Examples

```
>>> from sympy.utilities.iterables import generate_oriented_forest
>>> list(generate_oriented_forest(4))
[[0, 1, 2, 3], [0, 1, 2, 2], [0, 1, 2, 1], [0, 1, 2, 0],     [0, 1, 1,␣
↪1], [0, 1, 1, 0], [0, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 0]]
```

References

[R967], [R968]

sympy.utilities.iterables.**group**(*seq*, *multiple=True*)

Splits a sequence into a list of lists of equal, adjacent elements.

Examples

```
>>> from sympy import group
```

```
>>> group([1, 1, 1, 2, 2, 3])
[[1, 1, 1], [2, 2], [3]]
>>> group([1, 1, 1, 2, 2, 3], multiple=False)
[(1, 3), (2, 2), (3, 1)]
>>> group([1, 1, 3, 2, 2, 1], multiple=False)
[(1, 2), (3, 1), (2, 2), (1, 1)]
```

See also:

multiset (page 2083)

sympy.utilities.iterables.**has_dups**(*seq*)

Return True if there are any duplicate elements in seq.

Examples

```
>>> from sympy import has_dups, Dict, Set
>>> has_dups((1, 2, 1))
True
>>> has_dups(range(3))
False
>>> all(has_dups(c) is False for c in (set(), Set(), dict(), Dict()))
True
```

sympy.utilities.iterables.**has_variety**(*seq*)

Return True if there are any different elements in seq.

Examples

```
>>> from sympy import has_variety
```

```
>>> has_variety((1, 2, 1))
True
>>> has_variety((1, 1, 1))
False
```

sympy.utilities.iterables.**ibin**(*n*, *bits=None*, *str=False*)

Return a list of length `bits` corresponding to the binary value of `n` with small bits to the right (last). If bits is omitted, the length will be the number required to represent `n`. If the bits are desired in reversed order, use the `[::-1]` slice of the returned list.

If a sequence of all bits-length lists starting from `[0, 0,..., 0]` through `[1, 1, ..., 1]` are desired, pass a non-integer for bits, e.g. `'all'`.

If the bit *string* is desired pass `str=True`.

Examples

```
>>> from sympy.utilities.iterables import ibin
>>> ibin(2)
[1, 0]
>>> ibin(2, 4)
[0, 0, 1, 0]
```

If all lists corresponding to 0 to 2**n - 1, pass a non-integer for bits:

```
>>> bits = 2
>>> for i in ibin(2, 'all'):
...     print(i)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
```

If a bit string is desired of a given length, use str=True:

```
>>> n = 123
>>> bits = 10
>>> ibin(n, bits, str=True)
'0001111011'
>>> ibin(n, bits, str=True)[::-1]  # small bits left
'1101111000'
>>> list(ibin(3, 'all', str=True))
['000', '001', '010', '011', '100', '101', '110', '111']
```

sympy.utilities.iterables.**iproduct**(**iterables*)

Cartesian product of iterables.

Generator of the Cartesian product of iterables. This is analogous to itertools.product except that it works with infinite iterables and will yield any item from the infinite product eventually.

Examples

```
>>> from sympy.utilities.iterables import iproduct
>>> sorted(iproduct([1,2], [3,4]))
[(1, 3), (1, 4), (2, 3), (2, 4)]
```

With an infinite iterator:

```
>>> from sympy import S
>>> (3,) in iproduct(S.Integers)
True
>>> (3, 4) in iproduct(S.Integers, S.Integers)
True
```

See also:

itertools.product

sympy.utilities.iterables.**is_palindromic**(*s*, *i=0*, *j=None*)

Return True if the sequence is the same from left to right as it is from right to left in the whole sequence (default) or in the Python slice `s[i: j]`; else False.

Examples

```
>>> from sympy.utilities.iterables import is_palindromic
>>> is_palindromic([1, 0, 1])
True
>>> is_palindromic('abcbb')
False
>>> is_palindromic('abcbb', 1)
False
```

Normal Python slicing is performed in place so there is no need to create a slice of the sequence for testing:

```
>>> is_palindromic('abcbb', 1, -1)
True
>>> is_palindromic('abcbb', -4, -1)
True
```

See also:

sympy.ntheory.digits.is_palindromic **(page 1528)**
tests integers

sympy.utilities.iterables.**is_sequence**(*i*, *include=None*)

Return a boolean indicating whether `i` is a sequence in the SymPy sense. If anything that fails the test below should be included as being a sequence for your application, set 'include' to that object's type; multiple types should be passed as a tuple of types.

Note: although generators can generate a sequence, they often need special handling to make sure their elements are captured before the generator is exhausted, so these are not included by default in the definition of a sequence.

See also: iterable

Examples

```
>>> from sympy.utilities.iterables import is_sequence
>>> from types import GeneratorType
>>> is_sequence([])
True
>>> is_sequence(set())
False
>>> is_sequence('abc')
False
>>> is_sequence('abc', include=str)
True
>>> generator = (c for c in 'abc')
>>> is_sequence(generator)
False
>>> is_sequence(generator, include=(str, GeneratorType))
True
```

sympy.utilities.iterables.**iterable**(*i*, *exclude=(<class 'str'>, <class 'dict'>, <class 'sympy.utilities.iterables.NotIterable'>)*)

Return a boolean indicating whether `i` is SymPy iterable. True also indicates that the iterator is finite, e.g. you can call list(...) on the instance.

When SymPy is working with iterables, it is almost always assuming that the iterable is not a string or a mapping, so those are excluded by default. If you want a pure Python definition, make exclude=None. To exclude multiple items, pass them as a tuple.

You can also set the _iterable attribute to True or False on your class, which will override the checks here, including the exclude test.

As a rule of thumb, some SymPy functions use this to check if they should recursively map over an object. If an object is technically iterable in the Python sense but does not desire this behavior (e.g., because its iteration is not finite, or because iteration might induce an unwanted computation), it should disable it by setting the _iterable attribute to False.

See also: is_sequence

Examples

```
>>> from sympy.utilities.iterables import iterable
>>> from sympy import Tuple
>>> things = [[1], (1,), set([1]), Tuple(1), (j for j in [1, 2]), {1:2},
↪'1', 1]
>>> for i in things:
...     print('%s %s' % (iterable(i), type(i)))
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'sympy.core.containers.Tuple'>
```

(continues on next page)

(continued from previous page)

```
True <... 'generator'>
False <... 'dict'>
False <... 'str'>
False <... 'int'>
```

```
>>> iterable({}, exclude=None)
True
>>> iterable({}, exclude=str)
True
>>> iterable("no", exclude=str)
False
```

sympy.utilities.iterables.**kbins**(*l*, *k*, *ordered=None*)

Return sequence `l` partitioned into `k` bins.

Examples

The default is to give the items in the same order, but grouped into k partitions without any reordering:

```
>>> from sympy.utilities.iterables import kbins
>>> for p in kbins(list(range(5)), 2):
...     print(p)
...
[[0], [1, 2, 3, 4]]
[[0, 1], [2, 3, 4]]
[[0, 1, 2], [3, 4]]
[[0, 1, 2, 3], [4]]
```

The `ordered` flag is either None (to give the simple partition of the elements) or is a 2 digit integer indicating whether the order of the bins and the order of the items in the bins matters. Given:

```
A = [[0], [1, 2]]
B = [[1, 2], [0]]
C = [[2, 1], [0]]
D = [[0], [2, 1]]
```

the following values for `ordered` have the shown meanings:

```
00 means A == B == C == D
01 means A == B
10 means A == D
11 means A == A
```

```
>>> for ordered_flag in [None, 0, 1, 10, 11]:
...     print('ordered = %s' % ordered_flag)
...     for p in kbins(list(range(3)), 2, ordered=ordered_flag):
...         print('     %s' % p)
...
ordered = None
```

(continues on next page)

(continued from previous page)

```
     [[0], [1, 2]]
     [[0, 1], [2]]
ordered = 0
     [[0, 1], [2]]
     [[0, 2], [1]]
     [[0], [1, 2]]
ordered = 1
     [[0], [1, 2]]
     [[0], [2, 1]]
     [[1], [0, 2]]
     [[1], [2, 0]]
     [[2], [0, 1]]
     [[2], [1, 0]]
ordered = 10
     [[0, 1], [2]]
     [[2], [0, 1]]
     [[0, 2], [1]]
     [[1], [0, 2]]
     [[0], [1, 2]]
     [[1, 2], [0]]
ordered = 11
     [[0], [1, 2]]
     [[0, 1], [2]]
     [[0], [2, 1]]
     [[0, 2], [1]]
     [[1], [0, 2]]
     [[1, 0], [2]]
     [[1], [2, 0]]
     [[1, 2], [0]]
     [[2], [0, 1]]
     [[2, 0], [1]]
     [[2], [1, 0]]
     [[2, 1], [0]]
```

See also:

partitions (page 2089), *multiset_partitions* (page 2084)

sympy.utilities.iterables.**least_rotation**(*x*, *key=None*)

Returns the number of steps of left rotation required to obtain lexicographically minimal string/list/tuple, etc.

Examples

```
>>> from sympy.utilities.iterables import least_rotation, rotate_left
>>> a = [3, 1, 5, 1, 2]
>>> least_rotation(a)
3
>>> rotate_left(a, _)
[1, 2, 3, 1, 5]
```

References

[R969]

sympy.utilities.iterables.**minlex**(*seq*, *directed=True*, *key=None*)

Return the rotation of the sequence in which the lexically smallest elements appear first, e.g. $cba \to acb$.

The sequence returned is a tuple, unless the input sequence is a string in which case a string is returned.

If `directed` is False then the smaller of the sequence and the reversed sequence is returned, e.g. $cba \to abc$.

If `key` is not None then it is used to extract a comparison key from each element in iterable.

Examples

```
>>> from sympy.combinatorics.polyhedron import minlex
>>> minlex((1, 2, 0))
(0, 1, 2)
>>> minlex((1, 0, 2))
(0, 2, 1)
>>> minlex((1, 0, 2), directed=False)
(0, 1, 2)
```

```
>>> minlex('11010011000', directed=True)
'00011010011'
>>> minlex('11010011000', directed=False)
'00011001011'
```

```
>>> minlex(('bb', 'aaa', 'c', 'a'))
('a', 'bb', 'aaa', 'c')
>>> minlex(('bb', 'aaa', 'c', 'a'), key=len)
('c', 'a', 'bb', 'aaa')
```

sympy.utilities.iterables.**multiset**(*seq*)

Return the hashable sequence in multiset form with values being the multiplicity of the item in the sequence.

Examples

```
>>> from sympy.utilities.iterables import multiset
>>> multiset('mississippi')
{'i': 4, 'm': 1, 'p': 2, 's': 4}
```

See also:

group (page 2077)

sympy.utilities.iterables.**multiset_combinations**(*m*, *n*, *g=None*)

Return the unique combinations of size `n` from multiset `m`.

Examples

```
>>> from sympy.utilities.iterables import multiset_combinations
>>> from itertools import combinations
>>> [''.join(i) for i in  multiset_combinations('baby', 3)]
['abb', 'aby', 'bby']
```

```
>>> def count(f, s): return len(list(f(s, 3)))
```

The number of combinations depends on the number of letters; the number of unique combinations depends on how the letters are repeated.

```
>>> s1 = 'abracadabra'
>>> s2 = 'banana tree'
>>> count(combinations, s1), count(multiset_combinations, s1)
(165, 23)
>>> count(combinations, s2), count(multiset_combinations, s2)
(165, 54)
```

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

Generate derangements of the elements of s *in place*.

Examples

```
>>> from sympy.utilities.iterables import multiset_derangements, uniq
```

Because the derangements of multisets (not sets) are generated in place, copies of the return value must be made if a collection of derangements is desired or else all values will be the same:

```
>>> list(uniq([i for i in multiset_derangements('1233')]))
[[None, None, None, None]]
>>> [i.copy() for i in multiset_derangements('1233')]
[['3', '3', '1', '2'], ['3', '3', '2', '1']]
>>> [''.join(i) for i in multiset_derangements('1233')]
['3312', '3321']
```

sympy.utilities.iterables.**multiset_partitions**(*multiset*, *m=None*)

Return unique partitions of the given multiset (in list form). If `m` is None, all multisets will be returned, otherwise only partitions with `m` parts will be returned.

If `multiset` is an integer, a range [0, 1, ..., multiset - 1] will be supplied.

Examples

```
>>> from sympy.utilities.iterables import multiset_partitions
>>> list(multiset_partitions([1, 2, 3, 4], 2))
[[[1, 2, 3], [4]], [[1, 2, 4], [3]], [[1, 2], [3, 4]],
[[1, 3, 4], [2]], [[1, 3], [2, 4]], [[1, 4], [2, 3]],
[[1], [2, 3, 4]]]
>>> list(multiset_partitions([1, 2, 3, 4], 1))
[[[1, 2, 3, 4]]]
```

Only unique partitions are returned and these will be returned in a canonical order regardless of the order of the input:

```
>>> a = [1, 2, 2, 1]
>>> ans = list(multiset_partitions(a, 2))
>>> a.sort()
>>> list(multiset_partitions(a, 2)) == ans
True
>>> a = range(3, 1, -1)
>>> (list(multiset_partitions(a)) ==
...  list(multiset_partitions(sorted(a))))
True
```

If m is omitted then all partitions will be returned:

```
>>> list(multiset_partitions([1, 1, 2]))
[[[1, 1, 2]], [[1, 1], [2]], [[1, 2], [1]], [[1], [1], [2]]]
>>> list(multiset_partitions([1]*3))
[[[1, 1, 1]], [[1], [1, 1]], [[1], [1], [1]]]
```

Counting

The number of partitions of a set is given by the bell number:

```
>>> from sympy import bell
>>> len(list(multiset_partitions(5))) == bell(5) == 52
True
```

The number of partitions of length k from a set of size n is given by the Stirling Number of the 2nd kind:

```
>>> from sympy.functions.combinatorial.numbers import stirling
>>> stirling(5, 2) == len(list(multiset_partitions(5, 2))) == 15
True
```

These comments on counting apply to *sets*, not multisets.

Notes

When all the elements are the same in the multiset, the order of the returned partitions is determined by the `partitions` routine. If one is counting partitions then it is better to use the `nT` function.

See also:

partitions (page 2089), *sympy.combinatorics.partitions.Partition* (page 251), *sympy.combinatorics.partitions.IntegerPartition* (page 253), *sympy.functions.combinatorial.numbers.nT* (page 448)

sympy.utilities.iterables.**multiset_permutations**(*m*, *size=None*, *g=None*)

Return the unique permutations of multiset `m`.

Examples

```
>>> from sympy.utilities.iterables import multiset_permutations
>>> from sympy import factorial
>>> [''.join(i) for i in multiset_permutations('aab')]
['aab', 'aba', 'baa']
>>> factorial(len('banana'))
720
>>> len(list(multiset_permutations('banana')))
60
```

sympy.utilities.iterables.**necklaces**(*n*, *k*, *free=False*)

A routine to generate necklaces that may (free=True) or may not (free=False) be turned over to be viewed. The "necklaces" returned are comprised of `n` integers (beads) with `k` different values (colors). Only unique necklaces are returned.

Examples

```
>>> from sympy.utilities.iterables import necklaces, bracelets
>>> def show(s, i):
...     return ''.join(s[j] for j in i)
```

The "unrestricted necklace" is sometimes also referred to as a "bracelet" (an object that can be turned over, a sequence that can be reversed) and the term "necklace" is used to imply a sequence that cannot be reversed. So ACB == ABC for a bracelet (rotate and reverse) while the two are different for a necklace since rotation alone cannot make the two sequences the same.

(mnemonic: Bracelets can be viewed Backwards, but Not Necklaces.)

```
>>> B = [show('ABC', i) for i in bracelets(3, 3)]
>>> N = [show('ABC', i) for i in necklaces(3, 3)]
>>> set(N) - set(B)
{'ACB'}
```

```
>>> list(necklaces(4, 2))
[(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 1),
 (0, 1, 0, 1), (0, 1, 1, 1), (1, 1, 1, 1)]
```

```
>>> [show('.o', i) for i in bracelets(4, 2)]
['....', '...o', '..oo', '.o.o', '.ooo', 'oooo']
```

References

[R970]

sympy.utilities.iterables.**numbered_symbols**(*prefix='x'*, *cls=None*, *start=0*, *exclude=()*, **args*, ***assumptions*)

Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in `exclude`.

Parameters

prefix : str, optional

The prefix to use. By default, this function will generate symbols of the form “x0”, “x1”, etc.

cls : class, optional

The class to use. By default, it uses `Symbol`, but you can also use `Wild` or `Dummy`.

start : int, optional

The start number. By default, it is 0.

Returns

sym : Symbol

The subscripted symbols.

sympy.utilities.iterables.**ordered_partitions**(*n*, *m=None*, *sort=True*)

Generates ordered partitions of integer `n`.

Parameters

m : integer (default None)

The default value gives partitions of all sizes else only those with size m. In addition, if `m` is not None then partitions are generated *in place* (see examples).

sort : bool (default True)

Controls whether partitions are returned in sorted order when `m` is not None; when False, the partitions are returned as fast as possible with elements sorted, but when m|n the partitions will not be in ascending lexicographical order.

Examples

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

All partitions of 5 in ascending lexicographical:

```
>>> for p in ordered_partitions(5):
...     print(p)
[1, 1, 1, 1, 1]
[1, 1, 1, 2]
[1, 1, 3]
[1, 2, 2]
[1, 4]
[2, 3]
[5]
```

Only partitions of 5 with two parts:

```
>>> for p in ordered_partitions(5, 2):
...     print(p)
[1, 4]
[2, 3]
```

When `m` is given, a given list objects will be used more than once for speed reasons so you will not see the correct partitions unless you make a copy of each as it is generated:

```
>>> [p for p in ordered_partitions(7, 3)]
[[1, 1, 1], [1, 1, 1], [1, 1, 1], [2, 2, 2]]
>>> [list(p) for p in ordered_partitions(7, 3)]
[[1, 1, 5], [1, 2, 4], [1, 3, 3], [2, 2, 3]]
```

When `n` is a multiple of `m`, the elements are still sorted but the partitions themselves will be *unordered* if sort is False; the default is to return them in ascending lexicographical order.

```
>>> for p in ordered_partitions(6, 2):
...     print(p)
[1, 5]
[2, 4]
[3, 3]
```

But if speed is more important than ordering, sort can be set to False:

```
>>> for p in ordered_partitions(6, 2, sort=False):
...     print(p)
[1, 5]
[3, 3]
[2, 4]
```

References

[R971], [R972]

sympy.utilities.iterables.**partitions**(*n*, *m=None*, *k=None*, *size=False*)

Generate all partitions of positive integer, n.

Parameters

m : integer (default gives partitions of all sizes)

limits number of parts in partition (mnemonic: m, maximum parts)

k : integer (default gives partitions number from 1 through n)

limits the numbers that are kept in the partition (mnemonic: k, keys)

size : bool (default False, only partition is returned)

when `True` then (M, P) is returned where M is the sum of the multiplicities and P is the generated partition.

Each partition is represented as a dictionary, mapping an integer to the number of copies of that integer in the partition. For example, the first partition of 4 returned is {4: 1}, "4: one of them".

Examples

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

The numbers appearing in the partition (the key of the returned dict) are limited with k:

```
>>> for p in partitions(6, k=2):
...     print(p)
{2: 3}
{1: 2, 2: 2}
{1: 4, 2: 1}
{1: 6}
```

The maximum number of parts in the partition (the sum of the values in the returned dict) are limited with m (default value, None, gives partitions from 1 through n):

```
>>> for p in partitions(6, m=2):
...     print(p)
...
{6: 1}
{1: 1, 5: 1}
{2: 1, 4: 1}
{3: 2}
```

See also:

sympy.combinatorics.partitions.Partition (page 251), *sympy.combinatorics.partitions.IntegerPartition* (page 253)

References

[R973]

sympy.utilities.iterables.**permute_signs**(*t*)

Return iterator in which the signs of non-zero elements of t are permuted.

Examples

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

sympy.utilities.iterables.**postfixes**(*seq*)

Generate all postfixes of a sequence.

Examples

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

```
>>> list(postfixes([1,2,3,4]))
[[4], [3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

sympy.utilities.iterables.**prefixes**(*seq*)

Generate all prefixes of a sequence.

Examples

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

```
>>> list(prefixes([1,2,3,4]))
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
```

sympy.utilities.iterables.**random_derangement**(*t*, *choice=None*, *strict=True*)

Return a list of elements in which none are in the same positions as they were originally. If an element fills more than half of the positions then an error will be raised since no derangement is possible. To obtain a derangement of as many items as possible–with some of the most numerous remaining in their original positions–pass $strict = False$. To produce a pseudorandom derangment, pass a pseudorandom selector like $choice$ (see below).

Examples

```
>>> from sympy.utilities.iterables import random_derangement
>>> t = 'SymPy: a CAS in pure Python'
>>> d = random_derangement(t)
>>> all(i != j for i, j in zip(d, t))
True
```

A predictable result can be obtained by using a pseudorandom generator for the choice:

```
>>> from sympy.core.random import seed, choice as c
>>> seed(1)
>>> d = [''.join(random_derangement(t, c)) for i in range(5)]
>>> assert len(set(d)) != 1  # we got different values
```

By reseeding, the same sequence can be obtained:

```
>>> seed(1)
>>> d2 = [''.join(random_derangement(t, c)) for i in range(5)]
>>> assert d == d2
```

sympy.utilities.iterables.**reshape**(*seq*, *how*)

Reshape the sequence according to the template in how.

Examples

```
>>> from sympy.utilities import reshape
>>> seq = list(range(1, 9))
```

```
>>> reshape(seq, [4]) # lists of 4
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
>>> reshape(seq, (4,)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

```
>>> reshape(seq, (2, 2)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

```
>>> reshape(seq, (2, [2])) # (i, i, [i, i])
[(1, 2, [3, 4]), (5, 6, [7, 8])]
```

```
>>> reshape(seq, ((2,), [2])) # etc....
[((1, 2), [3, 4]), ((5, 6), [7, 8])]
```

```
>>> reshape(seq, (1, [2], 1))
[(1, [2, 3], 4), (5, [6, 7], 8)]
```

```
>>> reshape(tuple(seq), ([[1], 1, (2,)],))
((([1], 2, (3, 4)],), ([[5], 6, (7, 8)],))
```

```
>>> reshape(tuple(seq), ([1], 1, (2,)))
(([1], 2, (3, 4)), ([5], 6, (7, 8)))
```

```
>>> reshape(list(range(12)), [2, [3], {2}, (1, (3,), 1)])
[[0, 1, [2, 3, 4], {5, 6}, (7, (8, 9, 10), 11)]]
```

sympy.utilities.iterables.**rotate_left**(*x*, *y*)

Left rotates a list x by the number of steps specified in y.

Examples

```
>>> from sympy.utilities.iterables import rotate_left
>>> a = [0, 1, 2]
>>> rotate_left(a, 1)
[1, 2, 0]
```

sympy.utilities.iterables.**rotate_right**(*x*, *y*)

Right rotates a list x by the number of steps specified in y.

Examples

```
>>> from sympy.utilities.iterables import rotate_right
>>> a = [0, 1, 2]
>>> rotate_right(a, 1)
[2, 0, 1]
```

sympy.utilities.iterables.**rotations**(*s*, *dir=1*)

Return a generator giving the items in s as list where each subsequent list has the items rotated to the left (default) or right (`dir=-1`) relative to the previous list.

Examples

```
>>> from sympy import rotations
>>> list(rotations([1,2,3]))
[[1, 2, 3], [2, 3, 1], [3, 1, 2]]
>>> list(rotations([1,2,3], -1))
[[1, 2, 3], [3, 1, 2], [2, 3, 1]]
```

sympy.utilities.iterables.**roundrobin**(**iterables*)

roundrobin recipe taken from itertools documentation: https://docs.python.org/3/library/itertools.html#recipes

roundrobin('ABC', 'D', 'EF') -> A D E B F C

Recipe credited to George Sakkis

sympy.utilities.iterables.**runs**(*seq*, *op=<built-in function gt>*)

Group the sequence into lists in which successive elements all compare the same with the comparison operator, `op`: op(seq[i + 1], seq[i]) is True from all elements in a run.

Examples

```
>>> from sympy.utilities.iterables import runs
>>> from operator import ge
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2])
[[0, 1, 2], [2], [1, 4], [3], [2], [2]]
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2], op=ge)
[[0, 1, 2, 2], [1, 4], [3], [2, 2]]
```

sympy.utilities.iterables.**sift**(*seq*, *keyfunc*, *binary=False*)

Sift the sequence, `seq` according to `keyfunc`.

Returns

When `binary` is `False` (default), the output is a dictionary where elements of `seq` are stored in a list keyed to the value of keyfunc for that element. If `binary` is True then a tuple with lists `T` and `F` are returned where `T` is a list containing elements of seq for which `keyfunc` was `True` and `F` containing those elements for which `keyfunc` was `False`; a ValueError is raised if the `keyfunc` is not binary.

Examples

```
>>> from sympy.utilities import sift
>>> from sympy.abc import x, y
>>> from sympy import sqrt, exp, pi, Tuple
```

```
>>> sift(range(5), lambda x: x % 2)
{0: [0, 2, 4], 1: [1, 3]}
```

sift() returns a defaultdict() object, so any key that has no matches will give [].

```
>>> sift([x], lambda x: x.is_commutative)
{True: [x]}
>>> _[False]
[]
```

Sometimes you will not know how many keys you will get:

```
>>> sift([sqrt(x), exp(x), (y**x)**2],
...      lambda x: x.as_base_exp()[0])
{E: [exp(x)], x: [sqrt(x)], y: [y**(2*x)]}
```

Sometimes you expect the results to be binary; the results can be unpacked by setting `binary` to True:

```
>>> sift(range(4), lambda x: x % 2, binary=True)
([1, 3], [0, 2])
>>> sift(Tuple(1, pi), lambda x: x.is_rational, binary=True)
([1], [pi])
```

A ValueError is raised if the predicate was not actually binary (which is a good test for the logic where sifting is used and binary results were expected):

```
>>> unknown = exp(1) - pi  # the rationality of this is unknown
>>> args = Tuple(1, pi, unknown)
>>> sift(args, lambda x: x.is_rational, binary=True)
Traceback (most recent call last):
...
ValueError: keyfunc gave non-binary output
```

The non-binary sifting shows that there were 3 keys generated:

```
>>> set(sift(args, lambda x: x.is_rational).keys())
{None, False, True}
```

If you need to sort the sifted items it might be better to use `ordered` which can economically apply multiple sort keys to a sequence while sorting.

See also:

ordered (page 1076)

sympy.utilities.iterables.**signed_permutations**(*t*)

Return iterator in which the signs of non-zero elements of t and the order of the elements are permuted.

Examples

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

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

Strongly connected components of a directed graph in reverse topological order.

Parameters

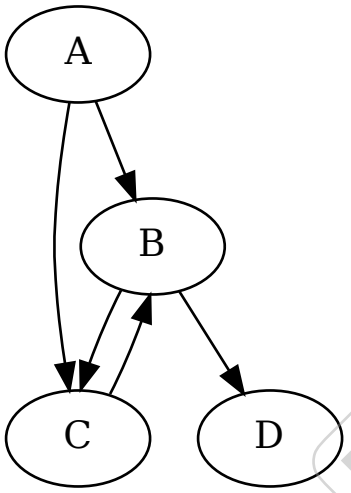
graph : tuple[list, list[tuple[T, T]]

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

Examples

Consider a directed graph (in dot notation):

```
digraph {
    A -> B
    A -> C
    B -> C
    C -> B
    B -> D
}
```



where vertices are the letters A, B, C and D. This graph can be encoded using Python's elementary data structures as follows:

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

The strongly connected components of this graph can be computed as

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

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

This also gives the components in reverse topological order.

Since the subgraph containing B and C has a cycle they must be together in a strongly connected component. A and D are connected to the rest of the graph but not in a cyclic manner so they appear as their own strongly connected components.

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 strongly connected components in $O(|V| + |E|)$ (linear) time.

See also:

sympy.utilities.iterables.connected_components (page 2072)

References

[R974], [R975]

sympy.utilities.iterables.**subsets**(*seq*, *k=None*, *repetition=False*)

Generates all k-subsets (combinations) from an n-element set, `seq`.

A k-subset of an n-element set is any subset of length exactly k. The number of k-subsets of an n-element set is given by `binomial(n, k)`, whereas there are 2^n subsets all together. If k is `None` then all 2^n subsets will be returned from shortest to longest.

Examples

```
>>> from sympy import subsets
```

`subsets(seq, k)` will return the $\frac{n!}{k!(n-k)!}$ k-subsets (combinations) without repetition, i.e. once an item has been removed, it can no longer be "taken":

```
>>> list(subsets([1, 2], 2))
[(1, 2)]
>>> list(subsets([1, 2]))
[(), (1,), (2,), (1, 2)]
>>> list(subsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
```

`subsets(seq, k, repetition=True)` will return the $\frac{(n-1+k)!}{k!(n-1)!}$ combinations *with* repetition:

```
>>> list(subsets([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(subsets([0, 1], 3, repetition=False))
[]
>>> list(subsets([0, 1], 3, repetition=True))
[(0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)]
```

sympy.utilities.iterables.**take**(*iter*, *n*)

Return `n` items from `iter` iterator.

sympy.utilities.iterables.**topological_sort**(*graph*, *key=None*)

Topological sort of graph's vertices.

Parameters

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

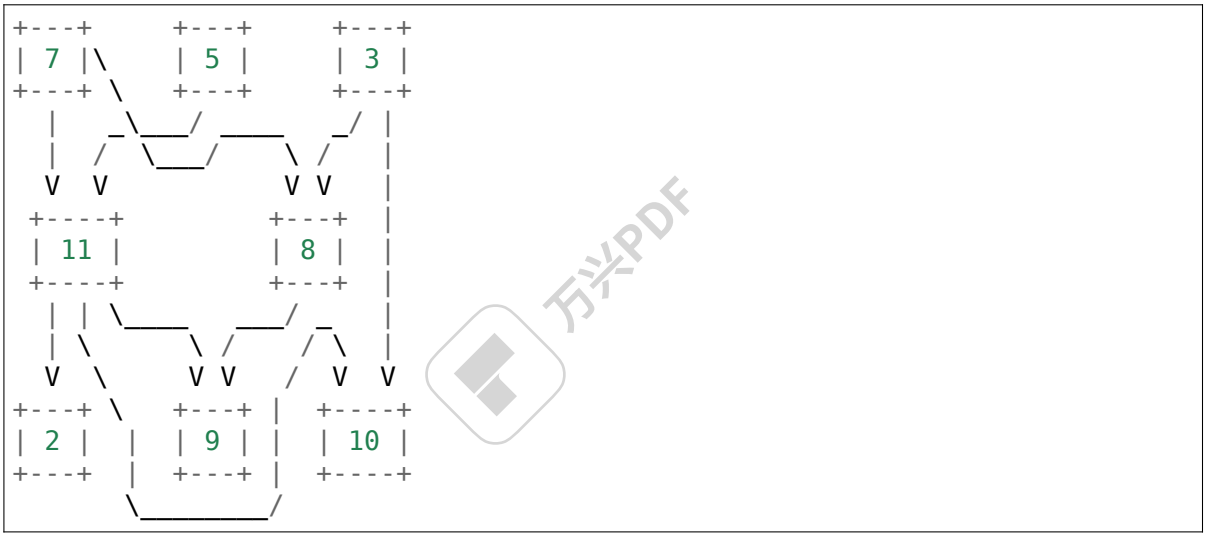
A tuple consisting of a list of vertices and a list of edges of a graph to be sorted topologically.

key : callable[T] (optional)

Ordering key for vertices on the same level. By default the natural (e.g. lexicographic) ordering is used (in this case the base type must implement ordering relations).

Examples

Consider a graph:



where vertices are integers. This graph can be encoded using elementary Python's data structures as follows:

```
>>> V = [2, 3, 5, 7, 8, 9, 10, 11]
>>> E = [(7, 11), (7, 8), (5, 11), (3, 8), (3, 10),
...      (11, 2), (11, 9), (11, 10), (8, 9)]
```

To compute a topological sort for graph `(V, E)` issue:

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

>>> topological_sort((V, E))
[3, 5, 7, 8, 11, 2, 9, 10]
```

If specific tie breaking approach is needed, use `key` parameter:

```
>>> topological_sort((V, E), key=lambda v: -v)
[7, 5, 11, 3, 10, 8, 9, 2]
```

Only acyclic graphs can be sorted. If the input graph has a cycle, then `ValueError` will be raised:

```
>>> topological_sort((V, E + [(10, 7)]))
Traceback (most recent call last):
...
ValueError: cycle detected
```

References

[R976]

sympy.utilities.iterables.**unflatten**(*iter*, *n=2*)

Group `iter` into tuples of length `n`. Raise an error if the length of `iter` is not a multiple of `n`.

sympy.utilities.iterables.**uniq**(*seq*, *result=None*)

Yield unique elements from `seq` as an iterator. The second parameter `result` is used internally; it is not necessary to pass anything for this.

Note: changing the sequence during iteration will raise a RuntimeError if the size of the sequence is known; if you pass an iterator and advance the iterator you will change the output of this routine but there will be no warning.

Examples

```
>>> from sympy.utilities.iterables import uniq
>>> dat = [1, 4, 1, 5, 4, 2, 1, 2]
>>> type(uniq(dat)) in (list, tuple)
False
```

```
>>> list(uniq(dat))
[1, 4, 5, 2]
>>> list(uniq(x for x in dat))
[1, 4, 5, 2]
>>> list(uniq([[1], [2, 1], [1]]))
[[1], [2, 1]]
```

sympy.utilities.iterables.**variations**(*seq*, *n*, *repetition=False*)

Returns an iterator over the n-sized variations of `seq` (size N). `repetition` controls whether items in `seq` can appear more than once;

Examples

`variations(seq, n)` will return $\frac{N!}{(N-n)!}$ permutations without repetition of `seq`'s elements:

```
>>> from sympy import variations
>>> list(variations([1, 2], 2))
[(1, 2), (2, 1)]
```

`variations(seq, n, True)` will return the N^n permutations obtained by allowing repetition of elements:

```
>>> list(variations([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(variations([0, 1], 3, repetition=False))
[]
>>> list(variations([0, 1], 3, repetition=True))[:4]
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)]
```

See also:

itertools.permutations, itertools.product

Lambdify

This module provides convenient functions to transform SymPy expressions to lambda functions which can be used to calculate numerical values very fast.

`sympy.utilities.lambdify.`**`implemented_function`**(*symfunc, implementation*)

Add numerical `implementation` to function `symfunc`.

`symfunc` can be an `UndefinedFunction` instance, or a name string. In the latter case we create an `UndefinedFunction` instance with that name.

Be aware that this is a quick workaround, not a general method to create special symbolic functions. If you want to create a symbolic function to be used by all the machinery of SymPy you should subclass the `Function` class.

Parameters

symfunc : `str` or `UndefinedFunction` instance

If `str`, then create new `UndefinedFunction` with this as name. If `symfunc` is an Undefined function, create a new function with the same name and the implemented function attached.

implementation : callable

numerical implementation to be called by `evalf()` or `lambdify`

Returns

afunc : sympy.FunctionClass instance

function with attached implementation

Examples

```
>>> from sympy.abc import x
>>> from sympy.utilities.lambdify import implemented_function
>>> from sympy import lambdify
>>> f = implemented_function('f', lambda x: x+1)
>>> lam_f = lambdify(x, f(x))
>>> lam_f(4)
5
```

sympy.utilities.lambdify.**lambdastr**(*args*, *expr*, *printer=None*, *dummify=None*)

Returns a string that can be evaluated to a lambda function.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.utilities.lambdify import lambdastr
>>> lambdastr(x, x**2)
'lambda x: (x**2)'
>>> lambdastr((x,y,z), [z,y,x])
'lambda x,y,z: ([z, y, x])'
```

Although tuples may not appear as arguments to lambda in Python 3, lambdastr will create a lambda function that will unpack the original arguments so that nested arguments can be handled:

```
>>> lambdastr((x, (y, z)), x + y)
'lambda _0,_1: (lambda x,y,z: (x + y))(_0,_1[0],_1[1])'
```

sympy.utilities.lambdify.**lambdify**(*args*, *expr*, *modules=None*, *printer=None*, *use_imps=True*, *dummify=False*, *cse=False*)

Convert a SymPy expression into a function that allows for fast numeric evaluation.

Warning: This function uses `exec`, and thus should not be used on unsanitized input.

Deprecated since version 1.7: Passing a set for the *args* parameter is deprecated as sets are unordered. Use an ordered iterable such as a list or tuple.

Parameters

args : List[Symbol]

A variable or a list of variables whose nesting represents the nesting of the arguments that will be passed to the function.

Variables can be symbols, undefined functions, or matrix symbols.

```
>>> from sympy import Eq
>>> from sympy.abc import x, y, z
```

The list of variables should match the structure of how the arguments will be passed to the function. Simply enclose the parameters as they will be passed in a list.

To call a function like `f(x)` then `[x]` should be the first argument to `lambdify`; for this case a single `x` can also be used:

```
>>> f = lambdify(x, x + 1)
>>> f(1)
2
>>> f = lambdify([x], x + 1)
>>> f(1)
2
```

To call a function like `f(x, y)` then `[x, y]` will be the first argument of the `lambdify`:

```
>>> f = lambdify([x, y], x + y)
>>> f(1, 1)
2
```

To call a function with a single 3-element tuple like `f((x, y, z))` then `[(x, y, z)]` will be the first argument of the `lambdify`:

```
>>> f = lambdify([(x, y, z)], Eq(z**2, x**2 + y**2))
>>> f((3, 4, 5))
True
```

If two args will be passed and the first is a scalar but the second is a tuple with two arguments then the items in the list should match that structure:

```
>>> f = lambdify([x, (y, z)], x + y + z)
>>> f(1, (2, 3))
6
```

expr : Expr

An expression, list of expressions, or matrix to be evaluated.

Lists may be nested. If the expression is a list, the output will also be a list.

```
>>> f = lambdify(x, [x, [x + 1, x + 2]])
>>> f(1)
[1, [2, 3]]
```

If it is a matrix, an array will be returned (for the NumPy module).

```
>>> from sympy import Matrix
>>> f = lambdify(x, Matrix([x, x + 1]))
>>> f(1)
[[1]
[2]]
```

Note that the argument order here (variables then expression) is used to emulate the Python `lambda` keyword. `lambdify(x, expr)` works (roughly) like `lambda x: expr` (see *How It Works* (page 2106) below).

modules : str, optional

Specifies the numeric library to use.

If not specified, *modules* defaults to:

- `["scipy", "numpy"]` if SciPy is installed
- `["numpy"]` if only NumPy is installed
- `["math", "mpmath", "sympy"]` if neither is installed.

That is, SymPy functions are replaced as far as possible by either `scipy` or `numpy` functions if available, and Python's standard library `math`, or `mpmath` functions otherwise.

modules can be one of the following types:

- The strings `"math"`, `"mpmath"`, `"numpy"`, `"numexpr"`, `"scipy"`, `"sympy"`, or `"tensorflow"` or `"jax"`. This uses the corresponding printer and namespace mapping for that module.
- A module (e.g., `math`). This uses the global namespace of the module. If the module is one of the above known modules, it will also use the corresponding printer and namespace mapping (i.e., `modules=numpy` is equivalent to `modules="numpy"`).
- A dictionary that maps names of SymPy functions to arbitrary functions (e.g., `{'sin': custom_sin}`).
- A list that contains a mix of the arguments above, with higher priority given to entries appearing first (e.g., to use the NumPy module but override the `sin` function with a custom version, you can use `[{'sin': custom_sin}, 'numpy']`).

dummify : bool, optional

Whether or not the variables in the provided expression that are not valid Python identifiers are substituted with dummy symbols.

This allows for undefined functions like `Function('f')(t)` to be supplied as arguments. By default, the variables are only dummified if they are not valid Python identifiers.

Set `dummify=True` to replace all arguments with dummy symbols (if `args` is not a string) - for example, to ensure that the arguments do not redefine any built-in names.

cse : bool, or callable, optional

Large expressions can be computed more efficiently when common subexpressions are identified and precomputed before being used multiple time. Finding the subexpressions will make creation of the 'lambdify' function slower, however.

When `True`, `sympy.simplify.cse` is used, otherwise (the default) the user may pass a function matching the `cse` signature.

Explanation

For example, to convert the SymPy expression `sin(x) + cos(x)` to an equivalent NumPy function that numerically evaluates it:

```
>>> from sympy import sin, cos, symbols, lambdify
>>> import numpy as np
>>> x = symbols('x')
>>> expr = sin(x) + cos(x)
>>> expr
sin(x) + cos(x)
>>> f = lambdify(x, expr, 'numpy')
>>> a = np.array([1, 2])
>>> f(a)
[1.38177329 0.49315059]
```

The primary purpose of this function is to provide a bridge from SymPy expressions to numerical libraries such as NumPy, SciPy, NumExpr, mpmath, and tensorflow. In general, SymPy functions do not work with objects from other libraries, such as NumPy arrays, and functions from numeric libraries like NumPy or mpmath do not work on SymPy expressions. `lambdify` bridges the two by converting a SymPy expression to an equivalent numeric function.

The basic workflow with `lambdify` is to first create a SymPy expression representing whatever mathematical function you wish to evaluate. This should be done using only SymPy functions and expressions. Then, use `lambdify` to convert this to an equivalent function for numerical evaluation. For instance, above we created `expr` using the SymPy symbol `x` and SymPy functions `sin` and `cos`, then converted it to an equivalent NumPy function `f`, and called it on a NumPy array `a`.

Examples

```
>>> from sympy.utilities.lambdify import implemented_function
>>> from sympy import sqrt, sin, Matrix
>>> from sympy import Function
>>> from sympy.abc import w, x, y, z
```

```
>>> f = lambdify(x, x**2)
>>> f(2)
4
>>> f = lambdify((x, y, z), [z, y, x])
>>> f(1,2,3)
[3, 2, 1]
>>> f = lambdify(x, sqrt(x))
>>> f(4)
2.0
>>> f = lambdify((x, y), sin(x*y)**2)
>>> f(0, 5)
0.0
>>> row = lambdify((x, y), Matrix((x, x + y)).T, modules='sympy')
>>> row(1, 2)
Matrix([[1, 3]])
```

`lambdify` can be used to translate SymPy expressions into mpmath functions. This may be preferable to using `evalf` (which uses mpmath on the backend) in some cases.

```
>>> f = lambdify(x, sin(x), 'mpmath')
>>> f(1)
0.8414709848078965
```

Tuple arguments are handled and the lambdified function should be called with the same type of arguments as were used to create the function:

```
>>> f = lambdify((x, (y, z)), x + y)
>>> f(1, (2, 4))
3
```

The `flatten` function can be used to always work with flattened arguments:

```
>>> from sympy.utilities.iterables import flatten
>>> args = w, (x, (y, z))
>>> vals = 1, (2, (3, 4))
>>> f = lambdify(flatten(args), w + x + y + z)
>>> f(*flatten(vals))
10
```

Functions present in `expr` can also carry their own numerical implementations, in a callable attached to the `_imp_` attribute. This can be used with undefined functions using the `implemented_function` factory:

```
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> func = lambdify(x, f(x))
>>> func(4)
5
```

`lambdify` always prefers `_imp_` implementations to implementations in other namespaces, unless the `use_imps` input parameter is False.

Usage with Tensorflow:

```
>>> import tensorflow as tf
>>> from sympy import Max, sin, lambdify
>>> from sympy.abc import x
```

```
>>> f = Max(x, sin(x))
>>> func = lambdify(x, f, 'tensorflow')
```

After tensorflow v2, eager execution is enabled by default. If you want to get the compatible result across tensorflow v1 and v2 as same as this tutorial, run this line.

```
>>> tf.compat.v1.enable_eager_execution()
```

If you have eager execution enabled, you can get the result out immediately as you can use numpy.

If you pass tensorflow objects, you may get an `EagerTensor` object instead of value.

```
>>> result = func(tf.constant(1.0))
>>> print(result)
tf.Tensor(1.0, shape=(), dtype=float32)
>>> print(result.__class__)
<class 'tensorflow.python.framework.ops.EagerTensor'>
```

You can use `.numpy()` to get the numpy value of the tensor.

```
>>> result.numpy()
1.0
```

```
>>> var = tf.Variable(2.0)
>>> result = func(var) # also works for tf.Variable and tf.Placeholder
>>> result.numpy()
2.0
```

And it works with any shape array.

```
>>> tensor = tf.constant([[1.0, 2.0], [3.0, 4.0]])
>>> result = func(tensor)
>>> result.numpy()
[[1. 2.]
 [3. 4.]]
```


Notes

- For functions involving large array calculations, numexpr can provide a significant speedup over numpy. Please note that the available functions for numexpr are more limited than numpy but can be expanded with `implemented_function` and user defined subclasses of Function. If specified, numexpr may be the only option in modules. The official list of numexpr functions can be found at: https://numexpr.readthedocs.io/en/latest/user_guide.html#supported-functions
- In the above examples, the generated functions can accept scalar values or numpy arrays as arguments. However, in some cases the generated function relies on the input being a numpy array:

```
>>> import numpy
>>> from sympy import Piecewise
>>> from sympy.testing.pytest import ignore_warnings
>>> f = lambdify(x, Piecewise((x, x <= 1), (1/x, x > 1)), "numpy")
```

```
>>> with ignore_warnings(RuntimeWarning):
...     f(numpy.array([-1, 0, 1, 2]))
[-1.   0.   1.   0.5]
```

```
>>> f(0)
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

In such cases, the input should be wrapped in a numpy array:

```
>>> with ignore_warnings(RuntimeWarning):
...     float(f(numpy.array([0])))
0.0
```

Or if numpy functionality is not required another module can be used:

```
>>> f = lambdify(x, Piecewise((x, x <= 1), (1/x, x > 1)), "math")
>>> f(0)
0
```

How It Works

When using this function, it helps a great deal to have an idea of what it is doing. At its core, lambdify is nothing more than a namespace translation, on top of a special printer that makes some corner cases work properly.

To understand lambdify, first we must properly understand how Python namespaces work. Say we had two files. One called `sin_cos_sympy.py`, with

```
# sin_cos_sympy.py

from sympy.functions.elementary.trigonometric import (cos, sin)

def sin_cos(x):
    return sin(x) + cos(x)
```

and one called `sin_cos_numpy.py` with

```
# sin_cos_numpy.py

from numpy import sin, cos

def sin_cos(x):
    return sin(x) + cos(x)
```

The two files define an identical function `sin_cos`. However, in the first file, `sin` and `cos` are defined as the SymPy `sin` and `cos`. In the second, they are defined as the NumPy versions.

If we were to import the first file and use the `sin_cos` function, we would get something like

```
>>> from sin_cos_sympy import sin_cos
>>> sin_cos(1)
cos(1) + sin(1)
```

On the other hand, if we imported `sin_cos` from the second file, we would get

```
>>> from sin_cos_numpy import sin_cos
>>> sin_cos(1)
1.38177329068
```

In the first case we got a symbolic output, because it used the symbolic `sin` and `cos` functions from SymPy. In the second, we got a numeric result, because `sin_cos` used the numeric `sin` and `cos` functions from NumPy. But notice that the versions of `sin` and `cos` that were used was not inherent to the `sin_cos` function definition. Both `sin_cos` definitions are exactly the same. Rather, it was based on the names defined at the module where the `sin_cos` function was defined.

The key point here is that when function in Python references a name that is not defined in the function, that name is looked up in the "global" namespace of the module where that function is defined.

Now, in Python, we can emulate this behavior without actually writing a file to disk using the `exec` function. `exec` takes a string containing a block of Python code, and a dictionary that should contain the global variables of the module. It then executes the code "in" that dictionary, as if it were the module globals. The following is equivalent to the `sin_cos` defined in `sin_cos_sympy.py`:

```
>>> import sympy
>>> module_dictionary = {'sin': sympy.sin, 'cos': sympy.cos}
>>> exec('''
... def sin_cos(x):
...     return sin(x) + cos(x)
... ''', module_dictionary)
>>> sin_cos = module_dictionary['sin_cos']
>>> sin_cos(1)
cos(1) + sin(1)
```

and similarly with `sin_cos_numpy`:

```
>>> import numpy
>>> module_dictionary = {'sin': numpy.sin, 'cos': numpy.cos}
>>> exec('''
... def sin_cos(x):
...     return sin(x) + cos(x)
... ''', module_dictionary)
>>> sin_cos = module_dictionary['sin_cos']
>>> sin_cos(1)
1.38177329068
```

So now we can get an idea of how `lambdify` works. The name "lambdify" comes from the fact that we can think of something like `lambdify(x, sin(x) + cos(x), 'numpy')` as `lambda x: sin(x) + cos(x)`, where `sin` and `cos` come from the `numpy` namespace. This is also why the symbols argument is first in `lambdify`, as opposed to most SymPy functions where it comes after the expression: to better mimic the `lambda` keyword.

`lambdify` takes the input expression (like `sin(x) + cos(x)`) and

1. Converts it to a string
2. Creates a module globals dictionary based on the modules that are passed in (by default, it uses the NumPy module)
3. Creates the string "`def func({vars}): return {expr}`", where `{vars}` is the list of variables separated by commas, and `{expr}` is the string created in step 1., then `exec``s that string with the module globals namespace and returns ``func`.

In fact, functions returned by `lambdify` support inspection. So you can see exactly how they are defined by using `inspect.getsource`, or `??` if you are using IPython or the Jupyter notebook.

```
>>> f = lambdify(x, sin(x) + cos(x))
>>> import inspect
>>> print(inspect.getsource(f))
def _lambdifygenerated(x):
    return sin(x) + cos(x)
```

This shows us the source code of the function, but not the namespace it was defined in. We can inspect that by looking at the `__globals__` attribute of `f`:

```
>>> f.__globals__['sin']
<ufunc 'sin'>
>>> f.__globals__['cos']
<ufunc 'cos'>
>>> f.__globals__['sin'] is numpy.sin
True
```

This shows us that `sin` and `cos` in the namespace of `f` will be `numpy.sin` and `numpy.cos`.

Note that there are some convenience layers in each of these steps, but at the core, this is how `lambdify` works. Step 1 is done using the `LambdaPrinter` printers defined in the printing module (see *sympy.printing.lambdarepr* (page 2170)). This allows different SymPy expressions to define how they should be converted to a string for different modules. You can change which printer `lambdify` uses by passing a custom printer in to the `printer` argument.

Step 2 is augmented by certain translations. There are default translations for each module, but you can provide your own by passing a list to the `modules` argument. For instance,

```
>>> def mysin(x):
...     print('taking the sin of', x)
...     return numpy.sin(x)
...
>>> f = lambdify(x, sin(x), [{'sin': mysin}, 'numpy'])
>>> f(1)
taking the sin of 1
0.8414709848078965
```

The globals dictionary is generated from the list by merging the dictionary `{'sin': mysin}` and the module dictionary for NumPy. The merging is done so that earlier items take precedence, which is why `mysin` is used above instead of `numpy.sin`.

If you want to modify the way `lambdify` works for a given function, it is usually easiest to do so by modifying the globals dictionary as such. In more complicated cases, it may be necessary to create and pass in a custom printer.

Finally, step 3 is augmented with certain convenience operations, such as the addition of a docstring.

Understanding how `lambdify` works can make it easier to avoid certain gotchas when using it. For instance, a common mistake is to create a lambdified function for one module (say, NumPy), and pass it objects from another (say, a SymPy expression).

For instance, say we create

```
>>> from sympy.abc import x
>>> f = lambdify(x, x + 1, 'numpy')
```

Now if we pass in a NumPy array, we get that array plus 1

```
>>> import numpy
>>> a = numpy.array([1, 2])
>>> f(a)
[2 3]
```

But what happens if you make the mistake of passing in a SymPy expression instead of a NumPy array:

```
>>> f(x + 1)
x + 2
```

This worked, but it was only by accident. Now take a different lambdified function:

```
>>> from sympy import sin
>>> g = lambdify(x, x + sin(x), 'numpy')
```

This works as expected on NumPy arrays:

```
>>> g(a)
[1.84147098 2.90929743]
```

But if we try to pass in a SymPy expression, it fails

```
>>> try:
...     g(x + 1)
... # NumPy release after 1.17 raises TypeError instead of
... # AttributeError
... except (AttributeError, TypeError):
...     raise AttributeError()
Traceback (most recent call last):
...
AttributeError:
```

Now, let's look at what happened. The reason this fails is that `g` calls `numpy.sin` on the input expression, and `numpy.sin` does not know how to operate on a SymPy object. **As a general rule, NumPy functions do not know how to operate on SymPy expressions, and SymPy functions do not know how to operate on NumPy arrays. This is why lambdify exists: to provide a bridge between SymPy and NumPy.**

However, why is it that `f` did work? That's because `f` does not call any functions, it only adds 1. So the resulting function that is created, `def _lambdifygenerated(x): return x + 1` does not depend on the globals namespace it is defined in. Thus it works, but only by accident. A future version of `lambdify` may remove this behavior.

Be aware that certain implementation details described here may change in future versions of SymPy. The API of passing in custom modules and printers will not change, but the details of how a lambda function is created may change. However, the basic idea will remain the same, and understanding it will be helpful to understanding the behavior of lambdify.

In general: you should create lambdified functions for one module (say, NumPy), and only pass it input types that are compatible with that module (say, NumPy arrays). Remember that by default, if the `module` argument is not provided, `lambdify` creates functions using the NumPy and SciPy namespaces.

Memoization

`sympy.utilities.memoization.`**`assoc_recurrence_memo`**(*base_seq*)

Memo decorator for associated sequences defined by recurrence starting from base

base_seq(n) – callable to get base sequence elements

XXX works only for Pn0 = base_seq(0) cases XXX works only for m <= n cases

`sympy.utilities.memoization.`**`recurrence_memo`**(*initial*)

Memo decorator for sequences defined by recurrence

See usage examples e.g. in the specfun/combinatorial module

Miscellaneous

Miscellaneous stuff that does not really fit anywhere else.

`sympy.utilities.misc.`**`as_int`**(*n*, *strict=True*)

Convert the argument to a builtin integer.

The return value is guaranteed to be equal to the input. ValueError is raised if the input has a non-integral value. When `strict` is True, this uses __index__ and when it is False it uses `int`.

Examples

```
>>> from sympy.utilities.misc import as_int
>>> from sympy import sqrt, S
```

The function is primarily concerned with sanitizing input for functions that need to work with builtin integers, so anything that is unambiguously an integer should be returned as an int:

```
>>> as_int(S(3))
3
```

Floats, being of limited precision, are not assumed to be exact and will raise an error unless the `strict` flag is False. This precision issue becomes apparent for large floating point numbers:

```
>>> big = 1e23
>>> type(big) is float
True
>>> big == int(big)
True
>>> as_int(big)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: ... is not an integer
>>> as_int(big, strict=False)
9999999999999991611392
```

Input that might be a complex representation of an integer value is also rejected by default:

```
>>> one = sqrt(3 + 2*sqrt(2)) - sqrt(2)
>>> int(one) == 1
True
>>> as_int(one)
Traceback (most recent call last):
...
ValueError: ... is not an integer
```

sympy.utilities.misc.**debug**(**args*)

Print *args if SYMPY_DEBUG is True, else do nothing.

sympy.utilities.misc.**debug_decorator**(*func*)

If SYMPY_DEBUG is True, it will print a nice execution tree with arguments and results of all decorated functions, else do nothing.

sympy.utilities.misc.**filldedent**(*s*, *w=70*, ***kwargs*)

Strips leading and trailing empty lines from a copy of `s`, then dedents, fills and returns it.

Empty line stripping serves to deal with docstrings like this one that start with a newline after the initial triple quote, inserting an empty line at the beginning of the string.

Additional keyword arguments will be passed to `textwrap.fill()`.

See also:

strlines (page 2114), *rawlines* (page 2112)

sympy.utilities.misc.**find_executable**(*executable*, *path=None*)

Try to find 'executable' in the directories listed in 'path' (a string listing directories separated by 'os.pathsep'; defaults to os.environ['PATH']). Returns the complete filename or None if not found

sympy.utilities.misc.**func_name**(*x*, *short=False*)

Return function name of x (if defined) else the $type(x)$. If short is True and there is a shorter alias for the result, return the alias.

Examples

```
>>> from sympy.utilities.misc import func_name
>>> from sympy import Matrix
>>> from sympy.abc import x
>>> func_name(Matrix.eye(3))
'MutableDenseMatrix'
>>> func_name(x < 1)
'StrictLessThan'
>>> func_name(x < 1, short=True)
'Lt'
```

sympy.utilities.misc.**ordinal**(*num*)

Return ordinal number string of num, e.g. 1 becomes 1st.

sympy.utilities.misc.**rawlines**(*s*)

Return a cut-and-pastable string that, when printed, is equivalent to the input. Use this when there is more than one line in the string. The string returned is formatted so it can be indented nicely within tests; in some cases it is wrapped in the dedent function which has to be imported from textwrap.

Examples

Note: because there are characters in the examples below that need to be escaped because they are themselves within a triple quoted docstring, expressions below look more complicated than they would be if they were printed in an interpreter window.

```
>>> from sympy.utilities.misc import rawlines
>>> from sympy import TableForm
>>> s = str(TableForm([[1, 10]], headings=(None, ['a', 'bee'])))
>>> print(rawlines(s))
(
    'a bee\n'
    '-----\n'
    '1 10 '
)
>>> print(rawlines('''this
... that'''))
dedent('''\
    this
    that''')
```

```
>>> print(rawlines('''this
... that
... '''))
dedent('''\
    this
    that
    ''')
```

```
>>> s = """this
... is a triple '''
... """
>>> print(rawlines(s))
dedent("""\
    this
    is a triple '''
    """)
```

```
>>> print(rawlines('''this
... that
...     '''))
(
    'this\n'
    'that\n'
    '    '
)
```

See also:

filldedent (page 2111), *strlines* (page 2114)

sympy.utilities.misc.**replace**(*string*, **reps*)

Return `string` with all keys in `reps` replaced with their corresponding values, longer strings first, irrespective of the order they are given. `reps` may be passed as tuples or a single mapping.

Examples

```
>>> from sympy.utilities.misc import replace
>>> replace('foo', {'oo': 'ar', 'f': 'b'})
'bar'
>>> replace("spamham sha", ("spam", "eggs"), ("sha","md5"))
'eggsham md5'
```

There is no guarantee that a unique answer will be obtained if keys in a mapping overlap (i.e. are the same length and have some identical sequence at the beginning/end):

```
>>> reps = [
...     ('ab', 'x'),
...     ('bc', 'y')]
>>> replace('abc', *reps) in ('xc', 'ay')
True
```

References

[R977]

sympy.utilities.misc.**strlines**(*s*, *c=64*, *short=False*)

Return a cut-and-pastable string that, when printed, is equivalent to the input. The lines will be surrounded by parentheses and no line will be longer than c (default 64) characters. If the line contains newlines characters, the *rawlines* result will be returned. If `short` is True (default is False) then if there is one line it will be returned without bounding parentheses.

Examples

```
>>> from sympy.utilities.misc import strlines
>>> q = 'this is a long string that should be broken into shorter lines'
>>> print(strlines(q, 40))
(
'this is a long string that should be b'
'roken into shorter lines'
)
>>> q == (
... 'this is a long string that should be b'
... 'roken into shorter lines'
... )
True
```

See also:

filldedent (page 2111), *rawlines* (page 2112)

sympy.utilities.misc.**translate**(*s*, *a*, *b=None*, *c=None*)

Return `s` where characters have been replaced or deleted.

Syntax

translate(s, None, deletechars):
all characters in `deletechars` are deleted

translate(s, map [,deletechars]):
all characters in `deletechars` (if provided) are deleted then the replacements defined by map are made; if the keys of map are strings then the longer ones are handled first. Multicharacter deletions should have a value of ‘’.

translate(s, oldchars, newchars, deletechars)
all characters in `deletechars` are deleted then each character in `oldchars` is replaced with the corresponding character in `newchars`

Examples

```
>>> from sympy.utilities.misc import translate
>>> abc = 'abc'
>>> translate(abc, None, 'a')
'bc'
>>> translate(abc, {'a': 'x'}, 'c')
'xb'
>>> translate(abc, {'abc': 'x', 'a': 'y'})
'x'
```

```
>>> translate('abcd', 'ac', 'AC', 'd')
'AbC'
```

There is no guarantee that a unique answer will be obtained if keys in a mapping overlap are the same length and have some identical sequences at the beginning/end:

```
>>> translate(abc, {'ab': 'x', 'bc': 'y'}) in ('xc', 'ay')
True
```

PKGDATA

pkgdata is a simple, extensible way for a package to acquire data file resources.

The getResource function is equivalent to the standard idioms, such as the following minimal implementation:

```
import sys, os

def getResource(identifier, pkgname=__name__):
    pkgpath = os.path.dirname(sys.modules[pkgname].__file__)
    path = os.path.join(pkgpath, identifier)
    return open(os.path.normpath(path), mode='rb')
```

When a __loader__ is present on the module given by __name__, it will defer getResource to its get_data implementation and return it as a file-like object (such as StringIO).

sympy.utilities.pkgdata.**get_resource**(*identifier*, *pkgname='sympy.utilities.pkgdata'*)

Acquire a readable object for a given package name and identifier. An IOError will be raised if the resource cannot be found.

For example:

```
mydata = get_resource('mypkgdata.jpg').read()
```

Note that the package name must be fully qualified, if given, such that it would be found in sys.modules.

In some cases, getResource will return a real file object. In that case, it may be useful to use its name attribute to get the path rather than use it as a file-like object. For example, you may be handing data off to a C API.

Source Code Inspection

This module adds several functions for interactive source code inspection.

sympy.utilities.source.**get_class**(*lookup_view*)

Convert a string version of a class name to the object.

For example, get_class('sympy.core.Basic') will return class Basic located in module sympy.core

sympy.utilities.source.**get_mod_func**(*callback*)

splits the string path to a class into a string path to the module and the name of the class.

Examples

```
>>> from sympy.utilities.source import get_mod_func
>>> get_mod_func('sympy.core.basic.Basic')
('sympy.core.basic', 'Basic')
```

sympy.utilities.source.**source**(*object*)

Prints the source code of a given object.

Deprecated since version 1.3: The `source()` function is deprecated. Use `inspect.getsource()` or `??` in IPython/Jupyter instead.

Timing Utilities

Simple tools for timing functions' execution, when IPython is not available.

sympy.utilities.timeutils.**timed**(*func*, *setup='pass'*, *limit=None*)

Adaptively measure execution time of a function.

Interactive

Helper module for setting up interactive SymPy sessions.

Session

Tools for setting up interactive sessions.

sympy.interactive.session.**enable_automatic_int_sympification**(*shell*)

Allow IPython to automatically convert integer literals to Integer.

sympy.interactive.session.**enable_automatic_symbols**(*shell*)

Allow IPython to automatically create symbols (`isympy -a`).

sympy.interactive.session.**init_ipython_session**(*shell=None*, *argv=[]*, *auto_symbols=False*, *auto_int_to_Integer=False*)

Construct new IPython session.