519758, 6.40207856, 4.94991743],
   [1.85819627, 6.83403519, 1.9412172 ]])
>>> G = Geometric('G', 0.5) # Discrete Random Variable
>>> samp_list = sample(G, size=3)
>>> samp_list
[1, 3, 2]
>>> [sam in G.pspace.domain.set for sam in samp_list]
[True, True, True]
>>> MN = Normal("MN", [3, 4], [[2, 1], [1, 2]]) # Joint Random Variable
>>> samp_list = sample(MN, size=4)
>>> samp_list
[array([2.85768055, 3.38954165]),
 array([4.11163337, 4.3176591 ]),
 array([0.79115232, 1.63232916]),
 array([4.01747268, 3.96716083])]
>>> [tuple(sam) in MN.pspace.domain.set for sam in samp_list]
[True, True, True, True]
```

Changed in version 1.7.0: sample used to return an iterator containing the samples instead of value.

Changed in version 1.9.0: sample returns values or array of values instead of an iterator and numsamples is deprecated.

sympy.stats.**sample_iter**(*expr*, *condition=None*, *size=()*, *library='scipy'*, *numsamples=oo*, *seed=None*, ***kwargs*)

Returns an iterator of realizations from the expression given a condition.

Parameters

expr: Expr

Random expression to be realized

condition: Expr, optional

A conditional expression

size : int, tuple

Represents size of each sample in numsamples

numsamples: integer, optional

Length of the iterator (defaults to infinity)

seed :

An object to be used as seed by the given external library for sampling $expr$. Following is the list of possible types of object for the supported libraries,

- 'scipy': int, numpy.random.RandomState, numpy.random.Generator
- 'numpy': int, numpy.random.RandomState, numpy.random.Generator
- 'pymc': int

Optional, by default None, in which case seed settings related to the given library will be used. No modifications to environment's global seed settings are done by this argument.

Returns

sample_iter: iterator object

iterator object containing the sample/samples of given expr

Examples

```
>>> from sympy.stats import Normal, sample_iter
>>> X = Normal('X', 0, 1)
>>> expr = X*X + 3
>>> iterator = sample_iter(expr, numsamples=3)
>>> list(iterator)
[12, 4, 7]
```

See also:

sample (page 2971), *sampling_P* (page 2976), *sampling_E* (page 2976)

sympy.stats.**factorial_moment**(*X*, *n*, *condition=None*, ***kwargs*)

The factorial moment is a mathematical quantity defined as the expectation or average of the falling factorial of a random variable.

$$factorial - moment(X, n) = E(X(X-1)(X-2)...(X-n+1))$$

Parameters

n: A natural number, n-th factorial moment.

condition : Expr containing RandomSymbols

A conditional expression.

Examples

```
>>> from sympy.stats import factorial_moment, Poisson, Binomial
>>> from sympy import Symbol, S
>>> lamda = Symbol('lamda')
>>> X = Poisson('X', lamda)
>>> factorial_moment(X, 2)
lamda**2
>>> Y = Binomial('Y', 2, S.Half)
>>> factorial_moment(Y, 2)
1/2
>>> factorial_moment(Y, 2, Y > 1) # find factorial moment for Y > 1
2
```

References

[R953], [R954]

sympy.stats.**kurtosis**(*X*, *condition=None*, ***kwargs*)

Characterizes the tails/outliers of a probability distribution.

Parameters

condition : Expr containing RandomSymbols

A conditional expression. kurtosis(X, X>0) is kurtosis of X given X > 0

Explanation

Kurtosis of any univariate normal distribution is 3. Kurtosis less than 3 means that the distribution produces fewer and less extreme outliers than the normal distribution.

$$kurtosis(X) = E(((X - E(X))/\sigma_X)^4)$$

Examples

```
>>> from sympy.stats import kurtosis, Exponential, Normal
>>> from sympy import Symbol
>>> X = Normal('X', 0, 1)
>>> kurtosis(X)
3
>>> kurtosis(X, X > 0) # find kurtosis given X > 0
(-4/pi - 12/pi**2 + 3)/(1 - 2/pi)**2
```

```
>>> rate = Symbol('lamda', positive=True, real=True)
>>> Y = Exponential('Y', rate)
>>> kurtosis(Y)
9
```

References

[R955], [R956]

`sympy.stats.`**`skewness`**(*X*, *condition=None*, ***kwargs*)

Measure of the asymmetry of the probability distribution.

Parameters

condition : Expr containing RandomSymbols

A conditional expression. skewness(X, X>0) is skewness of X given X > 0

Explanation

Positive skew indicates that most of the values lie to the right of the mean.

$$skewness(X) = E(((X - E(X))/\sigma_X)^3)$$

Examples

```
>>> from sympy.stats import skewness, Exponential, Normal
>>> from sympy import Symbol
>>> X = Normal('X', 0, 1)
>>> skewness(X)
0
>>> skewness(X, X > 0) # find skewness given X > 0
(-sqrt(2)/sqrt(pi) + 4*sqrt(2)/pi**(3/2))/(1 - 2/pi)**(3/2)
```

```
>>> rate = Symbol('lambda', positive=True, real=True)
>>> Y = Exponential('Y', rate)
>>> skewness(Y)
2
```

`sympy.stats.`**`correlation`**(*X*, *Y*, *condition=None*, ***kwargs*)

Correlation of two random expressions, also known as correlation coefficient or Pearson’s correlation.

Explanation

The normalized expectation that the two variables will rise and fall together

$$correlation(X, Y) = E((X - E(X))(Y - E(Y))/(\sigma_x \sigma_y))$$

Examples

```
>>> from sympy.stats import Exponential, correlation
>>> from sympy import Symbol
```

```
>>> rate = Symbol('lambda', positive=True, real=True)
>>> X = Exponential('X', rate)
>>> Y = Exponential('Y', rate)
```

```
>>> correlation(X, X)
1
>>> correlation(X, Y)
0
>>> correlation(X, Y + rate*X)
1/sqrt(1 + lambda**(-2))
```

`sympy.stats.rv.`**`sampling_density`**(*expr, given_condition=None, library='scipy', numsamples=1, seed=None, **kwargs*)

Sampling version of density.

See also:

density (page 2963), *sampling_P* (page 2976), *sampling_E* (page 2976)

`sympy.stats.rv.`**`sampling_P`**(*condition, given_condition=None, library='scipy', numsamples=1, evalf=True, seed=None, **kwargs*)

Sampling version of P.

See also:

P (page 2960), *sampling_E* (page 2976), *sampling_density* (page 2976)

`sympy.stats.rv.`**`sampling_E`**(*expr, given_condition=None, library='scipy', numsamples=1, evalf=True, seed=None, **kwargs*)

Sampling version of E.

See also:

P (page 2960), *sampling_P* (page 2976), *sampling_density* (page 2976)

`class` `sympy.stats.`**`Moment`**(*X, n, c=0, condition=None, **kwargs*)

Symbolic class for Moment

Examples

```
>>> from sympy import Symbol, Integral
>>> from sympy.stats import Normal, Expectation, Probability, Moment
>>> mu = Symbol('mu', real=True)
>>> sigma = Symbol('sigma', positive=True)
>>> X = Normal('X', mu, sigma)
>>> M = Moment(X, 3, 1)
```

To evaluate the result of Moment use $doit$:

```
>>> M.doit()
mu**3 - 3*mu**2 + 3*mu*sigma**2 + 3*mu - 3*sigma**2 - 1
```

Rewrite the Moment expression in terms of Expectation:

```
>>> M.rewrite(Expectation)
Expectation((X - 1)**3)
```

Rewrite the Moment expression in terms of Probability:

```
>>> M.rewrite(Probability)
Integral((x - 1)**3*Probability(Eq(X, x)), (x, -oo, oo))
```

Rewrite the Moment expression in terms of Integral:

```
>>> M.rewrite(Integral)
Integral(sqrt(2)*(X - 1)**3*exp(-(X - mu)**2/(2*sigma**2))/
↪(2*sqrt(pi)*sigma), (X, -oo, oo))
```

`sympy.stats.`**`moment`**(*X, n, c=0, condition=None, *, evaluate=True, **kwargs*)

Return the nth moment of a random expression about c.

$$moment(X, c, n) = E((X - c)^n)$$

Default value of c is 0.

Examples

```
>>> from sympy.stats import Die, moment, E
>>> X = Die('X', 6)
>>> moment(X, 1, 6)
-5/2
>>> moment(X, 2)
91/6
>>> moment(X, 1) == E(X)
True
```

class `sympy.stats.`**`CentralMoment`**(*X, n, condition=None, **kwargs*)

Symbolic class Central Moment

Examples

```
>>> from sympy import Symbol, Integral
>>> from sympy.stats import Normal, Expectation, Probability,␣
↪CentralMoment
>>> mu = Symbol('mu', real=True)
>>> sigma = Symbol('sigma', positive=True)
>>> X = Normal('X', mu, sigma)
>>> CM = CentralMoment(X, 4)
```

To evaluate the result of CentralMoment use *doit*:

```
>>> CM.doit().simplify()
3*sigma**4
```

Rewrite the CentralMoment expression in terms of Expectation:

```
>>> CM.rewrite(Expectation)
Expectation((X - Expectation(X))**4)
```

Rewrite the CentralMoment expression in terms of Probability:

```
>>> CM.rewrite(Probability)
Integral((x - Integral(x*Probability(True), (x, -oo,␣
↪oo)))**4*Probability(Eq(X, x)), (x, -oo, oo))
```

Rewrite the CentralMoment expression in terms of Integral:

```
>>> CM.rewrite(Integral)
Integral(sqrt(2)*(X - Integral(sqrt(2)*X*exp(-(X - mu)**2/(2*sigma**2))/
↪(2*sqrt(pi)*sigma), (X, -oo, oo)))**4*exp(-(X - mu)**2/(2*sigma**2))/
↪(2*sqrt(pi)*sigma), (X, -oo, oo))
```

sympy.stats.**cmoment**(*X, n, condition=None, *, evaluate=True, **kwargs*)

Return the nth central moment of a random expression about its mean.

$$cmoment(X, n) = E((X - E(X))^n)$$

Examples

```
>>> from sympy.stats import Die, cmoment, variance
>>> X = Die('X', 6)
>>> cmoment(X, 3)
0
>>> cmoment(X, 2)
35/12
>>> cmoment(X, 2) == variance(X)
True
```

class sympy.stats.**ExpectationMatrix**(*expr, condition=None*)

Expectation of a random matrix expression.

Examples

```
>>> from sympy.stats import ExpectationMatrix, Normal
>>> from sympy.stats.rv import RandomMatrixSymbol
>>> from sympy import symbols, MatrixSymbol, Matrix
>>> k = symbols("k")
>>> A, B = MatrixSymbol("A", k, k), MatrixSymbol("B", k, k)
>>> X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
>>> ExpectationMatrix(X)
ExpectationMatrix(X)
>>> ExpectationMatrix(A*X).shape
(k, 1)
```

To expand the expectation in its expression, use `expand()`:

```
>>> ExpectationMatrix(A*X + B*Y).expand()
A*ExpectationMatrix(X) + B*ExpectationMatrix(Y)
>>> ExpectationMatrix((X + Y)*(X - Y).T).expand()
ExpectationMatrix(X*X.T) - ExpectationMatrix(X*Y.T) +␣
↪ExpectationMatrix(Y*X.T) - ExpectationMatrix(Y*Y.T)
```

To evaluate the `ExpectationMatrix`, use `doit()`:

```
>>> N11, N12 = Normal('N11', 11, 1), Normal('N12', 12, 1)
>>> N21, N22 = Normal('N21', 21, 1), Normal('N22', 22, 1)
>>> M11, M12 = Normal('M11', 1, 1), Normal('M12', 2, 1)
>>> M21, M22 = Normal('M21', 3, 1), Normal('M22', 4, 1)
>>> x1 = Matrix([[N11, N12], [N21, N22]])
>>> x2 = Matrix([[M11, M12], [M21, M22]])
>>> ExpectationMatrix(x1 + x2).doit()
Matrix([
[12, 14],
[24, 26]])
```

class `sympy.stats.`**`VarianceMatrix`**(*arg*, *condition=None*)

Variance of a random matrix probability expression. Also known as Covariance matrix, auto-covariance matrix, dispersion matrix, or variance-covariance matrix.

Examples

```
>>> from sympy.stats import VarianceMatrix
>>> from sympy.stats.rv import RandomMatrixSymbol
>>> from sympy import symbols, MatrixSymbol
>>> k = symbols("k")
>>> A, B = MatrixSymbol("A", k, k), MatrixSymbol("B", k, k)
>>> X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
>>> VarianceMatrix(X)
VarianceMatrix(X)
>>> VarianceMatrix(X).shape
(k, k)
```

To expand the variance in its expression, use `expand()`:

```
>>> VarianceMatrix(A*X).expand()
A*VarianceMatrix(X)*A.T
>>> VarianceMatrix(A*X + B*Y).expand()
2*A*CrossCovarianceMatrix(X, Y)*B.T + A*VarianceMatrix(X)*A.T +␣
↪B*VarianceMatrix(Y)*B.T
```

class sympy.stats.**CrossCovarianceMatrix**(*arg1*, *arg2*, *condition=None*)

Covariance of a random matrix probability expression.

Examples

```
>>> from sympy.stats import CrossCovarianceMatrix
>>> from sympy.stats.rv import RandomMatrixSymbol
>>> from sympy import symbols, MatrixSymbol
>>> k = symbols("k")
>>> A, B = MatrixSymbol("A", k, k), MatrixSymbol("B", k, k)
>>> C, D = MatrixSymbol("C", k, k), MatrixSymbol("D", k, k)
>>> X, Y = RandomMatrixSymbol("X", k, 1), RandomMatrixSymbol("Y", k, 1)
>>> Z, W = RandomMatrixSymbol("Z", k, 1), RandomMatrixSymbol("W", k, 1)
>>> CrossCovarianceMatrix(X, Y)
CrossCovarianceMatrix(X, Y)
>>> CrossCovarianceMatrix(X, Y).shape
(k, k)
```

To expand the covariance in its expression, use expand():

```
>>> CrossCovarianceMatrix(X + Y, Z).expand()
CrossCovarianceMatrix(X, Z) + CrossCovarianceMatrix(Y, Z)
>>> CrossCovarianceMatrix(A*X, Y).expand()
A*CrossCovarianceMatrix(X, Y)
>>> CrossCovarianceMatrix(A*X, B.T*Y).expand()
A*CrossCovarianceMatrix(X, Y)*B
>>> CrossCovarianceMatrix(A*X + B*Y, C.T*Z + D.T*W).expand()
A*CrossCovarianceMatrix(X, W)*D + A*CrossCovarianceMatrix(X, Z)*C +␣
↪B*CrossCovarianceMatrix(Y, W)*D + B*CrossCovarianceMatrix(Y, Z)*C
```

Mechanics

SymPy Stats employs a relatively complex class hierarchy.

RandomDomains are a mapping of variables to possible values. For example, we might say that the symbol Symbol('x') can take on the values $\{1, 2, 3, 4, 5, 6\}$.

class sympy.stats.rv.**RandomDomain**

A PSpace, or Probability Space, combines a RandomDomain with a density to provide probabilistic information. For example the above domain could be enhanced by a finite density {1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6} to fully define the roll of a fair die named x.

class sympy.stats.rv.**PSpace**

A RandomSymbol represents the PSpace's symbol 'x' inside of SymPy expressions.

class sympy.stats.rv.**RandomSymbol**

The RandomDomain and PSpace classes are almost never directly instantiated. Instead they are subclassed for a variety of situations.

RandomDomains and PSpaces must be sufficiently general to represent domains and spaces of several variables with arbitrarily complex densities. This generality is often unnecessary. Instead we often build SingleDomains and SinglePSpaces to represent single, univariate events and processes such as a single die or a single normal variable.

class sympy.stats.rv.**SinglePSpace**

class sympy.stats.rv.**SingleDomain**

Another common case is to collect together a set of such univariate random variables. A collection of independent SinglePSpaces or SingleDomains can be brought together to form a ProductDomain or ProductPSpace. These objects would be useful in representing three dice rolled together for example.

class sympy.stats.rv.**ProductDomain**

class sympy.stats.rv.**ProductPSpace**

The Conditional adjective is added whenever we add a global condition to a RandomDomain or PSpace. A common example would be three independent dice where we know their sum to be greater than 12.

class sympy.stats.rv.**ConditionalDomain**

We specialize further into Finite and Continuous versions of these classes to represent finite (such as dice) and continuous (such as normals) random variables.

class sympy.stats.frv.**FiniteDomain**

class sympy.stats.frv.**FinitePSpace**

class sympy.stats.crv.**ContinuousDomain**

class sympy.stats.crv.**ContinuousPSpace**

Additionally there are a few specialized classes that implement certain common random variable types. There is for example a DiePSpace that implements SingleFinitePSpace and a NormalPSpace that implements SingleContinuousPSpace.

class sympy.stats.frv_types.**DiePSpace**

class sympy.stats.crv_types.**NormalPSpace**

RandomVariables can be extracted from these objects using the PSpace.values method.

As previously mentioned SymPy Stats employs a relatively complex class structure. Inheritance is widely used in the implementation of end-level classes. This tactic was chosen to balance between the need to allow SymPy to represent arbitrarily defined random variables and optimizing for common cases. This complicates the code but is structured to only be important to those working on extending SymPy Stats to other random variable types.

Users will not use this class structure. Instead these mechanics are exposed through variable creation functions Die, Coin, FiniteRV, Normal, Exponential, etc.... These build the appropriate SinglePSpaces and return the corresponding RandomVariable. Conditional and Product spaces are formed in the natural construction of SymPy expressions and the use of interface functions E, Given, Density, etc....

`sympy.stats.Die()`

`sympy.stats.Normal()`

There are some additional functions that may be useful. They are largely used internally.

`sympy.stats.rv.random_symbols(expr)`

Returns all RandomSymbols within a SymPy Expression.

`sympy.stats.rv.pspace(expr)`

Returns the underlying Probability Space of a random expression.

For internal use.

Examples

```
>>> from sympy.stats import pspace, Normal
>>> X = Normal('X', 0, 1)
>>> pspace(2*X + 1) == X.pspace
True
```

`sympy.stats.rv.rs_swap(a, b)`

Build a dictionary to swap RandomSymbols based on their underlying symbol.

i.e. if `X = ('x', pspace1)` and `Y = ('x', pspace2)` then X and Y match and the key, value pair `{X:Y}` will appear in the result

Inputs: collections a and b of random variables which share common symbols Output: dict mapping RVs in a to RVs in b

CHAPTER SIX

CONTRIBUTING

The contributing guide goes over the details necessary to contribute to SymPy. See also the full Development Workflow guide on the SymPy wiki.

Content

6.1 Development Environment Setup

The first step to contributing to the code base is creating your development environment.

6.1.1 Git Setup

SymPy is available on GitHub and uses Git for source control. The workflow is such that code is pulled and pushed to and from the main repository. Install the respective version of Git for your operating system to start development.

Note: Refer to the installation instructions in the Git installation instructions. Learn about the basic git commands in this Git Handbook or any other sources on the internet.

6.1.2 Get the SymPy Code

It is recommended practice to create a fork of the SymPy project for your development purposes. Create your own fork of the SymPy project (if you have not yet). Go to the SymPy GitHub repository:

```
https://github.com/sympy/sympy
```

You will now have a fork at <https://github.com/<your-user-name>/sympy>.

Then, nn your machine browse to where you would like to store SymPy, and clone (download) the latest code from SymPy's original repository (about 77 MiB):

```
$ git clone https://github.com/<your-user-name>/sympy
```

You must configure the remote repositories for collaboration with the upstream project:

```
$ cd sympy
$ git remote add upstream https://github.com/sympy/sympy
```

After the configuration, your setup should be similar to this:

```
$ git remote -v
origin   https://github.com/<your-user-name>/sympy (fetch)
origin   https://github.com/<your-user-name>/sympy (push)
upstream https://github.com/sympy/sympy (fetch)
upstream https://github.com/sympy/sympy (push)
```

For further development, it is recommended to create a development branch.

```
$ git checkout -b dev-branch
```

The new branch can be of any name.

6.1.3 Virtual Environment Setup

You may want to take advantage of using virtual environments to isolate your development version of SymPy from any system wide installed versions, e.g. from `apt-get install python-sympy`.

We recommend using `conda` to create a virtual environment:

```
$ conda create -n sympy-dev python=3 mpmath flake8
```

You now have a environment that you can use for testing your development copy of SymPy. For example, clone your SymPy fork from Github:

```
$ git clone git@github.com:<your-github-username>/sympy.git
$ cd sympy
```

Now activate the environment:

```
$ conda activate sympy-dev
```

6.1.4 Run the Tests

There are several ways of running SymPy tests but the easiest is to use the `bin/test` script, consult 'the wiki details on running tests <https://github.com/sympy/sympy/wiki/Running-tests>`_.

The script takes a number of options and arguments and then passes them to `sympy.test(*paths, **kwargs)`. Run `bin/test --help` for all supported arguments.

Run all tests by using the command:

```
$ bin/test
```

To run tests for a specific file, use:

```
$ bin/test test_basic
```

Where `test_basic` is from file `sympy/core/basic.py`.

To run tests for modules, use:

```
$  bin/test /core /utilities
```

This will run tests for the `core` and `utilities` modules.

Similary, run quality tests with:

```
$ bin/test code_quality
```

6.2 Dependencies

This page lists the hard and optional dependencies of SymPy.

There are several packages that, when installed, can enable certain additional SymPy functionality. Most users and contributors will not need to install any of the packages mentioned below (except for the hard dependencies), unless they intend to use or contribute to the parts of SymPy that can use those packages.

Every dependency listed below can be installed with conda via conda-forge, and most can also be installed with `pip`.

This page does not list packages which themselves depend on SymPy, only those packages that SymPy depends on. An incomplete list of packages that depend on SymPy can be found on the main SymPy webpage, and a more complete list can be found on GitHub or libraries.io.

6.2.1 Hard Dependencies

SymPy only has one hard dependency, which is required for it to work: mpmath.

- **mpmath**: mpmath is a pure Python package for arbitrary precision arithmetic. It is used under the hood whenever SymPy calculates the floating-point value of a function, e.g., when using *evalf* (page 1065).

  SymPy cannot function without mpmath and will fail to import if it is not installed. If you get an error like

  ```
  ImportError: SymPy now depends on mpmath as an external library. See
  https://docs.sympy.org/latest/install.html#mpmath for more information.
  ```

  this means that you did not install mpmath correctly. *This page* (page 2) explains how to install it.

  Most methods of installing SymPy, such as the ones outlined in the *installation* (page 1) guide, will install mpmath automatically. You typically only need to install mpmath manually if you did not actually install SymPy, e.g., if you are developing directly on SymPy in the git repository.

6.2.2 Optional Dependencies

These dependencies are not required to use SymPy. The vast majority of SymPy functions do not require them, however, a few functions such as plotting and automatic wrapping of code generated functions require additional dependencies to function.

Additionally, as a contributor, when running the SymPy tests, some tests will be skipped if a dependency they require is not installed. The GitHub Actions CI which is run on every SymPy pull request will automatically install these dependencies in the "optional-dependencies" build, but you may wish to install them locally if you are working on a part of SymPy that uses them.

Recommended Optional Dependencies

These dependencies are not required for SymPy to function, but it is recommended that all users install them if they can, as they will improve the general performance of SymPy.

- **gmpy2**: gmpy2 is a Python wrapper for the GMP multiple-precision library. It provides large integers that are faster than the built-in Python `int`. When gmpy2 is installed, it is used automatically by certain core functions that operate on integers, such as the *polys* (page 2341). See *Reference docs for the Poly Domains* (page 2504) for more details. SymPy uses `gmpy2` automatically when it is installed. No further action is required to enable it.

  The polys themselves are used by many parts of SymPy, such as the integration algorithms, simplification algorithms like `collect()` and `factor()`, the matrices, and some parts of the core. Thus, installing `gmpy2` can speed up many parts of SymPy. It is not a required dependency of SymPy because it makes use of a non-Python library (GMP), which is also non-BSD licensed. However, we recommended all users who are able to to install `gmpy2` to get a better SymPy experience.

Interactive Use

SymPy is designed to be used both interactively and as a library. When used interactively, SymPy is able to interface with IPython and Jupyter notebooks.

- **IPython**: The *`init_session()`* (page 2117) function and `isympy` command will automatically start IPython if it is installed. In addition to the usual benefits of using IPython, this enables interactive plotting with matplotlib. Also some flags such as `auto_symbols` and `auto_int_to_Integer` will only work in IPython.

  The `IPython` package is required to run some of the tests in sympy/interactive.

- **Jupyter Notebook and Qt Console**: SymPy expressions automatically print using MathJax in the Jupyter Notebook and with LaTeX Qt Console (if *LaTeX* (page 2987) is installed).

Printing

The *preview()* (page 2181) function automatically converts SymPy expressions into images rendered with LaTeX. `preview()` can either save the image to a file or show it with a viewer.

- **LaTeX**: A LaTeX distributions such as TeXLive or MiKTeX is required for *preview()* (page 2181) to function.

Parsing

Several functions in the *sympy.parsing* (page 2122) submodule require external dependencies to function. Note that not all parsers require external modules at this time. The Python (*parse_expr()* (page 2122)), Mathematca (*parse_mathematica()* (page 2125)), and Maxima (*parse_maxima()* (page 2125)) parsers do not require any external dependencies.

- **antlr-python-runtime**: Antlr is used for the *LaTeX parser* (page 2129) and *Autolev* (page 1724) parsers. They both require the Antlr Python runtime to be installed. The package for this is called `antlr-python-runtime` with conda and `antlr4-python3-runtime` with pip). Also be aware that the version of the Antlr Python runtime must match the version that was used to compile the LaTeX and Autolev parsers (4.10).
- **Clang Python Bindings**: The C parser (`sympy.parsing.c.parse_c`) requires the Clang Python bindings. The package for this is called `python-clang` with conda and `clang` with pip.
- **lfortran**: The Fortran parser (in `sympy.parsing.fortran`) requires LFortran.

Logic

The *satisfiable()* (page 1184) function includes a pure Python implementation of the DPLL satisfiability algorithm. But it can optionally use faster C SAT solvers if they are installed. Note that `satisfiable()` is also used by *ask()* (page 191).

- **pycosat**: Pycosat is used automatically if it is installed. The use of pycosat can be forced by using `satisfiable(algorithm='pycosat')`.
- **pysat**: Pysat is a library which wraps many SAT solvers. It can also be used as a backend to `satisfiable()`. Presently, only Minisat is implemented, using `satisfiable(algorithm=minisat22')`.

Plotting

The *sympy.plotting.plot* (page 2820) module makes heavy use of external plotting libraries to render plots. The primarily plotting module that is supported is Matplotlib.

- **matplotlib**: Most plotting functionality requires the Matplotlib plotting library. Without Matplotlib installed, most plotting functions will either fail or give rudimentary *text plots* (page 2872).
- **pyglet**: SymPy has a submodule *sympy.plotting.pygletplot* (page 2869) that can be used to interface with the pyglet module to do 2D and 3D plotting.

lambdify

lambdify() (page 2100) is a function that converts SymPy expressions into functions that can be evaluated numerically using various libraries as backends. `lambdify` is the primary vehicle by which users interface between SymPy and these libraries. It is the standard way to convert a symbolic SymPy expression into an evaluable numeric function.

In principle, `lambdify` can interface with any external library if the user passes in an appropriate namespace dictionary as the third argument, but by default, `lambdify` is aware of several popular numeric Python libraries. These libraries are enabled as backends in `lambdify` with build-in translations to convert SymPy expressions into the appropriate functions for those libraries.

- **NumPy**: By default, if it is installed, `lambdify` creates functions using NumPy (if NumPy is not installed, `lambdify` produces functions using the standard library math module, although this behavior is primarily provided for backwards compatibility).
- **SciPy**: If SciPy is installed, `lambdify` will use it automatically. SciPy is needed to lambdify certain special functions that are not included in NumPy.
- **CuPy**: CuPy is a library that provides a NumPy compatible interface for CUDA GPUs. `lambdify` can produce CuPy compatible functions using `lambdify(modules='cupy')`.
- **Jax**: JAX is a library that uses XLA to compile and run NumPy programs on GPUs and TPUs. `lambdify` can produce JAX compatibly functions using `lambdify(modules='jax')`.
- **TensorFlow**: TensorFlow is a popular machine learning library. `lambdify` can produce TensorFlow compatible functions using `lambdify(modules='tensorflow')`.
- **NumExpr**: NumExpr is a fast numerical expression evaluator for NumPy. `lambdify` can produce NumExpr compatible functions using `lambdify(modules='numexpr')`.
- **mpmath**: `lambdify` can also produce mpmath compatible functions. Note that mpmath is already a *required dependency* (page 2985) of SymPy. This functionality is useful for converting a SymPy expression to a function for use with pure mpmath.

Code Generation

SymPy can *generate code* (page 1108) for a large number of languages by converting SymPy expressions into valid code for those languages. It also has functionality for some languages to automatically compile and run the code.

Note that the dependencies below are **not** a list of supported languages that SymPy can generate code for. Rather it is a list of packages that SymPy can interface with in some way. For most languages that SymPy supports code generation, it simply generates a string representing the code for that language, so no dependency on that language is required to use the code generation functionality. A dependency is typically only required for features that automatically take the generated code and compile it to a function that can be used within Python. Note that *lambdify()* (page 2100) is a special case of this, but its dependencies are listed *above* (page 2988).

Autowrap

- **NumPy**: NumPy and, optionally, its subpackage f2py, can be used to generate Python functions using the *autowrap()* (page 2041) or *ufuncify()* (page 2043) functions.
- **Cython**: Cython can be used as a backend for *autowrap()* (page 2041) or *ufuncify()* (page 2043). Cython is also used in some of the sympy.codegen tests to compile some examples.
- **Compilers**: *autowrap()* (page 2041), *ufuncify()* (page 2043), and related functions rely on a compiler to compile the generated code to a function. Most standard C, C++, and Fortran compilers are supported, including Clang/LLVM, GCC, and ifort.

Code Printers

Most code printers generate Python strings, and therefore do not require the given library or language compiler as a dependency. However, a few code printers generate Python functions instead of strings:

- **Aesara**: The *sympy.printing.aesaracode* (page 2166) module contains functions to convert SymPy expressions into a functions using the Aeseara (previously Theano) library. The Aesara code generation functions return Aesara graph objects.
- **llvmlite**: The sympy.printing.llvmjitcode module supports generating LLVM Jit from a SymPy expression. The functions make use of llvmlite, a Python wrapper around LLVM. The llvm_callable() function generates callable functions.
- **TensorFlow**: The sympy.printing.tensorflow module supports generating functions using the TensorFlow, a popular machine learning library. Unlike the above two examples, tensorflow_code() function **does** generate Python strings. However, tensorflow is imported if available in order to automatically detect the TensorFlow version. If it is not installed, the tensorflow_code() function assumes the latest supported version of TensorFlow.

Testing-Only Dependencies

- **Wurlitzer**: Wurlitzer is a Python package that allows capturing output from C extensions. It is used by some of the tests in the sympy.codegen submodule. It is only used by the test suite. It is not used by any end-user functionality. If it is not installed, some tests will be skipped.
- **Cython**: Cython is also used in some of the sympy.codegen tests to compile some examples.
- **Compilers**: The various *compilers* (page 2989) mentioned above are used in some of the codegen and autowrap tests if they are installed.

Statistics

The *sympy.stats.sample()* (page 2971) function uses an external library to produce samples from the given distribution. At least one of the following libraries is required to use the sampling functionality of `sympy.stats`.

- **SciPy**: `sample(library='scipy')` is the default. This uses scipy.stats.
- **NumPy**: `sample(library='numpy')` uses the NumPy random module.
- **pymc**: `sample(library='pymc')` uses PyMC to do sampling.

Optional SymEngine Backend

- **python-symengine**: SymEngine is a fast symbolic manipulation library, written in C++. The SymEngine Python bindings may be used as an optional backend for SymPy core. To do this, first install the SymEngine Python bindings (with `pip install symengine` or `conda install -c conda-forge python-symengine`) and run SymPy with the `USE_SYMENGINE=1` environment variable.

  Presently, the SymEngine backend is only used by the *sympy.physics.mechanics* (page 1675) and *sympy.liealgebras* (page 2326) modules, although you can also interface with SymPy's SymEngine backend directly by importing things from `sympy.core.backend`:

  ```
  >>> from sympy.core.backend import Symbol
  >>> # This will create a SymEngine Symbol object if the USE_SYMENGINE
  >>> # environment variable is configured. Otherwise it will be an␣
  ↪ordinary
  >>> # SymPy Symbol object.
  >>> x = Symbol('x')
  ```

  SymEngine backend support is still experimental, so certain SymPy functions may not work correctly when it is enabled.

Experimental Rubi Integrator

- **MatchPy**: MatchPy is a library for doing pattern matching. It is used in the experimental sympy.integrals.rubi module, but presently, it is not used anywhere else in SymPy. SymPy and MatchPy are able to interface with each other.

Sage

Sage is an open source mathematics software that incorporates a large number of open source mathematics libraries. SymPy is one of the libraries used by Sage.

Most of the code that interfaces between SymPy and Sage is in Sage itself, but a few `_sage_` methods in SymPy that do some very basic setting up of the Sage/SymPy wrappers. These methods should typically only be called by Sage itself.

6.2.3 Development Dependencies

Typical development on SymPy does not require any additional dependencies beyond Python and mpmath.

Getting the Source Code

- **git**: The SymPy source code uses the git version control system. See the *installation guide* (page 1) and development workflow for instructions on how to get the development version of SymPy from git.

Running the Tests

The base SymPy tests do not require any additional dependencies, however most of the above dependencies may be required for some tests to run. Tests that depend on optional dependencies should be skipped when they are not installed, either by using the `sympy.testing.pytest.skip()` function or by setting `skip = True` to skip the entire test file. Optional modules in tests and SymPy library code should be imported with `import_module()`.

- **pytest**: Pytest is not a required dependency for the SymPy test suite. SymPy has its own test runner, which can be accessed via the `bin/test` script in the SymPy source directory or the *`test()`* (page 2035) function.

  However, if you prefer to use pytest, you can use it to run the tests instead of the SymPy test runner. Tests in SymPy should use the wrappers in *`sympy.testing.pytest`* (page 2027) instead of using pytest functions directly.

- **Cloudpickle**: The cloudpickle package can be used to more effectively pickle SymPy objects than the built-in Python pickle. Some tests in `sympy.utilities.tests.test_pickling.py` depend on cloudpickle to run. It is not otherwise required for any SymPy function.

Building the Documentation

Building the documentation requires several additional dependencies. *This page* (page 2992) outlines these dependencies and how to install them. It is only necessary to install these dependencies if you are contributing documentation to SymPy and want to check that the HTML or PDF documentation renders correctly. If you only want to view the documentation for the development version of SymPy, development builds of the docs are hosted online at https://docs.sympy.org/dev/index.html.

Running the Benchmarks

The benchmarks for SymPy are hosted at https://github.com/sympy/sympy_benchmarks. The README in that repository explains how to run the benchmarks.

Note that the benchmarks are also run automatically on the GitHub Actions CI, so it is generally not necessary to run them yourself as a contributor unless you want to reproduce the benchmarks results on your computer or add a new benchmark to the suite.

- **asv**: Airspeed Velocity is the package used for running the benchmarks. Note that the package name that you install is called `asv`.

6.3 Build the Documentation

Start by installing the required dependencies for the documentation.

6.3.1 Required dependencies

You can either install the dependencies locally on your machine, or you can build a Docker image containing them.

Docker

If you have Docker, then instead of following the OS-specific installation instructions below, you may choose to build a Docker image:

```
cd doc
docker build -f Dockerfile.htmldoc -t sympy_htmldoc .
```

If you choose this option, you can now skip down to the "Build the Docs" section below.

Debian/Ubuntu

For Debian/Ubuntu:

```
apt-get install python3-sphinx texlive-latex-recommended dvipng librsvg2-bin␣
↪imagemagick docbook2x graphviz
python -m pip install -r doc/requirements.txt
```

If you get mpmath error, install python-mpmath package:

```
apt-get install python-mpmath
```

If you get matplotlib error, install python-matplotlib package:

```
apt-get install python-matplotlib
```

Fedora

For Fedora (and maybe other RPM-based distributions), install the prerequisites:

```
dnf install python3-sphinx librsvg2 ImageMagick docbook2X texlive-dvipng-bin
texlive-scheme-medium librsvg2-tools
python -m pip install -r doc/requirements.txt
```

If you get mpmath error, install python3-mpmath package:

```
dnf install python3-mpmath
```

If you get matplotlib error, install python3-matplotlib package:

```
dnf install python3-matplotlib
```

Mac

For Mac, first install homebrew: https://brew.sh/

Then install these packages with homebrew:

```
brew install imagemagick graphviz docbook librsvg
```

Install the docs dependencies with either pip or conda:

```
python -m pip install -r requirements.txt
```

Or:

```
conda install -c conda-forge --file requirements.txt
```

Making your Sphinx build successful on the Windows system is tricky because some dependencies like `dvipng` or `docbook2x` are not available.

Windows 10

For Windows 10, however, the Windows Subsystem for Linux can be a possible workaround solution, and you can install Ubuntu shell on your Windows system after following the tutorial below:

https://github.com/MicrosoftDocs/WSL/blob/live/WSL/install-win10.md

In your command prompt, run `ubuntu` to transfer to Linux terminal, and follow the Debian/Ubuntu tutorial above to install the dependencies, and then you can run `make html` to build. (Note that you also have to install `make` via `apt-get install make`.)

If you want to change the directory in your prompt to your working folder of SymPy in the Windows file system, you can prepend `cd /mnt/` to your file path in Windows, and run in your shell to navigate to the folder. (Also note that Linux uses `/` instead of `\` for file paths.)

This method provides better compatibility than Cygwin or MSYS2 and more convenience than a virtual machine if you partially need a Linux environment for your workflow, however this method is only viable for Windows 10 64-bit users.

or

Follow instruction to install Chocolatey

Install make and other dependencies:

```
choco install make graphviz rsvg-convert imagemagick
```

Install python dependencies:

```
pip install -r doc/requirements.txt
```

6.3.2 Build the Docs

Docker

If you chose to build using Docker, and followed the instructions above to build the sympy_htmldoc image, then you can build the docs with:

```
docker run --rm -v /absolute/path/to/sympy:/sympy sympy_htmldoc
```

(Be sure to substitute the actual absolute filesystem path to sympy!) This command can be run from any directory.

Local Installation

If you chose to follow OS-specific instructions above and installed the required dependencies locally, the documentation can be built by running the makefile in the doc subdirectory:

```
cd doc
make html
```

SymPy Logos

SymPy has a collection of official logos, which can be generated from sympy.svg in your local copy of SymPy by:

```
$ cd doc
$ make logo # will be stored in the _build/logo subdirectory
```

The license of all the logos is the same as SymPy: BSD. See the LICENSE file for more information.

6.3.3 View the Docs

Once you have built the docs, the generated files will be found under doc/_build/html. To view them in your preferred web browser, use the drop down menu and select "open file", navigate into the sympy/doc/_build/html folder, and open the index.html file.

6.3.4 Auto-Rebuild with the Live Server

The instructions given above told you how to build the docs once, and load them in the browser. After you make changes to the document sources, you'll have to manually repeat the build step, and reload the pages in the browser.

There is an alternative approach that sets up a live server, which will monitor the docs directory, automatically rebuild when changes are detected, and automatically reload the page you are viewing in the browser.

If you want to use this option, the procedure again depends on whether you are using Docker, or a local installation.

Docker

To start the live server with Docker, you can use:

```
docker run --rm -it \
     -v /absolute/path/to/sympy:/sympy \
     -p 8000:80 \
     sympy_htmldoc live
```

and then navigate your browser to `localhost:8000`. You can use a different port by changing the `8000` in the command. Again, be sure to substitute the actual absolute filesystem path to sympy.

When finished, you can stop the server with `ctrl-c` in the terminal.

Alternatively, you may run the server in detached mode, using:

```
docker run --rm -d --name=sympy-livehtml \
     -v /absolute/path/to/sympy:/sympy \
     -p 8000:80 \
     sympy_htmldoc live
```

and then stop it with:

```
docker stop sympy-livehtml
```

Local Installation

If you installed the build dependencies locally, then simply use:

```
cd doc
make livehtml
```

to start the server. Your web browser should then automatically open a new tab, showing the index page of the SymPy docs.

When you are finished, you can use `ctrl-c` in the terminal to stop the server.

6.3.5 PDF Documentation

Note: It is not necessary for the majority of contributors to build the PDF documentation. The PDF documentation will be built automatically on GitHub Actions on pull requests. PDF documentation for each release is included on the GitHub releases page.

If the PDF documentation build fails on GitHub Actions, 99% of the time this is due to bad LaTeX math formatting. Double check that any math you have added is formatted correctly, and make sure you use ``double backticks`` for code (`single backticks` will render as math, not code). See the resources in the *style guide* (page 3013) for tips on formatting LaTeX math.

Building the PDF documentation requires a few extra dependencies. First you will need to have a TeXLive installation that includes XeLaTeX and latexmk. You will also need to have Chrome or Chromium installed, as it is used to convert some SVG files for the PDF.

On Ubuntu, you can install these with:

```
apt-get install chromium-browser texlive texlive-xetex texlive-fonts-
↪recommended texlive-latex-extra latexmk lmodern
```

On Mac, you can use:

```
brew install texlive
brew install --cask chromium
brew tap homebrew/cask-fonts
brew install font-dejavu
```

On Windows 10, you can use:

```
choco install chromium strawberryperl miktex dejavufonts
```

If DejaVu fonts are not installed in `C:\Windows\Fonts`, then open `~\AppData\Local\Microsoft\Windows\Fonts`, select all DejaVu fonts, right-click and click `Install for all users`.

To build the pdf docs run:

```
cd doc
make pdf
```

The resulting PDF will be in:

```
_build/latex/sympy-<version>.pdf
```

where `<version>` is the SymPy version (e.g., `sympy-1.10.dev.pdf`).

6.4 Debugging

To start sympy in debug mode set the SYMPY_DEBUG variable. For instance in a unix-like system you would do

$ SYMPY_DEBUG=True bin/isympy

or in Windows

> set SYMPY_DEBUG=True > python bin/isympy

Now just use for example the `limit()` function. You will get a nice printed tree, which is very useful for debugging.