万兴PDF

Turn on post-mortem pdb:

```
>>> sympy.test(pdb=True)
```

Turn off colors:

```
>>> sympy.test(colors=False)
```

Force colors, even when the output is not to a terminal (this is useful, e.g., if you are piping to `less -r` and you still want colors)

```
>>> sympy.test(force_colors=False)
```

The traceback verboseness can be set to "short" or "no" (default is "short")

```
>>> sympy.test(tb='no')
```

The `split` option can be passed to split the test run into parts. The split currently only splits the test files, though this may change in the future. `split` should be a string of the form 'a/b', which will run part `a` of `b`. For instance, to run the first half of the test suite:

```
>>> sympy.test(split='1/2')
```

The `time_balance` option can be passed in conjunction with `split`. If `time_balance=True` (the default for `sympy.test`), SymPy will attempt to split the tests such that each split takes equal time. This heuristic for balancing is based on pre-recorded test data.

```
>>> sympy.test(split='1/2', time_balance=True)
```

You can disable running the tests in a separate subprocess using `subprocess=False`. This is done to support seeding hash randomization, which is enabled by default in the Python versions where it is supported. If subprocess=False, hash randomization is enabled/disabled according to whether it has been enabled or not in the calling Python process. However, even if it is enabled, the seed cannot be printed unless it is called from a new Python process.

Hash randomization was added in the minor Python versions 2.6.8, 2.7.3, 3.1.5, and 3.2.3, and is enabled by default in all Python versions after and including 3.3.0.

If hash randomization is not supported `subprocess=False` is used automatically.

```
>>> sympy.test(subprocess=False)
```

To set the hash randomization seed, set the environment variable `PYTHONHASHSEED` before running the tests. This can be done from within Python using

```
>>> import os
>>> os.environ['PYTHONHASHSEED'] = '42'
```

Or from the command line using

$ PYTHONHASHSEED=42 ./bin/test

If the seed is not set, a random seed will be chosen.

Note that to reproduce the same hash values, you must use both the same seed as well as the same architecture (32-bit vs. 64-bit).

万兴PDF

Utilities

This module contains some general purpose utilities that are used across SymPy.

Contents:

Autowrap Module

The autowrap module works very well in tandem with the Indexed classes of the *Tensor* (page 1387). Here is a simple example that shows how to setup a binary routine that calculates a matrix-vector product.

```
>>> from sympy.utilities.autowrap import autowrap
>>> from sympy import symbols, IndexedBase, Idx, Eq
>>> A, x, y = map(IndexedBase, ['A', 'x', 'y'])
>>> m, n = symbols('m n', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> instruction = Eq(y[i], A[i, j]*x[j]); instruction
Eq(y[i], A[i, j]*x[j])
```

Because the code printers treat Indexed objects with repeated indices as a summation, the above equality instance will be translated to low-level code for a matrix vector product. This is how you tell SymPy to generate the code, compile it and wrap it as a python function:

```
>>> matvec = autowrap(instruction)
```

万兴PDF

That's it. Now let's test it with some numpy arrays. The default wrapper backend is f2py. The wrapper function it provides is set up to accept python lists, which it will silently convert to numpy arrays. So we can test the matrix vector product like this:

```
>>> M = [[0, 1],
...      [1, 0]]
>>> matvec(M, [2, 3])
[ 3.  2.]
```

Implementation details

The autowrap module is implemented with a backend consisting of CodeWrapper objects. The base class `CodeWrapper` takes care of details about module name, filenames and options. It also contains the driver routine, which runs through all steps in the correct order, and also takes care of setting up and removing the temporary working directory.

The actual compilation and wrapping is done by external resources, such as the system installed f2py command. The Cython backend runs a distutils setup script in a subprocess. Subclasses of CodeWrapper takes care of these backend-dependent details.

万兴PDF

API Reference

Module for compiling codegen output, and wrap the binary for use in python.

Note: To use the autowrap module it must first be imported

```
>>> from sympy.utilities.autowrap import autowrap
```

This module provides a common interface for different external backends, such as f2py, fwrap, Cython, SWIG(?) etc. (Currently only f2py and Cython are implemented) The goal is to provide access to compiled binaries of acceptable performance with a one-button user interface, e.g.,

```
>>> from sympy.abc import x,y
>>> expr = (x - y)**25
>>> flat = expr.expand()
>>> binary_callable = autowrap(flat)
>>> binary_callable(2, 3)
-1.0
```

Although a SymPy user might primarily be interested in working with mathematical expressions and not in the details of wrapping tools needed to evaluate such expressions efficiently in numerical form, the user cannot do so without some understanding of the limits in the target language. For example, the expanded expression contains large coefficients which result in loss of precision when computing the expression:

```
>>> binary_callable(3, 2)
0.0
>>> binary_callable(4, 5), binary_callable(5, 4)
(-22925376.0, 25165824.0)
```

万兴PDF

Wrapping the unexpanded expression gives the expected behavior:

```
>>> e = autowrap(expr)
>>> e(4, 5), e(5, 4)
(-1.0, 1.0)
```

The callable returned from autowrap() is a binary Python function, not a SymPy object. If it is desired to use the compiled function in symbolic expressions, it is better to use binary_function() which returns a SymPy Function object. The binary callable is attached as the _imp_ attribute and invoked when a numerical evaluation is requested with evalf(), or with lambdify().

```
>>> from sympy.utilities.autowrap import binary_function
>>> f = binary_function('f', expr)
>>> 2*f(x, y) + y
y + 2*f(x, y)
>>> (2*f(x, y) + y).evalf(2, subs={x: 1, y:2})
0.e-110
```

When is this useful?

1) For computations on large arrays, Python iterations may be too slow, and depending on the mathematical expression, it may be difficult to exploit the advanced index operations provided by NumPy.

万兴PDF

2) For *really* long expressions that will be called repeatedly, the compiled binary should be significantly faster than SymPy's .evalf()
3) If you are generating code with the codegen utility in order to use it in another project, the automatic Python wrappers let you test the binaries immediately from within SymPy.
4) To create customized ufuncs for use with numpy arrays. See *ufuncify*.

When is this module NOT the best approach?

1) If you are really concerned about speed or memory optimizations, you will probably get better results by working directly with the wrapper tools and the low level code. However, the files generated by this utility may provide a useful starting point and reference code. Temporary files will be left intact if you supply the keyword tempdir="path/to/files/".
2) If the array computation can be handled easily by numpy, and you do not need the binaries for another project.

class sympy.utilities.autowrap.**CodeWrapper**(*generator*, *filepath=None*, *flags=[]*, *verbose=False*)

Base Class for code wrappers

class sympy.utilities.autowrap.**CythonCodeWrapper**(**args*, ***kwargs*)

Wrapper that uses Cython

dump_pyx(*routines*, *f*, *prefix*)

Write a Cython file with Python wrappers

This file contains all the definitions of the routines in c code and refers to the header file.

万兴PDF

Arguments

routines
List of Routine instances

f
File-like object to write the file to

prefix
The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

class sympy.utilities.autowrap.**DummyWrapper**(*generator*, *filepath=None*, *flags=[]*, *verbose=False*)

Class used for testing independent of backends

class sympy.utilities.autowrap.**F2PyCodeWrapper**(**args*, ***kwargs*)

Wrapper that uses f2py

class sympy.utilities.autowrap.**UfuncifyCodeWrapper**(**args*, ***kwargs*)

Wrapper for Ufuncify

dump_c(*routines*, *f*, *prefix*, *funcname=None*)

Write a C file with Python wrappers

This file contains all the definitions of the routines in c code.

万兴PDF

Arguments

routines
List of Routine instances

f
File-like object to write the file to

prefix
The filename prefix, used to name the imported module.

funcname
Name of the main function to be returned.

`sympy.utilities.autowrap.`**`autowrap`**(*expr*, *language=None*, *backend='f2py'*, *tempdir=None*, *args=None*, *flags=None*, *verbose=False*, *helpers=None*, *code_gen=None*, ***kwargs*)

Generates Python callable binaries based on the math expression.

Parameters

expr

The SymPy expression that should be wrapped as a binary routine.

language : string, optional

If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If `None` [default], the language is inferred based upon the specified backend.

backend : string, optional

Backend used to wrap the generated code. Either 'f2py' [default], or 'cython'.

tempdir : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

args : iterable, optional

An ordered iterable of symbols. Specifies the argument sequence for the function.

flags : iterable, optional

Additional option flags that will be passed to the backend.

verbose : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

helpers : 3-tuple or iterable of 3-tuples, optional

Used to define auxiliary expressions needed for the main expr. If the main expression needs to call a specialized function it should be passed in via `helpers`. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should

万兴PDF

万兴PDF

be 3-tuples with (<function_name>, <sympy_expression>, <argument_tuple>). It is mandatory to supply an argument sequence to helper routines.

code_gen : CodeGen instance

An instance of a CodeGen subclass. Overrides `language`.

include_dirs : [string]

A list of directories to search for C/C++ header files (in Unix form for portability).

library_dirs : [string]

A list of directories to search for C/C++ libraries at link time.

libraries : [string]

A list of library names (not filenames or paths) to link against.

extra_compile_args : [string]

Any extra platform- and compiler-specific information to use when compiling the source files in 'sources'. For platforms and compilers where "command line" makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.

extra_link_args : [string]

Any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for 'extra_compile_args'.

万兴PDF

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.autowrap import autowrap
>>> expr = ((x - y + z)**(13)).expand()
>>> binary_func = autowrap(expr)
>>> binary_func(1, 4, 2)
-1.0
```

sympy.utilities.autowrap.**binary_function**(*symfunc*, *expr*, ***kwargs*)

Returns a SymPy function with expr as binary implementation

This is a convenience function that automates the steps needed to autowrap the SymPy expression and attaching it to a Function object with implemented_function().

Parameters

symfunc : SymPy Function

The function to bind the callable to.

expr : SymPy Expression

The expression used to generate the function.

kwargs : dict

万兴PDF

Any kwargs accepted by autowrap.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.utilities.autowrap import binary_function
>>> expr = ((x - y)**(25)).expand()
>>> f = binary_function('f', expr)
>>> type(f)
<class 'sympy.core.function.UndefinedFunction'>
>>> 2*f(x, y)
2*f(x, y)
>>> f(x, y).evalf(2, subs={x: 1, y: 2})
-1.0
```

`sympy.utilities.autowrap.`**`ufuncify`**(*args, expr, language=None, backend='numpy', tempdir=None, flags=None, verbose=False, helpers=None, **kwargs*)

Generates a binary function that supports broadcasting on numpy arrays.

Parameters

args : iterable

Either a Symbol or an iterable of symbols. Specifies the argument sequence for the function.

expr

A SymPy expression that defines the element wise operation.

language : string, optional

If supplied, (options: 'C' or 'F95'), specifies the language of the generated code. If `None` [default], the language is inferred based upon the specified backend.

backend : string, optional

Backend used to wrap the generated code. Either 'numpy' [default], 'cython', or 'f2py'.

tempdir : string, optional

Path to directory for temporary files. If this argument is supplied, the generated code and the wrapper input files are left intact in the specified path.

flags : iterable, optional

Additional option flags that will be passed to the backend.

verbose : bool, optional

If True, autowrap will not mute the command line backends. This can be helpful for debugging.

helpers : iterable, optional

Used to define auxiliary expressions needed for the main expr. If the main expression needs to call a specialized function it should be put in

万兴PDF

万兴PDF

the `helpers` iterable. Autowrap will then make sure that the compiled main expression can link to the helper routine. Items should be tuples with (<funtion_name>, <sympy_expression>, <arguments>). It is mandatory to supply an argument sequence to helper routines.

kwargs : dict

These kwargs will be passed to autowrap if the $f2py$ or $cython$ backend is used and ignored if the $numpy$ backend is used.

Notes

The default backend ('numpy') will create actual instances of `numpy.ufunc`. These support ndimensional broadcasting, and implicit type conversion. Use of the other backends will result in a "ufunc-like" function, which requires equal length 1-dimensional arrays for all arguments, and will not perform any type conversions.

Examples

```
>>> from sympy.utilities.autowrap import ufuncify
>>> from sympy.abc import x, y
>>> import numpy as np
>>> f = ufuncify((x, y), y + x**2)
>>> type(f)
<class 'numpy.ufunc'>
>>> f([1, 2, 3], 2)
array([  3.,   6.,  11.])
>>> f(np.arange(5), 3)
array([  3.,   4.,   7.,  12.,  19.])
```

万兴PDF

For the 'f2py' and 'cython' backends, inputs are required to be equal length 1-dimensional arrays. The 'f2py' backend will perform type conversion, but the Cython backend will error if the inputs are not of the expected type.

```
>>> f_fortran = ufuncify((x, y), y + x**2, backend='f2py')
>>> f_fortran(1, 2)
array([ 3.])
>>> f_fortran(np.array([1, 2, 3]), np.array([1.0, 2.0, 3.0]))
array([  2.,   6.,  12.])
>>> f_cython = ufuncify((x, y), y + x**2, backend='Cython')
>>> f_cython(1, 2)
Traceback (most recent call last):
  ...
TypeError: Argument '_x' has incorrect type (expected numpy.ndarray, got␣
↪int)
>>> f_cython(np.array([1.0]), np.array([2.0]))
array([ 3.])
```

万兴PDF

References

[R957]

Codegen

This module provides functionality to generate directly compilable code from SymPy expressions. The `codegen` function is the user interface to the code generation functionality in SymPy. Some details of the implementation is given below for advanced users that may want to use the framework directly.

Note: The `codegen` callable is not in the sympy namespace automatically, to use it you must first execute

```
>>> from sympy.utilities.codegen import codegen
```

Implementation Details

Here we present the most important pieces of the internal structure, as advanced users may want to use it directly, for instance by subclassing a code generator for a specialized application. **It is very likely that you would prefer to use the codegen() function documented above.**

Basic assumptions:

- A generic Routine data structure describes the routine that must be translated into C/Fortran/... code. This data structure covers all features present in one or more of the supported languages.
- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

Routine

The Routine class is a very important piece of the codegen module. Viewing the codegen utility as a translator of mathematical expressions into a set of statements in a programming language, the Routine instances are responsible for extracting and storing information about how the math can be encapsulated in a function call. Thus, it is the Routine constructor that decides what arguments the routine will need and if there should be a return value.

万兴PDF

万兴PDF

API Reference

module for generating C, C++, Fortran77, Fortran90, Julia, Rust and Octave/Matlab routines that evaluate SymPy expressions. This module is work in progress. Only the milestones with a '+' character in the list below have been completed.

— How is sympy.utilities.codegen different from sympy.printing.ccode? —

We considered the idea to extend the printing routines for SymPy functions in such a way that it prints complete compilable code, but this leads to a few unsurmountable issues that can only be tackled with dedicated code generator:

- For C, one needs both a code and a header file, while the printing routines generate just one string. This code generator can be extended to support .pyf files for f2py.
- SymPy functions are not concerned with programming-technical issues, such as input, output and input-output arguments. Other examples are contiguous or non-contiguous arrays, including headers of other libraries such as gsl or others.
- It is highly interesting to evaluate several SymPy functions in one C routine, eventually sharing common intermediate results with the help of the cse routine. This is more than just printing.
- From the programming perspective, expressions with constants should be evaluated in the code generator as much as possible. This is different for printing.

— Basic assumptions —

- A generic Routine data structure describes the routine that must be translated into C/Fortran/… code. This data structure covers all features present in one or more of the supported languages.
- Descendants from the CodeGen class transform multiple Routine instances into compilable code. Each derived class translates into a specific language.
- In many cases, one wants a simple workflow. The friendly functions in the last part are a simple api on top of the Routine/CodeGen stuff. They are easier to use, but are less powerful.

— Milestones —

- First working version with scalar input arguments, generating C code, tests
- Friendly functions that are easier to use than the rigorous Routine/CodeGen workflow.
- Integer and Real numbers as input and output
- Output arguments
- InputOutput arguments
- Sort input/output arguments properly
- Contiguous array arguments (numpy matrices)
- Also generate .pyf code for f2py (in autowrap module)
- Isolate constants and evaluate them beforehand in double precision
- Fortran 90
- Octave/Matlab
- Common Subexpression Elimination
- User defined comments in the generated code

万兴PDF

- Optional extra include lines for libraries/objects that can eval special functions
- Test other C compilers and libraries: gcc, tcc, libtcc, gcc+gsl, …
- Contiguous array arguments (SymPy matrices)
- Non-contiguous array arguments (SymPy matrices)
- ccode must raise an error when it encounters something that cannot be translated into c. ccode(integrate(sin(x)/x, x)) does not make sense.
- Complex numbers as input and output
- A default complex datatype
- Include extra information in the header: date, user, hostname, sha1 hash, …
- Fortran 77
- C++
- Python
- Julia
- Rust
- …

class `sympy.utilities.codegen.`**`Argument`**(*name*, *datatype=None*, *dimensions=None*, *precision=None*)

An abstract Argument data structure: a name and a data type.

This structure is refined in the descendants below.

万兴PDF

class `sympy.utilities.codegen.`**`CCodeGen`**(*project='project'*, *printer=None*, *preprocessor_statements=None*, *cse=False*)

Generator for C code.

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.c and <prefix>.h respectively.

`dump_c`(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file. [default : True]

万兴PDF

empty : bool, optional

When True, empty lines are included to structure the source files. [default : True]

dump_h(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Writes the C header file.

This file contains all the function declarations.

Parameters

routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to construct the include guards. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file. [default : True]

empty : bool, optional

When True, empty lines are included to structure the source files. [default : True]

万兴PDF

get_prototype(*routine*)

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an CodeGenError is raised.

See: https://en.wikipedia.org/wiki/Function_prototype

class sympy.utilities.codegen.**CodeGen**(*project='project'*, *cse=False*)

Abstract class for the code generators.

dump_code(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

万兴PDF

When True, a header comment is included on top of each source file. [default : True]

empty : bool, optional

When True, empty lines are included to structure the source files. [default : True]

routine(*name*, *expr*, *argument_sequence=None*, *global_vars=None*)

Creates an Routine object that is appropriate for this language.

This implementation is appropriate for at least C/Fortran. Subclasses can override this if necessary.

Here, we assume at most one return value (the l-value) which must be scalar. Additional outputs are OutputArguments (e.g., pointers on right-hand-side or pass-by-reference). Matrices are always returned via OutputArguments. If `argument_sequence` is None, arguments will be ordered alphabetically, but with all InputArguments first, and then OutputArgument and InOutArguments.

write(*routines*, *prefix*, *to_files=False*, *header=True*, *empty=True*)

Writes all the source code files for the given routines.

The generated source is returned as a list of (filename, contents) tuples, or is written to files (see below). Each filename consists of the given prefix, appended with an appropriate extension.

Parameters

routines : list

A list of Routine instances to be written

prefix : string

The prefix for the output files

to_files : bool, optional

When True, the output is written to files. Otherwise, a list of (filename, contents) tuples is returned. [default: False]

header : bool, optional

When True, a header comment is included on top of each source file. [default: True]

empty : bool, optional

When True, empty lines are included to structure the source files. [default: True]

万兴PDF

class sympy.utilities.codegen.**DataType**(*cname*, *fname*, *pyname*, *jlname*, *octname*, *rsname*)

Holds strings for a certain datatype in different languages.

class sympy.utilities.codegen.**FCodeGen**(*project='project'*, *printer=None*)

Generator for Fortran 95 code

The .write() method inherited from CodeGen will output a code file and an interface file, <prefix>.f90 and <prefix>.h respectively.

万兴PDF

dump_f95(*routines, f, prefix, header=True, empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file. [default : True]

empty : bool, optional

When True, empty lines are included to structure the source files. [default : True]

dump_h(*routines, f, prefix, header=True, empty=True*)

Writes the interface to a header file.

This file contains all the function declarations.

Parameters

routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix.

header : bool, optional

When True, a header comment is included on top of each source file. [default : True]

empty : bool, optional

When True, empty lines are included to structure the source files. [default : True]

万兴PDF

get_interface(*routine*)

Returns a string for the function interface.

The routine should have a single result object, which can be None. If the routine has multiple result objects, a CodeGenError is raised.

See: https://en.wikipedia.org/wiki/Function_prototype

万兴PDF

class sympy.utilities.codegen.**JuliaCodeGen**(*project='project'*, *printer=None*)

Generator for Julia code.

The .write() method inherited from CodeGen will output a code file <prefix>.jl.

dump_jl(*routines*, *f*, *prefix*, *header=True*, *empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file. [default : True]

empty : bool, optional

When True, empty lines are included to structure the source files. [default : True]

万兴PDF

routine(*name*, *expr*, *argument_sequence*, *global_vars*)

Specialized Routine creation for Julia.

class sympy.utilities.codegen.**OctaveCodeGen**(*project='project'*, *printer=None*)

Generator for Octave code.

The .write() method inherited from CodeGen will output a code file <prefix>.m.

Octave .m files usually contain one function. That function name should match the filename (`prefix`). If you pass multiple `name_expr` pairs, the latter ones are presumed to be private functions accessed by the primary function.

You should only pass inputs to `argument_sequence`: outputs are ordered according to their order in `name_expr`.

dump_m(*routines*, *f*, *prefix*, *header=True*, *empty=True*, *inline=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

routines : list

A list of Routine instances.

f : file-like

Where to write the file.

万兴PDF

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file. [default : True]

empty : bool, optional

When True, empty lines are included to structure the source files. [default : True]

routine(*name*, *expr*, *argument_sequence*, *global_vars*)

Specialized Routine creation for Octave.

class sympy.utilities.codegen.**OutputArgument**(*name*, *result_var*, *expr*, *datatype=None*, *dimensions=None*, *precision=None*)

OutputArgument are always initialized in the routine.

class sympy.utilities.codegen.**Result**(*expr*, *name=None*, *result_var=None*, *datatype=None*, *dimensions=None*, *precision=None*)

An expression for a return value.

The name result is used to avoid conflicts with the reserved word "return" in the Python language. It is also shorter than ReturnValue.

These may or may not need a name in the destination (e.g., "return(x*y)" might return a value without ever naming it).

万兴PDF

class sympy.utilities.codegen.**Routine**(*name*, *arguments*, *results*, *local_vars*, *global_vars*)

Generic description of evaluation routine for set of expressions.

A CodeGen class can translate instances of this class into code in a particular language. The routine specification covers all the features present in these languages. The CodeGen part must raise an exception when certain features are not present in the target language. For example, multiple return values are possible in Python, but not in C or Fortran. Another example: Fortran and Python support complex numbers, while C does not.

property result_variables

Returns a list of OutputArgument, InOutArgument and Result.

If return values are present, they are at the end ot the list.

property variables

Returns a set of all variables possibly used in the routine.

For routines with unnamed return values, the dummies that may or may not be used will be included in the set.

class sympy.utilities.codegen.**RustCodeGen**(*project='project'*, *printer=None*)

Generator for Rust code.

The .write() method inherited from CodeGen will output a code file <prefix>.rs

万兴PDF

dump_rs(*routines, f, prefix, header=True, empty=True*)

Write the code by calling language specific methods.

The generated file contains all the definitions of the routines in low-level code and refers to the header file if appropriate.

Parameters

routines : list

A list of Routine instances.

f : file-like

Where to write the file.

prefix : string

The filename prefix, used to refer to the proper header file. Only the basename of the prefix is used.

header : bool, optional

When True, a header comment is included on top of each source file. [default : True]

empty : bool, optional

When True, empty lines are included to structure the source files. [default : True]

万兴PDF

get_prototype(*routine*)

Returns a string for the function prototype of the routine.

If the routine has multiple result objects, an CodeGenError is raised.

See: https://en.wikipedia.org/wiki/Function_prototype

routine(*name, expr, argument_sequence, global_vars*)

Specialized Routine creation for Rust.

sympy.utilities.codegen.**codegen**(*name_expr, language=None, prefix=None, project='project', to_files=False, header=True, empty=True, argument_sequence=None, global_vars=None, standard=None, code_gen=None, printer=None*)

Generate source code for expressions in a given language.

Parameters

name_expr : tuple, or list of tuples

A single (name, expression) tuple or a list of (name, expression) tuples. Each tuple corresponds to a routine. If the expression is an equality (an instance of class Equality) the left hand side is considered an output argument. If expression is an iterable, then the routine will have multiple outputs.

language : string,

A string that indicates the source code language. This is case insensitive. Currently, 'C', 'F95' and 'Octave' are supported. 'Octave' generates code compatible with both Octave and Matlab.

prefix : string, optional

万兴PDF

A prefix for the names of the files that contain the source code. Language-dependent suffixes will be appended. If omitted, the name of the first name_expr tuple is used.

project : string, optional

A project name, used for making unique preprocessor instructions. [default: "project"]

to_files : bool, optional

When True, the code will be written to one or more files with the given prefix, otherwise strings with the names and contents of these files are returned. [default: False]

header : bool, optional

When True, a header is written on top of each source file. [default: True]

empty : bool, optional

When True, empty lines are used to structure the code. [default: True]

argument_sequence : iterable, optional

Sequence of arguments for the routine in a preferred order. A CodeGenError is raised if required arguments are missing. Redundant arguments are used without warning. If omitted, arguments will be ordered alphabetically, but with all input arguments first, and then output or in-out arguments.

global_vars : iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

standard : string

code_gen : CodeGen instance

An instance of a CodeGen subclass. Overrides `language`.

Examples

```
>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...     ("f", x+y*z), "C89", "test", header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double f(double x, double y, double z) {
   double f_result;
   f_result = x + y*z;
   return f_result;
```

(continues on next page)

万兴PDF

(continued from previous page)

```
}

>>> print(h_name)
test.h
>>> print(c_header)
#ifndef PROJECT__TEST__H
#define PROJECT__TEST__H
double f(double x, double y, double z);
#endif
```

Another example using Equality objects to give named outputs. Here the filename (prefix) is taken from the first (name, expr) pair.

```
>>> from sympy.abc import f, g
>>> from sympy import Eq
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...      [("myfcn", x + y), ("fcn2", [Eq(f, 2*x), Eq(g, y)])],
...      "C99", header=False, empty=False)
>>> print(c_name)
myfcn.c
>>> print(c_code)
#include "myfcn.h"
#include <math.h>
double myfcn(double x, double y) {
   double myfcn_result;
   myfcn_result = x + y;
   return myfcn_result;
}
void fcn2(double x, double y, double *f, double *g) {
   (*f) = 2*x;
   (*g) = y;
}
```

万兴PDF

If the generated function(s) will be part of a larger project where various global variables have been defined, the 'global_vars' option can be used to remove the specified variables from the function signature

```
>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(f_name, f_code), header] = codegen(
...     ("f", x+y*z), "F95", header=False, empty=False,
...     argument_sequence=(x, y), global_vars=(z,))
>>> print(f_code)
REAL*8 function f(x, y)
implicit none
REAL*8, intent(in) :: x
REAL*8, intent(in) :: y
f = x + y*z
end function
```

sympy.utilities.codegen.**get_default_datatype**(*expr*, *complex_allowed=None*)

Derives an appropriate datatype based on the expression.

万兴PDF

`sympy.utilities.codegen.`**`make_routine`**(*name, expr, argument_sequence=None, global_vars=None, language='F95'*)

A factory that makes an appropriate Routine from an expression.

Parameters

name : string

The name of this routine in the generated code.

expr : expression or list/tuple of expressions

A SymPy expression that the Routine instance will represent. If given a list or tuple of expressions, the routine will be considered to have multiple return values and/or output arguments.

argument_sequence : list or tuple, optional

List arguments for the routine in a preferred order. If omitted, the results are language dependent, for example, alphabetical order or in the same order as the given expressions.

global_vars : iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

language : string, optional

Specify a target language. The Routine itself should be language-agnostic but the precise way one is created, error checking, etc depend on the language. [default: "F95"].

万兴PDF

Notes

A decision about whether to use output arguments or return values is made depending on both the language and the particular mathematical expressions. For an expression of type Equality, the left hand side is typically made into an OutputArgument (or perhaps an InOutArgument if appropriate). Otherwise, typically, the calculated expression is made a return values of the routine.

Examples

```
>>> from sympy.utilities.codegen import make_routine
>>> from sympy.abc import x, y, f, g
>>> from sympy import Eq
>>> r = make_routine('test', [Eq(f, 2*x), Eq(g, x + y)])
>>> [arg.result_var for arg in r.results]
[]
>>> [arg.name for arg in r.arguments]
[x, y, f, g]
>>> [arg.name for arg in r.result_variables]
[f, g]
>>> r.local_vars
set()
```

万兴PDF

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output.

```
>>> from sympy import Matrix
>>> r = make_routine('fcn', [x*y, Eq(f, 1), Eq(g, x + g), Matrix([[x,␣
↪2]])])
>>> [arg.result_var for arg in r.results]
[result_5397460570204848505]
>>> [arg.expr for arg in r.results]
[x*y]
>>> [arg.name for arg in r.arguments]
[x, y, f, g, out_8598435338387848786]
```

We can examine the various arguments more closely:

```
>>> from sympy.utilities.codegen import (InputArgument, OutputArgument,
...                                      InOutArgument)
>>> [a.name for a in r.arguments if isinstance(a, InputArgument)]
[x, y]
```

```
>>> [a.name for a in r.arguments if isinstance(a, OutputArgument)]
[f, out_8598435338387848786]
>>> [a.expr for a in r.arguments if isinstance(a, OutputArgument)]
[1, Matrix([[x, 2]])]
```

```
>>> [a.name for a in r.arguments if isinstance(a, InOutArgument)]
[g]
>>> [a.expr for a in r.arguments if isinstance(a, InOutArgument)]
[g + x]
```

万兴PDF

Decorator

Useful utility decorators.

`@sympy.utilities.decorator.`**`deprecated`**`(`*message, *, deprecated_since_version, active_deprecations_target, stacklevel=3*`)`

Mark a function as deprecated.

This decorator should be used if an entire function or class is deprecated. If only a certain functionality is deprecated, you should use *`warns_deprecated_sympy()`* (page 2029) directly. This decorator is just a convenience. There is no functional difference between using this decorator and calling `warns_deprecated_sympy()` at the top of the function.

The decorator takes the same arguments as *`warns_deprecated_sympy()`* (page 2029). See its documentation for details on what the keywords to this decorator do.

See the *Deprecation Policy* (page 3017) document for details on when and how things should be deprecated in SymPy.

万兴PDF

Examples

```
>>> from sympy.utilities.decorator import deprecated
>>> from sympy import simplify
>>> @deprecated("""    ... The simplify_this(expr) function is␣
↪deprecated. Use simplify(expr)
... instead.""", deprecated_since_version="1.1",
... active_deprecations_target='simplify-this-deprecation')
... def simplify_this(expr):
...     """
...     Simplify ``expr``.
...
...     .. deprecated:: 1.1
...
...        The ``simplify_this`` function is deprecated. Use␣
↪:func:`simplify`
...        instead. See its documentation for more information. See
...        :ref:`simplify-this-deprecation` for details.
...
...     """
...     return simplify(expr)
>>> from sympy.abc import x
>>> simplify_this(x*(x + 1) - x**2)
<stdin>:1: SymPyDeprecationWarning:

The simplify_this(expr) function is deprecated. Use simplify(expr)
instead.

See https://docs.sympy.org/latest/explanation/active-deprecations.html
↪#simplify-this-deprecation
for details.

This has been deprecated since SymPy version 1.1. It
will be removed in a future version of SymPy.

  simplify_this(x)
x
```

万兴PDF

See also:

sympy.utilities.exceptions.SymPyDeprecationWarning (page 2066), *sympy.utilities.exceptions.sympy_deprecation_warning* (page 2067), *sympy.utilities.exceptions.ignore_warnings* (page 2067), *sympy.testing.pytest.warns_deprecated_sympy* (page 2029)

sympy.utilities.decorator.**conserve_mpmath_dps**(*func*)

After the function finishes, resets the value of mpmath.mp.dps to the value it had before the function was run.

sympy.utilities.decorator.**doctest_depends_on**(*exe=None*, *modules=None*, *disable_viewers=None*, *python_version=None*)

Adds metadata about the dependencies which need to be met for doctesting the docstrings of the decorated objects.

万兴PDF

exe should be a list of executables

modules should be a list of modules

disable_viewers should be a list of viewers for preview() to disable

python_version should be the minimum Python version required, as a tuple (like (3, 0))

`sympy.utilities.decorator.`**`memoize_property`**`(`*propfunc*`)`

Property decorator that caches the value of potentially expensive $propfunc$ after the first evaluation. The cached value is stored in the corresponding property name with an attached underscore.

`class` `sympy.utilities.decorator.`**`no_attrs_in_subclass`**`(`*cls*, *f*`)`

Don't 'inherit' certain attributes from a base class

```
>>> from sympy.utilities.decorator import no_attrs_in_subclass
```

```
>>> class A(object):
...     x = 'test'
```

```
>>> A.x = no_attrs_in_subclass(A, A.x)
```

```
>>> class B(A):
...     pass
```

```
>>> hasattr(A, 'x')
True
>>> hasattr(B, 'x')
False
```

万兴PDF

`sympy.utilities.decorator.`**`public`**`(`*obj*`)`

Append `obj`'s name to global `__all__` variable (call site).

By using this decorator on functions or classes you achieve the same goal as by filling `__all__` variables manually, you just do not have to repeat yourself (object's name). You also know if object is public at definition site, not at some random location (where `__all__` was set).

Note that in multiple decorator setup (in almost all cases) `@public` decorator must be applied before any other decorators, because it relies on the pointer to object's global namespace. If you apply other decorators first, `@public` may end up modifying the wrong namespace.

Examples

```
>>> from sympy.utilities.decorator import public
```

```
>>> __all__ # noqa: F821
Traceback (most recent call last):
...
NameError: name '__all__' is not defined
```

```
>>> @public
... def some_function():
...     pass
```

```
>>> __all__ # noqa: F821
['some_function']
```

`sympy.utilities.decorator.`**`threaded`**`(`*func*`)`

Apply `func` to sub-elements of an object, including *Add* (page 1013).

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of *threaded()* (page 2060) decorator allows threading over elements of *Add* (page 1013) class. If this behavior is not desirable use *xthreaded()* (page 2060) decorator.

Functions using this decorator must have the following signature:

```
@threaded
def function(expr, *args, **kwargs):
```

`sympy.utilities.decorator.`**`threaded_factory`**`(`*func, use_add*`)`

A factory for `threaded` decorators.

`sympy.utilities.decorator.`**`xthreaded`**`(`*func*`)`

Apply `func` to sub-elements of an object, excluding *Add* (page 1013).

This decorator is intended to make it uniformly possible to apply a function to all elements of composite objects, e.g. matrices, lists, tuples and other iterable containers, or just expressions.

This version of *threaded()* (page 2060) decorator disallows threading over elements of *Add* (page 1013) class. If this behavior is not desirable use *threaded()* (page 2060) decorator.

Functions using this decorator must have the following signature:

```
@xthreaded
def function(expr, *args, **kwargs):
```

Enumerative

This module includes functions and classes for enumerating and counting multiset partitions.

`sympy.utilities.enumerative.`**`multiset_partitions_taocp`**`(`*multiplicities*`)`

Enumerates partitions of a multiset.

Parameters

multiplicities

list of integer multiplicities of the components of the multiset.

Yields

state

万兴PDF

Internal data structure which encodes a particular partition. This output is then usually processed by a visitor function which combines the information from this data structure with the components themselves to produce an actual partition.

Unless they wish to create their own visitor function, users will have little need to look inside this data structure. But, for reference, it is a 3-element list with components:

f
is a frame array, which is used to divide pstack into parts.

lpart
points to the base of the topmost part.

pstack
is an array of PartComponent objects.

The `state` output offers a peek into the internal data structures of the enumeration function. The client should treat this as read-only; any modification of the data structure will cause unpredictable (and almost certainly incorrect) results. Also, the components of `state` are modified in place at each iteration. Hence, the visitor must be called at each loop iteration. Accumulating the `state` instances and processing them later will not work.

万兴PDF

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> # variables components and multiplicities represent the multiset 'abb
↪'
>>> components = 'ab'
>>> multiplicities = [1, 2]
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(list_visitor(state, components) for state in states)
[[['a', 'b', 'b']],
[['a', 'b'], ['b']],
[['a'], ['b', 'b']],
[['a'], ['b'], ['b']]]
```

See also:

sympy.utilities.iterables.multiset_partitions **(page 2084)**
Takes a multiset as input and directly yields multiset partitions. It dispatches to a number of functions, including this one, for implementation. Most users will find it more convenient to use than multiset_partitions_taocp.

sympy.utilities.enumerative.**factoring_visitor**(*state*, *primes*)

Use with multiset_partitions_taocp to enumerate the ways a number can be expressed as a product of factors. For this usage, the exponents of the prime factors of a number are arguments to the partition enumerator, while the corresponding prime factors are input here.

万兴PDF

Examples

To enumerate the factorings of a number we can think of the elements of the partition as being the prime factors and the multiplicities as being their exponents.

```
>>> from sympy.utilities.enumerative import factoring_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> from sympy import factorint
>>> primes, multiplicities = zip(*factorint(24).items())
>>> primes
(2, 3)
>>> multiplicities
(3, 1)
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(factoring_visitor(state, primes) for state in states)
[[24], [8, 3], [12, 2], [4, 6], [4, 2, 3], [6, 2, 2], [2, 2, 2, 3]]
```

sympy.utilities.enumerative.**list_visitor**(*state, components*)

Return a list of lists to represent the partition.

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import multiset_partitions_taocp
>>> states = multiset_partitions_taocp([1, 2, 1])
>>> s = next(states)
>>> list_visitor(s, 'abc')  # for multiset 'a b b c'
[['a', 'b', 'b', 'c']]
>>> s = next(states)
>>> list_visitor(s, [1, 2, 3])  # for multiset '1 2 2 3
[[1, 2, 2], [3]]
```

万兴PDF

The approach of the function multiset_partitions_taocp is extended and generalized by the class MultisetPartitionTraverser.

class sympy.utilities.enumerative.**MultisetPartitionTraverser**

Has methods to enumerate and count the partitions of a multiset.

This implements a refactored and extended version of Knuth's algorithm 7.1.2.5M [AOCP]."

The enumeration methods of this class are generators and return data structures which can be interpreted by the same visitor functions used for the output of multiset_partitions_taocp.

万兴PDF

Examples

```
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([4,4,4,2])
127750
>>> m.count_partitions([3,3,3])
686
```

See also:

multiset_partitions_taocp (page 2060), *sympy.utilities.iterables.multiset_partitions* (page 2084)

References

[AOCP], [Factorisatio], [Yorgey]

count_partitions(*multiplicities*)

Returns the number of partitions of a multiset whose components have the multiplicities given in multiplicities.

For larger counts, this method is much faster than calling one of the enumerators and counting the result. Uses dynamic programming to cut down on the number of nodes actually explored. The dictionary used in order to accelerate the counting process is stored in the MultisetPartitionTraverser object and persists across calls. If the user does not expect to call count_partitions for any additional multisets, the object should be cleared to save memory. On the other hand, the cache built up from one count run can significantly speed up subsequent calls to count_partitions, so it may be advantageous not to clear the object.

万兴PDF

Examples

```
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([9,8,2])
288716
>>> m.count_partitions([2,2])
9
>>> del m
```

Notes

If one looks at the workings of Knuth's algorithm M [AOCP], it can be viewed as a traversal of a binary tree of parts. A part has (up to) two children, the left child resulting from the spread operation, and the right child from the decrement operation. The ordinary enumeration of multiset partitions is an in-order traversal of this tree, and with the partitions corresponding to paths from the root to the leaves. The mapping from paths to partitions is a little complicated, since the partition would contain only those parts which are leaves or the parents of a spread link, not those which are parents of a decrement link.

万兴PDF

For counting purposes, it is sufficient to count leaves, and this can be done with a recursive in-order traversal. The number of leaves of a subtree rooted at a particular part is a function only of that part itself, so memoizing has the potential to speed up the counting dramatically.

This method follows a computational approach which is similar to the hypothetical memoized recursive function, but with two differences:

1) This method is iterative, borrowing its structure from the other enumerations and maintaining an explicit stack of parts which are in the process of being counted. (There may be multisets which can be counted reasonably quickly by this implementation, but which would overflow the default Python recursion limit with a recursive implementation.)

2) Instead of using the part data structure directly, a more compact key is constructed. This saves space, but more importantly coalesces some parts which would remain separate with physical keys.

Unlike the enumeration functions, there is currently no _range version of count_partitions. If someone wants to stretch their brain, it should be possible to construct one by memoizing with a histogram of counts rather than a single count, and combining the histograms.

enum_all(*multiplicities*)

Enumerate the partitions of a multiset.

Examples

万兴PDF

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_all([2,2])
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
[['a', 'a', 'b'], ['b']],
[['a', 'a'], ['b', 'b']],
[['a', 'a'], ['b'], ['b']],
[['a', 'b', 'b'], ['a']],
[['a', 'b'], ['a', 'b']],
[['a', 'b'], ['a'], ['b']],
[['a'], ['a'], ['b', 'b']],
[['a'], ['a'], ['b'], ['b']]]
```

See also:

multiset_partitions_taocp **(page 2060)**
which provides the same result as this method, but is about twice as fast. Hence, enum_all is primarily useful for testing. Also see the function for a discussion of states and visitors.

enum_large(*multiplicities, lb*)

Enumerate the partitions of a multiset with lb < num(parts)

Equivalent to enum_range(multiplicities, lb, sum(multiplicities))

万兴PDF

Parameters

multiplicities

list of multiplicities of the components of the multiset.

lb

Number of parts in the partition must be greater than this lower bound.

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_large([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a'], ['b'], ['b']],
[['a', 'b'], ['a'], ['b']],
[['a'], ['a'], ['b', 'b']],
[['a'], ['a'], ['b'], ['b']]]
```

See also:

enum_all (page 2064), *enum_small* (page 2065), *enum_range* (page 2065)

enum_range(*multiplicities*, *lb*, *ub*)

Enumerate the partitions of a multiset with `lb < num(parts) <= ub`.

In particular, if partitions with exactly `k` parts are desired, call with `(multiplicities, k - 1, k)`. This method generalizes enum_all, enum_small, and enum_large.

万兴PDF

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_range([2,2], 1, 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b'], ['b']],
[['a', 'a'], ['b', 'b']],
[['a', 'b', 'b'], ['a']],
[['a', 'b'], ['a', 'b']]]
```

enum_small(*multiplicities*, *ub*)

Enumerate multiset partitions with no more than ub parts.

Equivalent to enum_range(multiplicities, 0, ub)

Parameters

multiplicities

list of multiplicities of the components of the multiset.

ub

万兴PDF

Maximum number of parts

Examples

```
>>> from sympy.utilities.enumerative import list_visitor
>>> from sympy.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_small([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
[['a', 'a', 'b'], ['b']],
[['a', 'a'], ['b', 'b']],
[['a', 'b', 'b'], ['a']],
[['a', 'b'], ['a', 'b']]]
```

The implementation is based, in part, on the answer given to exercise 69, in Knuth [AOCP].

See also:

enum_all (page 2064), *enum_large* (page 2064), *enum_range* (page 2065)

Exceptions and Warnings

General SymPy exceptions and warnings.

exception sympy.utilities.exceptions.**SymPyDeprecationWarning**(*message*, ***, *deprecated_since_version*, *active_deprecations_target*)

A warning for deprecated features of SymPy.

See the *Deprecation Policy* (page 3017) document for details on when and how things should be deprecated in SymPy.

Note that simply constructing this class will not cause a warning to be issued. To do that, you must call the :func`sympy_deprecation_warning` function. For this reason, it is not recommended to ever construct this class directly.

Explanation

The `SymPyDeprecationWarning` class is a subclass of `DeprecationWarning` that is used for all deprecations in SymPy. A special subclass is used so that we can automatically augment the warning message with additional metadata about the version the deprecation was introduced in and a link to the documentation. This also allows users to explicitly filter deprecation warnings from SymPy using `warnings` filters (see *Silencing SymPy Deprecation Warnings* (page 164)).

Additionally, `SymPyDeprecationWarning` is enabled to be shown by default, unlike normal `DeprecationWarning`s, which are only shown by default in interactive sessions. This ensures that deprecation warnings in SymPy will actually be seen by users.

See the documentation of *sympy_deprecation_warning()* (page 2067) for a description of the parameters to this function.

万兴PDF

万兴PDF

To mark a function as deprecated, you can use the *@deprecated* (page 2057) decorator.

See also:

sympy.utilities.exceptions.sympy_deprecation_warning (page 2067), *sympy.utilities.exceptions.ignore_warnings* (page 2067), *sympy.utilities.decorator.deprecated* (page 2057), *sympy.testing.pytest.warns_deprecated_sympy* (page 2029)

sympy.utilities.exceptions.**ignore_warnings**(*warningcls*)

Context manager to suppress warnings during tests.

Note: Do not use this with SymPyDeprecationWarning in the tests. warns_deprecated_sympy() should be used instead.

This function is useful for suppressing warnings during tests. The warns function should be used to assert that a warning is raised. The ignore_warnings function is useful in situation when the warning is not guaranteed to be raised (e.g. on importing a module) or if the warning comes from third-party code.

This function is also useful to prevent the same or similar warnings from being issue twice due to recursive calls.

When the warning is coming (reliably) from SymPy the warns function should be preferred to ignore_warnings.

```
>>> from sympy.utilities.exceptions import ignore_warnings
>>> import warnings
```

Here's a warning:

```
>>> with warnings.catch_warnings():  # reset warnings in doctest
...     warnings.simplefilter('error')
...     warnings.warn('deprecated', UserWarning)
Traceback (most recent call last):
  ...
UserWarning: deprecated
```

Let's suppress it with ignore_warnings:

```
>>> with warnings.catch_warnings():  # reset warnings in doctest
...     warnings.simplefilter('error')
...     with ignore_warnings(UserWarning):
...         warnings.warn('deprecated', UserWarning)
```

(No warning emitted)

See also:

sympy.utilities.exceptions.SymPyDeprecationWarning (page 2066), *sympy.utilities.exceptions.sympy_deprecation_warning* (page 2067), *sympy.utilities.decorator.deprecated* (page 2057), *sympy.testing.pytest.warns_deprecated_sympy* (page 2029)

sympy.utilities.exceptions.**sympy_deprecation_warning**(*message*, **, *deprecated_since_version*, *active_deprecations_target*, *stacklevel=3*)

万兴PDF

Warn that a feature is deprecated in SymPy.

See the *Deprecation Policy* (page 3017) document for details on when and how things should be deprecated in SymPy.

To mark an entire function or class as deprecated, you can use the *@deprecated* (page 2057) decorator.

Parameters

message: str

The deprecation message. This may span multiple lines and contain code examples. Messages should be wrapped to 80 characters. The message is automatically dedented and leading and trailing whitespace stripped. Messages may include dynamic content based on the user input, but avoid using `str(expression)` if an expression can be arbitrary, as it might be huge and make the warning message unreadable.

deprecated_since_version: str

The version of SymPy the feature has been deprecated since. For new deprecations, this should be the version in sympy/release.py without the `.dev`. If the next SymPy version ends up being different from this, the release manager will need to update any `SymPyDeprecationWarning`s using the incorrect version. This argument is required and must be passed as a keyword argument. (example: `deprecated_since_version="1.10"`).

active_deprecations_target: str

The Sphinx target corresponding to the section for the deprecation in the *List of active deprecations* (page 163) document (see `doc/src/explanation/active-deprecations.md`). This is used to automatically generate a URL to the page in the warning message. This argument is required and must be passed as a keyword argument. (example: `active_deprecations_target="deprecated-feature-abc"`)

stacklevel: int (default: 3)

The `stacklevel` parameter that is passed to `warnings.warn`. If you create a wrapper that calls this function, this should be increased so that the warning message shows the user line of code that produced the warning. Note that in some cases there will be multiple possible different user code paths that could result in the warning. In that case, just choose the smallest common stacklevel.

Examples

```
>>> from sympy.utilities.exceptions import sympy_deprecation_warning
>>> def is_this_zero(x, y=0):
...     """
...     Determine if x = 0.
...
...     Parameters
...     ==========
```

(continues on next page)

万兴PDF

(continued from previous page)

```
...
...     x : Expr
...       The expression to check.
...
...     y : Expr, optional
...       If provided, check if x = y.
...
...       .. deprecated:: 1.1
...
...          The ``y`` argument to ``is_this_zero`` is deprecated. Use
...          ``is_this_zero(x - y)`` instead.
...
...     """
...     from sympy import simplify
...
...     if y != 0:
...         sympy_deprecation_warning("""
...     The y argument to is_zero() is deprecated. Use is_zero(x - y)␣
↪instead.""",
...             deprecated_since_version="1.1",
...             active_deprecations_target='is-this-zero-y-deprecation')
...     return simplify(x - y) == 0
>>> is_this_zero(0)
True
>>> is_this_zero(1, 1)
<stdin>:1: SymPyDeprecationWarning:

The y argument to is_zero() is deprecated. Use is_zero(x - y) instead.

See https://docs.sympy.org/latest/explanation/active-deprecations.html
↪#is-this-zero-y-deprecation
for details.

This has been deprecated since SymPy version 1.1. It
will be removed in a future version of SymPy.

  is_this_zero(1, 1)
True
```

万兴PDF

See also:

sympy.utilities.exceptions.SymPyDeprecationWarning (page 2066), *sympy.utilities.exceptions.ignore_warnings* (page 2067), *sympy.utilities.decorator.deprecated* (page 2057), *sympy.testing.pytest.warns_deprecated_sympy* (page 2029)

万兴PDF

Iterables

variations

variations(seq, n) Returns all the variations of the list of size n.

Has an optional third argument. Must be a boolean value and makes the method return the variations with repetition if set to True, or the variations without repetition if set to False.

Examples::

```
>>> from sympy.utilities.iterables import variations
>>> list(variations([1,2,3], 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> list(variations([1,2,3], 2, True))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

partitions

Although the combinatorics module contains Partition and IntegerPartition classes for investigation and manipulation of partitions, there are a few functions to generate partitions that can be used as low-level tools for routines: `partitions` and `multiset_partitions`. The former gives integer partitions, and the latter gives enumerated partitions of elements. There is also a routine `kbins` that will give a variety of permutations of partions.

partitions:

```
>>> from sympy.utilities.iterables import partitions
>>> [p.copy() for s, p in partitions(7, m=2, size=True) if s == 2]
[{1: 1, 6: 1}, {2: 1, 5: 1}, {3: 1, 4: 1}]
```

multiset_partitions:

```
>>> from sympy.utilities.iterables import multiset_partitions
>>> [p for p in multiset_partitions(3, 2)]
[[[0, 1], [2]], [[0, 2], [1]], [[0], [1, 2]]]
>>> [p for p in multiset_partitions([1, 1, 1, 2], 2)]
[[[1, 1, 1], [2]], [[1, 1, 2], [1]], [[1, 1], [1, 2]]]
```

kbins:

```
>>> from sympy.utilities.iterables import kbins
>>> def show(k):
...     rv = []
...     for p in k:
...         rv.append(','.join([''.join(j) for j in p]))
...     return sorted(rv)
...
>>> show(kbins("ABCD", 2))
['A,BCD', 'AB,CD', 'ABC,D']
>>> show(kbins("ABC", 2))
['A,BC', 'AB,C']
>>> show(kbins("ABC", 2, ordered=0))  # same as multiset_partitions
```

(continues on next page)

万兴PDF

(continued from previous page)

```
['A,BC', 'AB,C', 'AC,B']
>>> show(kbins("ABC", 2, ordered=1))
['A,BC', 'A,CB',
 'B,AC', 'B,CA',
 'C,AB', 'C,BA']
>>> show(kbins("ABC", 2, ordered=10))
['A,BC', 'AB,C', 'AC,B',
 'B,AC', 'BC,A',
 'C,AB']
>>> show(kbins("ABC", 2, ordered=11))
['A,BC', 'A,CB', 'AB,C', 'AC,B',
 'B,AC', 'B,CA', 'BA,C', 'BC,A',
 'C,AB', 'C,BA', 'CA,B', 'CB,A']
```

Docstring

class sympy.utilities.iterables.**NotIterable**

Use this as mixin when creating a class which is not supposed to return true when iterable() is called on its instances because calling list() on the instance, for example, would result in an infinite loop.

sympy.utilities.iterables.**binary_partitions**(*n*)

Generates the binary partition of n.

A binary partition consists only of numbers that are powers of two. Each step reduces a 2^{k+1} to 2^k and 2^k. Thus 16 is converted to 8 and 8.

万兴PDF

Examples

```
>>> from sympy.utilities.iterables import binary_partitions
>>> for i in binary_partitions(5):
...     print(i)
...
[4, 1]
[2, 2, 1]
[2, 1, 1, 1]
[1, 1, 1, 1, 1]
```

References

[R958]

sympy.utilities.iterables.**bracelets**(*n*, *k*)

Wrapper to necklaces to return a free (unrestricted) necklace.

sympy.utilities.iterables.**capture**(*func*)

Return the printed output of func().

`func` should be a function without arguments that produces output with print statements.

万兴PDF

```
>>> from sympy.utilities.iterables import capture
>>> from sympy import pprint
>>> from sympy.abc import x
>>> def foo():
...     print('hello world!')
...
>>> 'hello' in capture(foo) # foo, not foo()
True
>>> capture(lambda: pprint(2/x))
'2\n-\nx\n'
```

sympy.utilities.iterables.**common_prefix**(**seqs*)

Return the subsequence that is a common start of sequences in seqs.

```
>>> from sympy.utilities.iterables import common_prefix
>>> common_prefix(list(range(3)))
[0, 1, 2]
>>> common_prefix(list(range(3)), list(range(4)))
[0, 1, 2]
>>> common_prefix([1, 2, 3], [1, 2, 5])
[1, 2]
>>> common_prefix([1, 2, 3], [1, 3, 5])
[1]
```

sympy.utilities.iterables.**common_suffix**(**seqs*)

Return the subsequence that is a common ending of sequences in seqs.

万兴PDF

```
>>> from sympy.utilities.iterables import common_suffix
>>> common_suffix(list(range(3)))
[0, 1, 2]
>>> common_suffix(list(range(3)), list(range(4)))
[]
>>> common_suffix([1, 2, 3], [9, 2, 3])
[2, 3]
>>> common_suffix([1, 2, 3], [9, 7, 3])
[3]
```

sympy.utilities.iterables.**connected_components**(*G*)

Connected components of an undirected graph or weakly connected components of a directed graph.

Parameters

graph : tuple[list, list[tuple[T, T]]

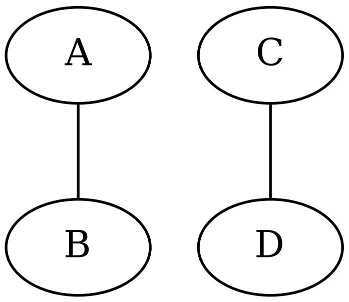
A tuple consisting of a list of vertices and a list of edges of a graph whose connected components are to be found.

万兴PDF

Examples

Given an undirected graph:

```
graph {
    A -- B
    C -- D
}
```



We can find the connected components using this function if we include each edge in both directions:

```
>>> from sympy.utilities.iterables import connected_components

>>> V = ['A', 'B', 'C', 'D']
>>> E = [('A', 'B'), ('B', 'A'), ('C', 'D'), ('D', 'C')]
>>> connected_components((V, E))
[['A', 'B'], ['C', 'D']]
```

The weakly connected components of a directed graph can found the same way.

Notes

The vertices of the graph must be hashable for the data structures used. If the vertices are unhashable replace them with integer indices.

This function uses Tarjan's algorithm to compute the connected components in $O(|V|+|E|)$ (linear) time.

See also:

sympy.utilities.iterables.strongly_connected_components (page 2094)

万兴PDF

万兴PDF

References

[R959], [R960]

sympy.utilities.iterables.**dict_merge**(**dicts*)

Merge dictionaries into a single dictionary.

sympy.utilities.iterables.**filter_symbols**(*iterator*, *exclude*)

Only yield elements from $iterator$ that do not occur in $exclude$.

Parameters

iterator : iterable

iterator to take elements from

exclude : iterable

elements to exclude

Returns

iterator : iterator

filtered iterator

sympy.utilities.iterables.**flatten**(*iterable*, *levels=None*, *cls=None*)

Recursively denest iterable containers.

```
>>> from sympy import flatten
```

```
>>> flatten([1, 2, 3])
[1, 2, 3]
>>> flatten([1, 2, [3]])
[1, 2, 3]
>>> flatten([1, [2, 3], [4, 5]])
[1, 2, 3, 4, 5]
>>> flatten([1.0, 2, (1, None)])
[1.0, 2, 1, None]
```

If you want to denest only a specified number of levels of nested containers, then set `levels` flag to the desired number of levels:

```
>>> ls = [[(-2, -1), (1, 2)], [(0, 0)]]
```

```
>>> flatten(ls, levels=1)
[(-2, -1), (1, 2), (0, 0)]
```

If cls argument is specified, it will only flatten instances of that class, for example:

```
>>> from sympy import Basic, S
>>> class MyOp(Basic):
...     pass
...
>>> flatten([MyOp(S(1), MyOp(S(2), S(3)))], cls=MyOp)
[1, 2, 3]
```

adapted from https://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks

万兴PDF

万兴PDF

sympy.utilities.iterables.**generate_bell**(*n*)

Return permutations of [0, 1, ..., n - 1] such that each permutation differs from the last by the exchange of a single pair of neighbors. The n! permutations are returned as an iterator. In order to obtain the next permutation from a random starting permutation, use the next_trotterjohnson method of the Permutation class (which generates the same sequence in a different manner).

Examples

```
>>> from itertools import permutations
>>> from sympy.utilities.iterables import generate_bell
>>> from sympy import zeros, Matrix
```

This is the sort of permutation used in the ringing of physical bells, and does not produce permutations in lexicographical order. Rather, the permutations differ from each other by exactly one inversion, and the position at which the swapping occurs varies periodically in a simple fashion. Consider the first few permutations of 4 elements generated by permutations and generate_bell:

```
>>> list(permutations(range(4)))[:5]
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 2, 1, 3), (0, 2, 3, 1), (0, 3, 1, 2)]
>>> list(generate_bell(4))[:5]
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 3, 1, 2), (3, 0, 1, 2), (3, 0, 2, 1)]
```

Notice how the 2nd and 3rd lexicographical permutations have 3 elements out of place whereas each "bell" permutation always has only two elements out of place relative to the previous permutation (and so the signature (+/-1) of a permutation is opposite of the signature of the previous permutation).

How the position of inversion varies across the elements can be seen by tracing out where the largest number appears in the permutations:

```
>>> m = zeros(4, 24)
>>> for i, p in enumerate(generate_bell(4)):
...     m[:, i] = Matrix([j - 3 for j in list(p)])  # make largest zero
>>> m.print_nonzero('X')
[XXX  XXXXXX  XXXXXX  XXX]
[XX XX XXXX XX XXXX XX XX]
[X XXXX XX XXXX XX XXXX X]
[ XXXXXX  XXXXXX  XXXXXX ]
```

See also:

sympy.combinatorics.permutations.Permutation.next_trotterjohnson (page 277)

万兴PDF

References

[R961], [R962], [R963], [R964], [R965]

sympy.utilities.iterables.**generate_derangements**(*s*)

Return unique derangements of the elements of iterable `s`.

Examples

```
>>> from sympy.utilities.iterables import generate_derangements
>>> list(generate_derangements([0, 1, 2]))
[[1, 2, 0], [2, 0, 1]]
>>> list(generate_derangements([0, 1, 2, 2]))
[[2, 2, 0, 1], [2, 2, 1, 0]]
>>> list(generate_derangements([0, 1, 1]))
[]
```

See also:

sympy.functions.combinatorial.factorials.subfactorial (page 435)

sympy.utilities.iterables.**generate_involutions**(*n*)

Generates involutions.

An involution is a permutation that when multiplied by itself equals the identity permutation. In this implementation the involutions are generated using Fixed Points.

Alternatively, an involution can be considered as a permutation that does not contain any cycles with a length that is greater than two.

万兴PDF

Examples

```
>>> from sympy.utilities.iterables import generate_involutions
>>> list(generate_involutions(3))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (2, 1, 0)]
>>> len(list(generate_involutions(4)))
10
```

References

[R966]

sympy.utilities.iterables.**generate_oriented_forest**(*n*)

This algorithm generates oriented forests.

An oriented graph is a directed graph having no symmetric pair of directed edges. A forest is an acyclic graph, i.e., it has no cycles. A forest can also be described as a disjoint union of trees, which are graphs in which any two vertices are connected by exactly one simple path.