sympy.calculus.singularities.**is_strictly_decreasing**(*expression*, *interval=Reals*, *symbol=None*)

Return whether the function is strictly decreasing in the given interval.

**Parameters**

**expression** : Expr

The target function which is being checked.

**interval** : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

**symbol** : Symbol, optional

The symbol present in expression which gets varied over the given range.

**Returns**

Boolean

True if `expression` is strictly decreasing in the given `interval`, False otherwise.

## Examples

```
>>> from sympy import is_strictly_decreasing
>>> from sympy.abc import x, y
>>> from sympy import S, Interval, oo
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Lopen(3, oo))
True
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, S(3)/2))
False
>>> is_strictly_decreasing(1/(x**2 - 3*x), Interval.Ropen(-oo, 1.5))
False
>>> is_strictly_decreasing(-x**2, Interval(-oo, 0))
False
>>> is_strictly_decreasing(-x**2 + y, Interval(-oo, 0), x)
False
```

sympy.calculus.singularities.**is_strictly_increasing**(*expression*, *interval=Reals*, *symbol=None*)

Return whether the function is strictly increasing in the given interval.

**Parameters**

**expression** : Expr

The target function which is being checked.

**interval** : Set, optional

The range of values in which we are testing (defaults to set of all real numbers).

**symbol** : Symbol, optional

The symbol present in expression which gets varied over the given range.

**Returns**
Boolean

True if `expression` is strictly increasing in the given `interval`, False otherwise.

## Examples

```
>>> from sympy import is_strictly_increasing
>>> from sympy.abc import x, y
>>> from sympy import Interval, oo
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.Ropen(-
↪oo, -2))
True
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.Lopen(3,
↪ oo))
True
>>> is_strictly_increasing(4*x**3 - 6*x**2 - 72*x + 30, Interval.open(-2,
↪ 3))
False
>>> is_strictly_increasing(-x**2, Interval(0, oo))
False
>>> is_strictly_increasing(-x**2 + y, Interval(-oo, 0), x)
False
```

`sympy.calculus.singularities.`**`monotonicity_helper`**(*expression*, *predicate*, *interval=Reals*, *symbol=None*)

Helper function for functions checking function monotonicity.

**Parameters**
**expression** : Expr

The target function which is being checked

**predicate** : function

The property being tested for. The function takes in an integer and returns a boolean. The integer input is the derivative and the boolean result should be true if the property is being held, and false otherwise.

**interval** : Set, optional

The range of values in which we are testing, defaults to all reals.

**symbol** : Symbol, optional

The symbol present in expression which gets varied over the given range.

**It returns a boolean indicating whether the interval in which**

**the function's derivative satisfies given predicate is a superset**

**of the given interval.**

**Returns**
Boolean

True if `predicate` is true for all the derivatives when `symbol` is varied in `range`, False otherwise.

`sympy.calculus.singularities.`**`singularities`**(*expression, symbol, domain=None*)

Find singularities of a given function.

**Parameters**

**expression** : Expr

The target function in which singularities need to be found.

**symbol** : Symbol

The symbol over the values of which the singularity in expression in being searched for.

**Returns**

Set

A set of values for `symbol` for which `expression` has a singularity. An `EmptySet` is returned if `expression` has no singularities for any given value of `Symbol`.

**Raises**

**NotImplementedError**

Methods for determining the singularities of this function have not been developed.

### Notes

This function does not find non-isolated singularities nor does it find branch points of the expression.

**Currently supported functions are:**

- univariate continuous (real or complex) functions

### Examples

```
>>> from sympy import singularities, Symbol, log
>>> x = Symbol('x', real=True)
>>> y = Symbol('y', real=False)
>>> singularities(x**2 + x + 1, x)
EmptySet
>>> singularities(1/(x + 1), x)
{-1}
>>> singularities(1/(y**2 + 1), y)
{-I, I}
>>> singularities(1/(y**3 + 1), y)
{-1, 1/2 - sqrt(3)*I/2, 1/2 + sqrt(3)*I/2}
>>> singularities(log(x), x)
{0}
```

### References

[R30]

## Finite difference weights

This module implements an algorithm for efficient generation of finite difference weights for ordinary differentials of functions for derivatives from 0 (interpolation) up to arbitrary order.

The core algorithm is provided in the finite difference weight generating function (`finite_diff_weights`), and two convenience functions are provided for:

- **estimating a derivative (or interpolate) directly from a series of points**
  is also provided (`apply_finite_diff`).
- **differentiating by using finite difference approximations**
  (`differentiate_finite`).

`sympy.calculus.finite_diff.`**`apply_finite_diff`**(*order, x_list, y_list, x0=0*)

Calculates the finite difference approximation of the derivative of requested order at `x0` from points provided in `x_list` and `y_list`.

**Parameters**

**order: int**

order of derivative to approximate. 0 corresponds to interpolation.

**x_list: sequence**

Sequence of (unique) values for the independent variable.

**y_list: sequence**

The function value at corresponding values for the independent variable in x_list.

**x0: Number or Symbol**

At what value of the independent variable the derivative should be evaluated. Defaults to 0.

**Returns**

sympy.core.add.Add or sympy.core.numbers.Number

The finite difference expression approximating the requested derivative order at `x0`.

### Examples

```
>>> from sympy import apply_finite_diff
>>> cube = lambda arg: (1.0*arg)**3
>>> xlist = range(-3,3+1)
>>> apply_finite_diff(2, xlist, map(cube, xlist), 2) - 12
-3.55271367880050e-15
```

we see that the example above only contain rounding errors. apply_finite_diff can also be used on more abstract objects:

```
>>> from sympy import IndexedBase, Idx
>>> x, y = map(IndexedBase, 'xy')
>>> i = Idx('i')
>>> x_list, y_list = zip(*[(x[i+j], y[i+j]) for j in range(-1,2)])
>>> apply_finite_diff(1, x_list, y_list, x[i])
((x[i + 1] - x[i])/(-x[i - 1] + x[i]) - 1)*y[i]/(x[i + 1] - x[i]) -
(x[i + 1] - x[i])*y[i - 1]/((x[i + 1] - x[i - 1])*(-x[i - 1] + x[i])) +
(-x[i - 1] + x[i])*y[i + 1]/((x[i + 1] - x[i - 1])*(x[i + 1] - x[i]))
```

## Notes

Order = 0 corresponds to interpolation. Only supply so many points you think makes sense to around x0 when extracting the derivative (the function need to be well behaved within that region). Also beware of Runge's phenomenon.

**See also:**

*sympy.calculus.finite_diff.finite_diff_weights* (page 242)

## References

Fortran 90 implementation with Python interface for numerics: finitediff

sympy.calculus.finite_diff.**differentiate_finite**(*expr, *symbols, points=1, x0=None, wrt=None, evaluate=False*)

Differentiate expr and replace Derivatives with finite differences.

**Parameters**

**expr** : expression

***symbols** : differentiate with respect to symbols

**points: sequence, coefficient or undefined function, optional**

see `Derivative.as_finite_difference`

**x0: number or Symbol, optional**

see `Derivative.as_finite_difference`

**wrt: Symbol, optional**

see `Derivative.as_finite_difference`

## Examples

```
>>> from sympy import sin, Function, differentiate_finite
>>> from sympy.abc import x, y, h
>>> f, g = Function('f'), Function('g')
>>> differentiate_finite(f(x)*g(x), x, points=[x-h, x+h])
-f(-h + x)*g(-h + x)/(2*h) + f(h + x)*g(h + x)/(2*h)
```

`differentiate_finite` works on any expression, including the expressions with embedded derivatives:

```
>>> differentiate_finite(f(x) + sin(x), x, 2)
-2*f(x) + f(x - 1) + f(x + 1) - 2*sin(x) + sin(x - 1) + sin(x + 1)
>>> differentiate_finite(f(x, y), x, y)
f(x - 1/2, y - 1/2) - f(x - 1/2, y + 1/2) - f(x + 1/2, y - 1/2) + f(x +␣
↪1/2, y + 1/2)
>>> differentiate_finite(f(x)*g(x).diff(x), x)
(-g(x) + g(x + 1))*f(x + 1/2) - (g(x) - g(x - 1))*f(x - 1/2)
```

To make finite difference with non-constant discretization step use undefined functions:

```
>>> dx = Function('dx')
>>> differentiate_finite(f(x)*g(x).diff(x), points=dx(x))
-(-g(x - dx(x)/2 - dx(x - dx(x)/2)/2)/dx(x - dx(x)/2) +
g(x - dx(x)/2 + dx(x - dx(x)/2)/2)/dx(x - dx(x)/2))*f(x - dx(x)/2)/dx(x)␣
↪+
(-g(x + dx(x)/2 - dx(x + dx(x)/2)/2)/dx(x + dx(x)/2) +
g(x + dx(x)/2 + dx(x + dx(x)/2)/2)/dx(x + dx(x)/2))*f(x + dx(x)/2)/dx(x)
```

`sympy.calculus.finite_diff.`**`finite_diff_weights`**`(`*order, x_list, x0=1*`)`

Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid (`x_list`) for derivatives at `x0` of order 0, 1, ..., up to `order` using a recursive formula. Order of accuracy is at least `len(x_list) - order`, if `x_list` is defined correctly.

**Parameters**

**order: int**

Up to what derivative order weights should be calculated. 0 corresponds to interpolation.

**x_list: sequence**

Sequence of (unique) values for the independent variable. It is useful (but not necessary) to order `x_list` from nearest to furthest from `x0`; see examples below.

**x0: Number or Symbol**

Root or value of the independent variable for which the finite difference weights should be generated. Default is `S.One`.

**Returns**

list

A list of sublists, each corresponding to coefficients for increasing derivative order, and each containing lists of coefficients for increasing subsets of x_list.

### Examples

```
>>> from sympy import finite_diff_weights, S
>>> res = finite_diff_weights(1, [-S(1)/2, S(1)/2, S(3)/2, S(5)/2], 0)
>>> res
[[[1, 0, 0, 0],
  [1/2, 1/2, 0, 0],
  [3/8, 3/4, -1/8, 0],
  [5/16, 15/16, -5/16, 1/16]],
 [[0, 0, 0, 0],
  [-1, 1, 0, 0],
  [-1, 1, 0, 0],
  [-23/24, 7/8, 1/8, -1/24]]]
>>> res[0][-1]  # FD weights for 0th derivative, using full x_list
[5/16, 15/16, -5/16, 1/16]
>>> res[1][-1]  # FD weights for 1st derivative
[-23/24, 7/8, 1/8, -1/24]
>>> res[1][-2]  # FD weights for 1st derivative, using x_list[:-1]
[-1, 1, 0, 0]
>>> res[1][-1][0]  # FD weight for 1st deriv. for x_list[0]
-23/24
>>> res[1][-1][1]  # FD weight for 1st deriv. for x_list[1], etc.
7/8
```

Each sublist contains the most accurate formula at the end. Note, that in the above example `res[1][1]` is the same as `res[1][2]`. Since res[1][2] has an order of accuracy of `len(x_list[:3]) - order = 3 - 1 = 2`, the same is true for `res[1][1]`!

```
>>> res = finite_diff_weights(1, [S(0), S(1), -S(1), S(2), -S(2)], 0)[1]
>>> res
[[0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0],
 [0, 1/2, -1/2, 0, 0],
 [-1/2, 1, -1/3, -1/6, 0],
 [0, 2/3, -2/3, -1/12, 1/12]]
>>> res[0]  # no approximation possible, using x_list[0] only
[0, 0, 0, 0, 0]
>>> res[1]  # classic forward step approximation
[-1, 1, 0, 0, 0]
>>> res[2]  # classic centered approximation
[0, 1/2, -1/2, 0, 0]
>>> res[3:]  # higher order approximations
[[-1/2, 1, -1/3, -1/6, 0], [0, 2/3, -2/3, -1/12, 1/12]]
```

Let us compare this to a differently defined `x_list`. Pay attention to `foo[i][k]` corresponding to the gridpoint defined by `x_list[k]`.

```
>>> foo = finite_diff_weights(1, [-S(2), -S(1), S(0), S(1), S(2)], 0)[1]
>>> foo
[[0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0],
 [1/2, -2, 3/2, 0, 0],
 [1/6, -1, 1/2, 1/3, 0],
```

(continues on next page)

(continued from previous page)

```
 [1/12, -2/3, 0, 2/3, -1/12]]
>>> foo[1]  # not the same and of lower accuracy as res[1]!
[-1, 1, 0, 0, 0]
>>> foo[2]  # classic double backward step approximation
[1/2, -2, 3/2, 0, 0]
>>> foo[4]  # the same as res[4]
[1/12, -2/3, 0, 2/3, -1/12]
```

Note that, unless you plan on using approximations based on subsets of x_list, the order of gridpoints does not matter.

The capability to generate weights at arbitrary points can be used e.g. to minimize Runge's phenomenon by using Chebyshev nodes:

```
>>> from sympy import cos, symbols, pi, simplify
>>> N, (h, x) = 4, symbols('h x')
>>> x_list = [x+h*cos(i*pi/(N)) for i in range(N,-1,-1)] # chebyshev␣
↪nodes
>>> print(x_list)
[-h + x, -sqrt(2)*h/2 + x, x, sqrt(2)*h/2 + x, h + x]
>>> mycoeffs = finite_diff_weights(1, x_list, 0)[1][4]
>>> [simplify(c) for c in  mycoeffs]
[(h**3/2 + h**2*x - 3*h*x**2 - 4*x**3)/h**4,
(-sqrt(2)*h**3 - 4*h**2*x + 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
(6*h**2*x - 8*x**3)/h**4,
(sqrt(2)*h**3 - 4*h**2*x - 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
(-h**3/2 + h**2*x + 3*h*x**2 - 4*x**3)/h**4]
```

## Notes

If weights for a finite difference approximation of 3rd order derivative is wanted, weights for 0th, 1st and 2nd order are calculated "for free", so are formulae using subsets of x_list. This is something one can take advantage of to save computational cost. Be aware that one should define x_list from nearest to furthest from x0. If not, subsets of x_list will yield poorer approximations, which might not grand an order of accuracy of len(x_list) - order.

**See also:**

*sympy.calculus.finite_diff.apply_finite_diff* (page 240)

## References

[R31]

sympy.calculus.util.**continuous_domain**(*f, symbol, domain*)

Returns the intervals in the given domain for which the function is continuous. This method is limited by the ability to determine the various singularities and discontinuities of the given function.

**Parameters**

**f** : *Expr* (page 947)

The concerned function.

**symbol** : *Symbol* (page 976)

The variable for which the intervals are to be determined.

**domain** : *Interval* (page 1194)

The domain over which the continuity of the symbol has to be checked.

**Returns**

*Interval* (page 1194)

Union of all intervals where the function is continuous.

**Raises**

**NotImplementedError**

If the method to determine continuity of such a function has not yet been developed.

## Examples

```
>>> from sympy import Interval, Symbol, S, tan, log, pi, sqrt
>>> from sympy.calculus.util import continuous_domain
>>> x = Symbol('x')
>>> continuous_domain(1/x, x, S.Reals)
Union(Interval.open(-oo, 0), Interval.open(0, oo))
>>> continuous_domain(tan(x), x, Interval(0, pi))
Union(Interval.Ropen(0, pi/2), Interval.Lopen(pi/2, pi))
>>> continuous_domain(sqrt(x - 2), x, Interval(-5, 5))
Interval(2, 5)
>>> continuous_domain(log(2*x - 1), x, S.Reals)
Interval.open(1/2, oo)
```

sympy.calculus.util.**function_range**(*f, symbol, domain*)

Finds the range of a function in a given domain. This method is limited by the ability to determine the singularities and determine limits.

**Parameters**

**f** : *Expr* (page 947)

The concerned function.

**symbol** : *Symbol* (page 976)

The variable for which the range of function is to be determined.

**domain** : *Interval* (page 1194)

The domain under which the range of the function has to be found.

**Returns**

*Interval* (page 1194)

Union of all ranges for all intervals under domain where function is continuous.

**Raises**

**NotImplementedError**

If any of the intervals, in the given domain, for which function is continuous are not finite or real, OR if the critical points of the function on the domain cannot be found.

## Examples

```
>>> from sympy import Interval, Symbol, S, exp, log, pi, sqrt, sin, tan
>>> from sympy.calculus.util import function_range
>>> x = Symbol('x')
>>> function_range(sin(x), x, Interval(0, 2*pi))
Interval(-1, 1)
>>> function_range(tan(x), x, Interval(-pi/2, pi/2))
Interval(-oo, oo)
>>> function_range(1/x, x, S.Reals)
Union(Interval.open(-oo, 0), Interval.open(0, oo))
>>> function_range(exp(x), x, S.Reals)
Interval.open(0, oo)
>>> function_range(log(x), x, S.Reals)
Interval(-oo, oo)
>>> function_range(sqrt(x), x, Interval(-5, 9))
Interval(0, 3)
```

`sympy.calculus.util.`**`is_convex`**(*f, *syms, domain=Reals*)

Determines the convexity of the function passed in the argument.

**Parameters**

**f** : *`Expr`* (page 947)

The concerned function.

**syms** : Tuple of *`Symbol`* (page 976)

The variables with respect to which the convexity is to be determined.

**domain** : *`Interval`* (page 1194), optional

The domain over which the convexity of the function has to be checked. If unspecified, S.Reals will be the default domain.

**Returns**

bool

The method returns `True` if the function is convex otherwise it returns `False`.

**Raises**

**NotImplementedError**

The check for the convexity of multivariate functions is not implemented yet.

### Notes

To determine concavity of a function pass $-f$ as the concerned function. To determine logarithmic convexity of a function pass $\log(f)$ as concerned function. To determine logartihmic concavity of a function pass $-\log(f)$ as concerned function.

Currently, convexity check of multivariate functions is not handled.

### Examples

```
>>> from sympy import is_convex, symbols, exp, oo, Interval
>>> x = symbols('x')
>>> is_convex(exp(x), x)
True
>>> is_convex(x**3, x, domain = Interval(-1, oo))
False
>>> is_convex(1/x**2, x, domain=Interval.open(0, oo))
True
```

### References

[R32], [R33], [R34], [R35], [R36]

sympy.calculus.util.**lcim**(*numbers*)

Returns the least common integral multiple of a list of numbers.

The numbers can be rational or irrational or a mixture of both. $None$ is returned for incommensurable numbers.

**Parameters**

**numbers** : list

Numbers (rational and/or irrational) for which lcim is to be found.

**Returns**

number

lcim if it exists, otherwise None for incommensurable numbers.

### Examples

```
>>> from sympy.calculus.util import lcim
>>> from sympy import S, pi
>>> lcim([S(1)/2, S(3)/4, S(5)/6])
15/2
>>> lcim([2*pi, 3*pi, pi, pi/2])
6*pi
>>> lcim([S(1), 2*pi])
```

sympy.calculus.util.**maximum**(*f, symbol, domain=Reals*)

Returns the maximum value of a function in the given domain.

**Parameters**

**f** : *Expr* (page 947)

The concerned function.

**symbol** : *Symbol* (page 976)

The variable for maximum value needs to be determined.

**domain** : *Interval* (page 1194)

The domain over which the maximum have to be checked. If unspecified, then the global maximum is returned.

**Returns**

number

Maximum value of the function in given domain.

## Examples

```
>>> from sympy import Interval, Symbol, S, sin, cos, pi, maximum
>>> x = Symbol('x')
```

```
>>> f = -x**2 + 2*x + 5
>>> maximum(f, x, S.Reals)
6
```

```
>>> maximum(sin(x), x, Interval(-pi, pi/4))
sqrt(2)/2
```

```
>>> maximum(sin(x)*cos(x), x)
1/2
```

sympy.calculus.util.**minimum**(*f, symbol, domain=Reals*)

Returns the minimum value of a function in the given domain.

**Parameters**

**f** : *Expr* (page 947)

The concerned function.

**symbol** : *Symbol* (page 976)

The variable for minimum value needs to be determined.

**domain** : *Interval* (page 1194)

The domain over which the minimum have to be checked. If unspecified, then the global minimum is returned.

**Returns**

number

Minimum value of the function in the given domain.

### Examples

```
>>> from sympy import Interval, Symbol, S, sin, cos, minimum
>>> x = Symbol('x')
```

```
>>> f = x**2 + 2*x + 5
>>> minimum(f, x, S.Reals)
4
```

```
>>> minimum(sin(x), x, Interval(2, 3))
sin(3)
```

```
>>> minimum(sin(x)*cos(x), x)
-1/2
```

`sympy.calculus.util.`**`not_empty_in`**(*finset_intersection*, **syms*)

Finds the domain of the functions in `finset_intersection` in which the `finite_set` is not-empty

**Parameters**

**finset_intersection** : Intersection of FiniteSet

The unevaluated intersection of FiniteSet containing real-valued functions with Union of Sets

**syms** : Tuple of symbols

Symbol for which domain is to be found

**Raises**

**NotImplementedError**

The algorithms to find the non-emptiness of the given FiniteSet are not yet implemented.

**ValueError**

The input is not valid.

**RuntimeError**

It is a bug, please report it to the github issue tracker (https://github.com/sympy/sympy/issues).

### Examples

```
>>> from sympy import FiniteSet, Interval, not_empty_in, oo
>>> from sympy.abc import x
>>> not_empty_in(FiniteSet(x/2).intersect(Interval(0, 1)), x)
Interval(0, 2)
>>> not_empty_in(FiniteSet(x, x**2).intersect(Interval(1, 2)), x)
Union(Interval(1, 2), Interval(-sqrt(2), -1))
>>> not_empty_in(FiniteSet(x**2/(x + 2)).intersect(Interval(1, oo)), x)
Union(Interval.Lopen(-2, -1), Interval(2, oo))
```

sympy.calculus.util.**periodicity**(*f*, *symbol*, *check=False*)

Tests the given function for periodicity in the given symbol.

**Parameters**

**f** : *Expr* (page 947).

The concerned function.

**symbol** : *Symbol* (page 976)

The variable for which the period is to be determined.

**check** : bool, optional

The flag to verify whether the value being returned is a period or not.

**Returns**

period

The period of the function is returned. None is returned when the function is aperiodic or has a complex period. The value of $0$ is returned as the period of a constant function.

**Raises**

**NotImplementedError**

The value of the period computed cannot be verified.

## Notes

Currently, we do not support functions with a complex period. The period of functions having complex periodic values such as exp, sinh is evaluated to None.

The value returned might not be the "fundamental" period of the given function i.e. it may not be the smallest periodic value of the function.

The verification of the period through the check flag is not reliable due to internal simplification of the given expression. Hence, it is set to False by default.

## Examples

```
>>> from sympy import periodicity, Symbol, sin, cos, tan, exp
>>> x = Symbol('x')
>>> f = sin(x) + sin(2*x) + sin(3*x)
>>> periodicity(f, x)
2*pi
>>> periodicity(sin(x)*cos(x), x)
pi
>>> periodicity(exp(tan(2*x) - 1), x)
pi/2
>>> periodicity(sin(4*x)**cos(2*x), x)
pi
>>> periodicity(exp(x), x)
```

sympy.calculus.util.**stationary_points**(*f*, *symbol*, *domain=Reals*)

Returns the stationary points of a function (where derivative of the function is 0) in the given domain.

**Parameters**

**f** : *Expr* (page 947)

The concerned function.

**symbol** : *Symbol* (page 976)

The variable for which the stationary points are to be determined.

**domain** : *Interval* (page 1194)

The domain over which the stationary points have to be checked. If unspecified, `S.Reals` will be the default domain.

**Returns**

Set

A set of stationary points for the function. If there are no stationary point, an *EmptySet* (page 1202) is returned.

### Examples

```
>>> from sympy import Interval, Symbol, S, sin, pi, pprint, stationary_
↪points
>>> x = Symbol('x')
```

```
>>> stationary_points(1/x, x, S.Reals)
EmptySet
```

```
>>> pprint(stationary_points(sin(x), x), use_unicode=False)
          pi                              3*pi
{2*n*pi + -- | n in Integers} U {2*n*pi + ---- | n in Integers}
          2                                2
```

```
>>> stationary_points(sin(x),x, Interval(0, 4*pi))
{pi/2, 3*pi/2, 5*pi/2, 7*pi/2}
```

## Combinatorics

### Contents

### Partitions

**class** sympy.combinatorics.partitions.**Partition**(**partition*)

This class represents an abstract partition.

A partition is a set of disjoint sets whose union equals a given set.

**See also:**

*sympy.utilities.iterables.partitions* (page 2089), *sympy.utilities.iterables.multiset_partitions* (page 2084)

**property RGS**

Returns the "restricted growth string" of the partition.

### Explanation

The RGS is returned as a list of indices, L, where L[i] indicates the block in which element i appears. For example, in a partition of 3 elements (a, b, c) into 2 blocks ([c], [a, b]) the RGS is [1, 1, 0]: “a” is in block 1, “b” is in block 1 and “c” is in block 0.

### Examples

```
>>> from sympy.combinatorics import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.members
(1, 2, 3, 4, 5)
>>> a.RGS
(0, 0, 1, 2, 2)
>>> a + 1
Partition({3}, {4}, {5}, {1, 2})
>>> _.RGS
(0, 0, 1, 2, 3)
```

**classmethod from_rgs**(*rgs*, *elements*)

Creates a set partition from a restricted growth string.

### Explanation

The indices given in rgs are assumed to be the index of the element as given in elements *as provided* (the elements are not sorted by this routine). Block numbering starts from 0. If any block was not referenced in `rgs` an error will be raised.

### Examples

```
>>> from sympy.combinatorics import Partition
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('abcde'))
Partition({c}, {a, d}, {b, e})
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('cbead'))
Partition({e}, {a, c}, {b, d})
>>> a = Partition([1, 4], [2], [3, 5])
>>> Partition.from_rgs(a.RGS, a.members)
Partition({2}, {1, 4}, {3, 5})
```

**property partition**

Return partition as a sorted list of lists.

### Examples

```
>>> from sympy.combinatorics import Partition
>>> Partition([1], [2, 3]).partition
[[1], [2, 3]]
```

**property rank**

Gets the rank of a partition.

### Examples

```
>>> from sympy.combinatorics import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.rank
13
```

**sort_key**(*order=None*)

Return a canonical key that can be used for sorting.

Ordering is based on the size and sorted elements of the partition and ties are broken with the rank.

### Examples

```
>>> from sympy import default_sort_key
>>> from sympy.combinatorics import Partition
>>> from sympy.abc import x
>>> a = Partition([1, 2])
>>> b = Partition([3, 4])
>>> c = Partition([1, x])
>>> d = Partition(list(range(4)))
>>> l = [d, b, a + 1, a, c]
>>> l.sort(key=default_sort_key); l
[Partition({1, 2}), Partition({1}, {2}), Partition({1, x}), Partition(
↪{3, 4}), Partition({0, 1, 2, 3})]
```

**class** sympy.combinatorics.partitions.**IntegerPartition**(*partition*, *integer=None*)

This class represents an integer partition.

### Explanation

In number theory and combinatorics, a partition of a positive integer, n, also called an integer partition, is a way of writing n as a list of positive integers that sum to n. Two partitions that differ only in the order of summands are considered to be the same partition; if order matters then the partitions are referred to as compositions. For example, 4 has five partitions: [4], [3, 1], [2, 2], [2, 1, 1], and [1, 1, 1, 1]; the compositions [1, 2, 1] and [1, 1, 2] are the same as partition [2, 1, 1].

**See also:**

*sympy.utilities.iterables.partitions* (page 2089), *sympy.utilities.iterables.multiset_partitions* (page 2084)

## References

[R50]

**as_dict**()

Return the partition as a dictionary whose keys are the partition integers and the values are the multiplicity of that integer.

### Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> IntegerPartition([1]*3 + [2] + [3]*4).as_dict()
{1: 3, 2: 1, 3: 4}
```

**as_ferrers**(*char='#'*)

Prints the ferrer diagram of a partition.

### Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> print(IntegerPartition([1, 1, 5]).as_ferrers())
#####
#
#
```

**property conjugate**

Computes the conjugate partition of itself.

### Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> a = IntegerPartition([6, 3, 3, 2, 1])
>>> a.conjugate
[5, 4, 3, 1, 1]
```

**next_lex**()

Return the next partition of the integer, n, in lexical order, wrapping around to [n] if the partition is [1, ..., 1].

### Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([3, 1])
>>> print(p.next_lex())
[4]
>>> p.partition < p.next_lex().partition
True
```

**prev_lex**()

Return the previous partition of the integer, n, in lexical order, wrapping around to [1, …, 1] if the partition is [n].

### Examples

```
>>> from sympy.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([4])
>>> print(p.prev_lex())
[3, 1]
>>> p.partition > p.prev_lex().partition
True
```

sympy.combinatorics.partitions.**random_integer_partition**(*n*, *seed=None*)

Generates a random integer partition summing to n as a list of reverse-sorted integers.

### Examples

```
>>> from sympy.combinatorics.partitions import random_integer_partition
```

For the following, a seed is given so a known value can be shown; in practice, the seed would not be given.

```
>>> random_integer_partition(100, seed=[1, 1, 12, 1, 2, 1, 85, 1])
[85, 12, 2, 1]
>>> random_integer_partition(10, seed=[1, 2, 3, 1, 5, 1])
[5, 3, 1, 1]
>>> random_integer_partition(1)
[1]
```

sympy.combinatorics.partitions.**RGS_generalized**(*m*)

Computes the m + 1 generalized unrestricted growth strings and returns them as rows in matrix.

### Examples

```
>>> from sympy.combinatorics.partitions import RGS_generalized
>>> RGS_generalized(6)
Matrix([
[  1,   1,   1,  1,  1, 1, 1],
[  1,   2,   3,  4,  5, 6, 0],
[  2,   5,  10, 17, 26, 0, 0],
[  5,  15,  37, 77,  0, 0, 0],
[ 15,  52, 151,  0,  0, 0, 0],
[ 52, 203,   0,  0,  0, 0, 0],
[203,   0,   0,  0,  0, 0, 0]])
```

sympy.combinatorics.partitions.**RGS_enum**(*m*)

RGS_enum computes the total number of restricted growth strings possible for a superset of size m.

### Examples

```
>>> from sympy.combinatorics.partitions import RGS_enum
>>> from sympy.combinatorics import Partition
>>> RGS_enum(4)
15
>>> RGS_enum(5)
52
>>> RGS_enum(6)
203
```

We can check that the enumeration is correct by actually generating the partitions. Here, the 15 partitions of 4 items are generated:

```
>>> a = Partition(list(range(4)))
>>> s = set()
>>> for i in range(20):
...     s.add(a)
...     a += 1
...
>>> assert len(s) == 15
```

sympy.combinatorics.partitions.**RGS_unrank**(*rank*, *m*)

Gives the unranked restricted growth string for a given superset size.

### Examples

```
>>> from sympy.combinatorics.partitions import RGS_unrank
>>> RGS_unrank(14, 4)
[0, 1, 2, 3]
>>> RGS_unrank(0, 4)
[0, 0, 0, 0]
```

sympy.combinatorics.partitions.**RGS_rank**(*rgs*)

Computes the rank of a restricted growth string.

### Examples

```
>>> from sympy.combinatorics.partitions import RGS_rank, RGS_unrank
>>> RGS_rank([0, 1, 2, 1, 3])
42
>>> RGS_rank(RGS_unrank(4, 7))
4
```

## Permutations

**class** sympy.combinatorics.permutations.**Permutation**(**args*, *size=None*, ***kwargs*)

A permutation, alternatively known as an 'arrangement number' or 'ordering' is an arrangement of the elements of an ordered list into a one-to-one mapping with itself. The permutation of a given arrangement is given by indicating the positions of the elements after re-arrangement [R69]. For example, if one started with elements `[x, y, a, b]` (in that order) and they were reordered as `[x, y, b, a]` then the permutation would be `[0, 1, 3, 2]`. Notice that (in SymPy) the first element is always referred to as 0 and the permutation uses the indices of the elements in the original ordering, not the elements `(a, b, ...)` themselves.

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
```

### Permutations Notation

Permutations are commonly represented in disjoint cycle or array forms.

### Array Notation And 2-line Form

In the 2-line form, the elements and their final positions are shown as a matrix with 2 rows:

[0 1 2 … n-1] [p(0) p(1) p(2) … p(n-1)]

Since the first line is always `range(n)`, where n is the size of p, it is sufficient to represent the permutation by the second line, referred to as the "array form" of the permutation. This is entered in brackets as the argument to the Permutation class:

```
>>> p = Permutation([0, 2, 1]); p
Permutation([0, 2, 1])
```

Given i in range(p.size), the permutation maps i to i^p

```
>>> [i^p for i in range(p.size)]
[0, 2, 1]
```

The composite of two permutations p*q means first apply p, then q, so i^(p*q) = (i^p)^q which is i^p^q according to Python precedence rules:

```
>>> q = Permutation([2, 1, 0])
>>> [i^p^q for i in range(3)]
[2, 0, 1]
>>> [i^(p*q) for i in range(3)]
[2, 0, 1]
```

One can use also the notation p(i) = i^p, but then the composition rule is (p*q)(i) = q(p(i)), not p(q(i)):

```
>>> [(p*q)(i) for i in range(p.size)]
[2, 0, 1]
>>> [q(p(i)) for i in range(p.size)]
[2, 0, 1]
>>> [p(q(i)) for i in range(p.size)]
[1, 2, 0]
```

### Disjoint Cycle Notation

In disjoint cycle notation, only the elements that have shifted are indicated.

For example, [1, 3, 2, 0] can be represented as (0, 1, 3)(2). This can be understood from the 2 line format of the given permutation. In the 2-line form, [0 1 2 3] [1 3 2 0]

The element in the 0th position is 1, so 0 -> 1. The element in the 1st position is three, so 1 -> 3. And the element in the third position is again 0, so 3 -> 0. Thus, 0 -> 1 -> 3 -> 0, and 2 -> 2. Thus, this can be represented as 2 cycles: (0, 1, 3)(2). In common notation, singular cycles are not explicitly written as they can be inferred implicitly.

Only the relative ordering of elements in a cycle matter:

```
>>> Permutation(1,2,3) == Permutation(2,3,1) == Permutation(3,1,2)
True
```

The disjoint cycle notation is convenient when representing permutations that have several cycles in them:

```
>>> Permutation(1, 2)(3, 5) == Permutation([[1, 2], [3, 5]])
True
```

It also provides some economy in entry when computing products of permutations that are written in disjoint cycle notation:

```
>>> Permutation(1, 2)(1, 3)(2, 3)
Permutation([0, 3, 2, 1])
>>> _ == Permutation([[1, 2]])*Permutation([[1, 3]])*Permutation([[2,␣
↪3]])
True
```

Caution: when the cycles have common elements between them then the order in which the permutations are applied matters. This module applies the permutations from *left to right*.

```
>>> Permutation(1, 2)(2, 3) == Permutation([(1, 2), (2, 3)])
True
>>> Permutation(1, 2)(2, 3).list()
[0, 3, 1, 2]
```

In the above case, (1,2) is computed before (2,3). As 0 -> 0, 0 -> 0, element in position 0 is 0. As 1 -> 2, 2 -> 3, element in position 1 is 3. As 2 -> 1, 1 -> 1, element in position 2 is 1. As 3 -> 3, 3 -> 2, element in position 3 is 2.

If the first and second elements had been swapped first, followed by the swapping of the second and third, the result would have been [0, 2, 3, 1]. If, you want to apply the cycles in the conventional right to left order, call the function with arguments in reverse order as demonstrated below:

```
>>> Permutation([(1, 2), (2, 3)][::-1]).list()
[0, 2, 3, 1]
```

Entering a singleton in a permutation is a way to indicate the size of the permutation. The `size` keyword can also be used.

Array-form entry:

```
>>> Permutation([[1, 2], [9]])
Permutation([0, 2, 1], size=10)
>>> Permutation([[1, 2]], size=10)
Permutation([0, 2, 1], size=10)
```

Cyclic-form entry:

```
>>> Permutation(1, 2, size=10)
Permutation([0, 2, 1], size=10)
>>> Permutation(9)(1, 2)
Permutation([0, 2, 1], size=10)
```

Caution: no singleton containing an element larger than the largest in any previous cycle can be entered. This is an important difference in how Permutation and Cycle handle the `__call__` syntax. A singleton argument at the start of a Permutation performs instantiation of the Permutation and is permitted:

```
>>> Permutation(5)
Permutation([], size=6)
```

A singleton entered after instantiation is a call to the permutation – a function call – and if the argument is out of range it will trigger an error. For this reason, it is better to start the cycle with the singleton:

The following fails because there is no element 3:

```
>>> Permutation(1, 2)(3)
Traceback (most recent call last):
...
IndexError: list index out of range
```

This is ok: only the call to an out of range singleton is prohibited; otherwise the permutation autosizes:

```
>>> Permutation(3)(1, 2)
Permutation([0, 2, 1, 3])
>>> Permutation(1, 2)(3, 4) == Permutation(3, 4)(1, 2)
True
```

### Equality Testing

The array forms must be the same in order for permutations to be equal:

```
>>> Permutation([1, 0, 2, 3]) == Permutation([1, 0])
False
```

### Identity Permutation

The identity permutation is a permutation in which no element is out of place. It can be entered in a variety of ways. All the following create an identity permutation of size 4:

```
>>> I = Permutation([0, 1, 2, 3])
>>> all(p == I for p in [
... Permutation(3),
... Permutation(range(4)),
... Permutation([], size=4),
... Permutation(size=4)])
True
```

Watch out for entering the range *inside* a set of brackets (which is cycle notation):

```
>>> I == Permutation([range(4)])
False
```

### Permutation Printing

There are a few things to note about how Permutations are printed.

Deprecated since version 1.6: Configuring Permutation printing by setting `Permutation.print_cyclic` is deprecated. Users should use the `perm_cyclic` flag to the printers, as described below.

1) If you prefer one form (array or cycle) over another, you can set `init_printing` with the `perm_cyclic` flag.

```
>>> from sympy import init_printing
>>> p = Permutation(1, 2)(4, 5)(3, 4)
>>> p
Permutation([0, 2, 1, 4, 5, 3])
```

```
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> p
(1 2)(3 4 5)
```

2) Regardless of the setting, a list of elements in the array for cyclic form can be obtained and either of those can be copied and supplied as the argument to Permutation:

```
>>> p.array_form
[0, 2, 1, 4, 5, 3]
>>> p.cyclic_form
[[1, 2], [3, 4, 5]]
>>> Permutation(_) == p
True
```

3) Printing is economical in that as little as possible is printed while retaining all information about the size of the permutation:

```
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation([1, 0, 2, 3])
Permutation([1, 0, 2, 3])
>>> Permutation([1, 0, 2, 3], size=20)
Permutation([1, 0], size=20)
>>> Permutation([1, 0, 2, 4, 3, 5, 6], size=20)
Permutation([1, 0, 2, 4, 3], size=20)
```

```
>>> p = Permutation([1, 0, 2, 3])
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> p
(3)(0 1)
>>> init_printing(perm_cyclic=False, pretty_print=False)
```

The 2 was not printed but it is still there as can be seen with the array_form and size methods:

```
>>> p.array_form
[1, 0, 2, 3]
>>> p.size
4
```

### Short Introduction To Other Methods

The permutation can act as a bijective function, telling what element is located at a given position

```
>>> q = Permutation([5, 2, 3, 4, 1, 0])
>>> q.array_form[1] # the hard way
2
>>> q(1) # the easy way
2
>>> {i: q(i) for i in range(q.size)} # showing the bijection
{0: 5, 1: 2, 2: 3, 3: 4, 4: 1, 5: 0}
```

The full cyclic form (including singletons) can be obtained:

```
>>> p.full_cyclic_form
[[0, 1], [2], [3]]
```

Any permutation can be factored into transpositions of pairs of elements:

```
>>> Permutation([[1, 2], [3, 4, 5]]).transpositions()
[(1, 2), (3, 5), (3, 4)]
>>> Permutation.rmul(*[Permutation([ti], size=6) for ti in _]).cyclic_
↪form
[[1, 2], [3, 4, 5]]
```

The number of permutations on a set of n elements is given by n! and is called the cardinality.

```
>>> p.size
4
>>> p.cardinality
24
```

A given permutation has a rank among all the possible permutations of the same elements, but what that rank is depends on how the permutations are enumerated. (There are a number of different methods of doing so.) The lexicographic rank is given by the rank method and this rank is used to increment a permutation with addition/subtraction:

```
>>> p.rank()
6
>>> p + 1
Permutation([1, 0, 3, 2])
>>> p.next_lex()
Permutation([1, 0, 3, 2])
>>> _.rank()
7
>>> p.unrank_lex(p.size, rank=7)
Permutation([1, 0, 3, 2])
```

The product of two permutations p and q is defined as their composition as functions, (p*q)(i) = q(p(i)) [R73].

```
>>> p = Permutation([1, 0, 2, 3])
>>> q = Permutation([2, 3, 1, 0])
>>> list(q*p)
[2, 3, 0, 1]
>>> list(p*q)
[3, 2, 1, 0]
>>> [q(p(i)) for i in range(p.size)]
[3, 2, 1, 0]
```

The permutation can be ‘applied’ to any list-like object, not only Permutations:

```
>>> p(['zero', 'one', 'four', 'two'])
['one', 'zero', 'four', 'two']
>>> p('zo42')
['o', 'z', '4', '2']
```

If you have a list of arbitrary elements, the corresponding permutation can be found with the from_sequence method:

```
>>> Permutation.from_sequence('SymPy')
Permutation([1, 3, 2, 0, 4])
```

### Checking If A Permutation Is Contained In A Group

Generally if you have a group of permutations G on n symbols, and you’re checking if a permutation on less than n symbols is part of that group, the check will fail.

Here is an example for n=5 and we check if the cycle (1,2,3) is in G:

```
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=True, pretty_print=False)
>>> from sympy.combinatorics import Cycle, Permutation
>>> from sympy.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup(Cycle(2, 3)(4, 5), Cycle(1, 2, 3, 4, 5))
>>> p1 = Permutation(Cycle(2, 5, 3))
>>> p2 = Permutation(Cycle(1, 2, 3))
>>> a1 = Permutation(Cycle(1, 2, 3).list(6))
>>> a2 = Permutation(Cycle(1, 2, 3)(5))
>>> a3 = Permutation(Cycle(1, 2, 3),size=6)
>>> for p in [p1,p2,a1,a2,a3]: p, G.contains(p)
((2 5 3), True)
((1 2 3), False)
((5)(1 2 3), True)
((5)(1 2 3), True)
((5)(1 2 3), True)
```

The check for p2 above will fail.

Checking if p1 is in G works because SymPy knows G is a group on 5 symbols, and p1 is also on 5 symbols (its largest element is 5).

For `a1`, the `.list(6)` call will extend the permutation to 5 symbols, so the test will work as well. In the case of `a2` the permutation is being extended to 5 symbols by using a singleton, and in the case of `a3` it’s extended through the constructor argument `size=6`.

There is another way to do this, which is to tell the contains method that the number of symbols the group is on does not need to match perfectly the number of symbols for the permutation:

```
>>> G.contains(p2,strict=False)
True
```

This can be via the strict argument to the contains method, and SymPy will try to extend the permutation on its own and then perform the containment check.

**See also:**

*Cycle* (page 284)

## References

[R68], [R69], [R70], [R71], [R72], [R73], [R74]

**apply**(*i*)

Apply the permutation to an expression.

**Parameters**

**i** : Expr

It should be an integer between $0$ and $n-1$ where $n$ is the size of the permutation.

If it is a symbol or a symbolic expression that can have integer values, an AppliedPermutation object will be returned which can represent an unevaluated function.

## Notes

Any permutation can be defined as a bijective function $\sigma : \{0, 1, \ldots, n-1\} \rightarrow \{0, 1, \ldots, n-1\}$ where $n$ denotes the size of the permutation.

The definition may even be extended for any set with distinctive elements, such that the permutation can even be applied for real numbers or such, however, it is not implemented for now for computational reasons and the integrity with the group theory module.

This function is similar to the __call__ magic, however, __call__ magic already has some other applications like permuting an array or attatching new cycles, which would not always be mathematically consistent.

This also guarantees that the return type is a SymPy integer, which guarantees the safety to use assumptions.

**property array_form**

Return a copy of the attribute _array_form Examples ========

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([[2, 0], [3, 1]])
>>> p.array_form
[2, 3, 0, 1]
>>> Permutation([[2, 0, 3, 1]]).array_form
```

(continues on next page)

(continued from previous page)

```
[3, 2, 0, 1]
>>> Permutation([2, 0, 3, 1]).array_form
[2, 0, 3, 1]
>>> Permutation([[1, 2], [4, 5]]).array_form
[0, 2, 1, 3, 5, 4]
```

**ascents**()

Returns the positions of ascents in a permutation, ie, the location where p[i] < p[i+1]

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.ascents()
[1, 2]
```

**See also:**

*descents* (page 267), *inversions* (page 272), *min* (page 276), *max* (page 275)

**atoms**()

Returns all the elements of a permutation

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2, 3, 4, 5]).atoms()
{0, 1, 2, 3, 4, 5}
>>> Permutation([[0, 1], [2, 3], [4, 5]]).atoms()
{0, 1, 2, 3, 4, 5}
```

**property cardinality**

Returns the number of all possible permutations.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.cardinality
24
```

**See also:**

*length* (page 275), *order* (page 277), *rank* (page 279), *size* (page 282)

**commutator**(*x*)

Return the commutator of `self` and x: `~x*~self*x*self`

If f and g are part of a group, G, then the commutator of f and g is the group identity iff f and g commute, i.e. fg == gf.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation([0, 2, 3, 1])
>>> x = Permutation([2, 0, 3, 1])
>>> c = p.commutator(x); c
Permutation([2, 1, 3, 0])
>>> c == ~x*~p*x*p
True
```

```
>>> I = Permutation(3)
>>> p = [I + i for i in range(6)]
>>> for i in range(len(p)):
...     for j in range(len(p)):
...         c = p[i].commutator(p[j])
...         if p[i]*p[j] == p[j]*p[i]:
...             assert c == I
...         else:
...             assert c != I
...
```

### References

[R75]

**commutes_with**(*other*)

Checks if the elements are commuting.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> a = Permutation([1, 4, 3, 0, 2, 5])
>>> b = Permutation([0, 1, 2, 3, 4, 5])
>>> a.commutes_with(b)
True
>>> b = Permutation([2, 3, 5, 4, 1, 0])
>>> a.commutes_with(b)
False
```

**property cycle_structure**

Return the cycle structure of the permutation as a dictionary indicating the multiplicity of each cycle length.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation(3).cycle_structure
{1: 4}
>>> Permutation(0, 4, 3)(1, 2)(5, 6).cycle_structure
{2: 2, 3: 1}
```

**property cycles**

Returns the number of cycles contained in the permutation (including singletons).

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 1, 2]).cycles
3
>>> Permutation([0, 1, 2]).full_cyclic_form
[[0], [1], [2]]
>>> Permutation(0, 1)(2, 3).cycles
2
```

**See also:**

*sympy.functions.combinatorial.numbers.stirling* (page 444)

**property cyclic_form**

This is used to convert to the cyclic notation from the canonical notation. Singletons are omitted.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 3, 1, 2])
>>> p.cyclic_form
[[1, 3, 2]]
>>> Permutation([1, 0, 2, 4, 3, 5]).cyclic_form
[[0, 1], [3, 4]]
```

**See also:**

*array_form* (page 264), *full_cyclic_form* (page 268)

**descents()**

Returns the positions of descents in a permutation, ie, the location where p[i] > p[i+1]

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.descents()
[0, 3]
```

**See also:**

*ascents* (page 265), *inversions* (page 272), *min* (page 276), *max* (page 275)

**classmethod from_inversion_vector**(*inversion*)

Calculates the permutation from the inversion vector.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> Permutation.from_inversion_vector([3, 2, 1, 0, 0])
Permutation([3, 2, 1, 0, 4, 5])
```

**classmethod from_sequence**(*i*, *key=None*)

Return the permutation needed to obtain i from the sorted elements of i. If custom sorting is desired, a key can be given.

### Examples

```
>>> from sympy.combinatorics import Permutation
```

```
>>> Permutation.from_sequence('SymPy')
(4)(0 1 3)
>>> _(sorted("SymPy"))
['S', 'y', 'm', 'P', 'y']
>>> Permutation.from_sequence('SymPy', key=lambda x: x.lower())
(4)(0 2)(1 3)
```

**property full_cyclic_form**

Return permutation in cyclic form including singletons.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 2, 1]).full_cyclic_form
[[0], [1, 2]]
```

**get_adjacency_distance**(*other*)

Computes the adjacency distance between two permutations.

### Explanation

This metric counts the number of times a pair i,j of jobs is adjacent in both p and p'. If n_adj is this quantity then the adjacency distance is n - n_adj - 1 [1]

[1] Reeves, Colin R. Landscapes, Operators and Heuristic search, Annals of Operational Research, 86, pp 473-490. (1999)

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> p.get_adjacency_distance(q)
3
>>> r = Permutation([0, 2, 1, 4, 3])
>>> p.get_adjacency_distance(r)
4
```

**See also:**

*get_precedence_matrix* (page 270), *get_precedence_distance* (page 270), *get_adjacency_matrix* (page 269)

**get_adjacency_matrix**()

Computes the adjacency matrix of a permutation.

### Explanation

If job i is adjacent to job j in a permutation p then we set m[i, j] = 1 where m is the adjacency matrix of p.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation.josephus(3, 6, 1)
>>> p.get_adjacency_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1],
[0, 1, 0, 0, 0, 0],
[1, 0, 0, 0, 0, 0],
[0, 0, 0, 1, 0, 0]])
>>> q = Permutation([0, 1, 2, 3])
>>> q.get_adjacency_matrix()
Matrix([
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1],
[0, 0, 0, 0]])
```

**See also:**

*get_precedence_matrix* (page 270), *get_precedence_distance* (page 270), *get_adjacency_distance* (page 268)

**get_positional_distance**(*other*)

Computes the positional distance between two permutations.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> r = Permutation([3, 1, 4, 0, 2])
>>> p.get_positional_distance(q)
12
>>> p.get_positional_distance(r)
12
```

**See also:**

*get_precedence_distance* (page 270), *get_adjacency_distance* (page 268)

**get_precedence_distance**(*other*)

Computes the precedence distance between two permutations.

### Explanation

Suppose p and p' represent n jobs. The precedence metric counts the number of times a job j is preceded by job i in both p and p'. This metric is commutative.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([2, 0, 4, 3, 1])
>>> q = Permutation([3, 1, 2, 4, 0])
>>> p.get_precedence_distance(q)
7
>>> q.get_precedence_distance(p)
7
```

**See also:**

*get_precedence_matrix* (page 270), *get_adjacency_matrix* (page 269), *get_adjacency_distance* (page 268)

**get_precedence_matrix**()

Gets the precedence matrix. This is used for computing the distance between two permutations.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> from sympy import init_printing
>>> init_printing(perm_cyclic=False, pretty_print=False)
>>> p = Permutation.josephus(3, 6, 1)
>>> p
Permutation([2, 5, 3, 1, 4, 0])
>>> p.get_precedence_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[1, 0, 0, 0, 1, 0],
[1, 1, 0, 1, 1, 1],
[1, 1, 0, 0, 1, 0],
[1, 0, 0, 0, 0, 0],
[1, 1, 0, 1, 1, 0]])
```

**See also:**

*get_precedence_distance* (page 270), *get_adjacency_matrix* (page 269), *get_adjacency_distance* (page 268)

**index**()

Returns the index of a permutation.

The index of a permutation is the sum of all subscripts j such that p[j] is greater than p[j+1].

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([3, 0, 2, 1, 4])
>>> p.index()
2
```

**inversion_vector**()

Return the inversion vector of the permutation.

The inversion vector consists of elements whose value indicates the number of elements in the permutation that are lesser than it and lie on its right hand side.

The inversion vector is the same as the Lehmer encoding of a permutation.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([4, 8, 0, 7, 1, 5, 3, 6, 2])
>>> p.inversion_vector()
[4, 7, 0, 5, 0, 2, 1, 1]
>>> p = Permutation([3, 2, 1, 0])
>>> p.inversion_vector()
[3, 2, 1]
```

The inversion vector increases lexicographically with the rank of the permutation, the -ith element cycling through 0..i.

```
>>> p = Permutation(2)
>>> while p:
...     print('%s %s %s' % (p, p.inversion_vector(), p.rank()))
...     p = p.next_lex()
(2) [0, 0] 0
(1 2) [0, 1] 1
(2)(0 1) [1, 0] 2
(0 1 2) [1, 1] 3
(0 2 1) [2, 0] 4
(0 2) [2, 1] 5
```

**See also:**

*from_inversion_vector* (page 268)

**inversions**()

Computes the number of inversions of a permutation.

### Explanation

An inversion is where i > j but p[i] < p[j].

For small length of p, it iterates over all i and j values and calculates the number of inversions. For large length of p, it uses a variation of merge sort to calculate the number of inversions.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3, 4, 5])
>>> p.inversions()
0
>>> Permutation([3, 2, 1, 0]).inversions()
6
```

**See also:**

*descents* (page 267), *ascents* (page 265), *min* (page 276), *max* (page 275)

### References

[R76]

**property is_Empty**

Checks to see if the permutation is a set with zero elements

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([]).is_Empty
True
>>> Permutation([0]).is_Empty
False
```

**See also:**

*is_Singleton* (page 273)

**property is_Identity**

Returns True if the Permutation is an identity permutation.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([])
>>> p.is_Identity
True
>>> p = Permutation([[0], [1], [2]])
>>> p.is_Identity
True
>>> p = Permutation([0, 1, 2])
>>> p.is_Identity
True
>>> p = Permutation([0, 2, 1])
>>> p.is_Identity
False
```

**See also:**

*order* (page 277)

**property is_Singleton**

Checks to see if the permutation contains only one number and is thus the only possible permutation of this set of numbers

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0]).is_Singleton
True
>>> Permutation([0, 1]).is_Singleton
False
```

**See also:**

*is_Empty* (page 272)

**property is_even**

Checks if a permutation is even.

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_even
True
>>> p = Permutation([3, 2, 1, 0])
>>> p.is_even
True
```

**See also:**

*is_odd* (page 274)

**property is_odd**

Checks if a permutation is odd.

## Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_odd
False
>>> p = Permutation([3, 2, 0, 1])
>>> p.is_odd
True
```

**See also:**

*is_even* (page 273)

**classmethod josephus**(*m*, *n*, *s=1*)

Return as a permutation the shuffling of range(n) using the Josephus scheme in which every m-th item is selected until all have been chosen. The returned permutation has elements listed by the order in which they were selected.

The parameter s stops the selection process when there are s items remaining and these are selected by continuing the selection, counting by 1 rather than by m.

Consider selecting every 3rd item from 6 until only 2 remain:

```
choices    chosen
========   ======
  012345
  01 345   2
  01 34    25
  01  4    253
  0   4    2531
  0        25314
           253140
```

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation.josephus(3, 6, 2).array_form
[2, 5, 3, 1, 4, 0]
```

### References

[R77], [R78], [R79]

**length**()

Returns the number of integers moved by a permutation.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> Permutation([0, 3, 2, 1]).length()
2
>>> Permutation([[0, 1], [2, 3]]).length()
4
```

**See also:**

*min* (page 276), *max* (page 275), *support* (page 282), *cardinality* (page 265), *order* (page 277), *rank* (page 279), *size* (page 282)

**list**(*size=None*)

Return the permutation as an explicit list, possibly trimming unmoved elements if size is less than the maximum element in the permutation; if this is desired, setting `size=-1` will guarantee such trimming.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Permutation(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
>>> Permutation(3).list(-1)
[]
```

**max**()

The maximum element moved by the permutation.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([1, 0, 2, 3, 4])
>>> p.max()
1
```

**See also:**

*min* (page 276), *descents* (page 267), *ascents* (page 265), *inversions* (page 272)

**min**()

The minimum element moved by the permutation.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([0, 1, 4, 3, 2])
>>> p.min()
2
```

**See also:**

*max* (page 275), *descents* (page 267), *ascents* (page 265), *inversions* (page 272)

**mul_inv**(*other*)

other*~self, self and other have _array_form

**next_lex**()

Returns the next permutation in lexicographical order. If self is the last permutation in lexicographical order it returns None. See [4] section 2.4.

### Examples

```
>>> from sympy.combinatorics import Permutation
>>> p = Permutation([2, 3, 1, 0])
>>> p = Permutation([2, 3, 1, 0]); p.rank()
17
>>> p = p.next_lex(); p.rank()
18
```

**See also:**

*rank* (page 279), *unrank_lex* (page 283)

**next_nonlex**()

Returns the next permutation in nonlex order [3]. If self is the last permutation in this order it returns None.