permutation

Release 0.1.0

2017 Oct 02

Contents

permutation provides a *Permutation* class for representing permutations of finitely many positive integers in Python. Supported operations & properties include inverses, (group theoretic) order, parity, composition/multiplication, cycle decomposition, cycle notation, word representation, Lehmer codes, and, of course, use as a callable on integers.

CHAPTER 1

Installation

`permutation` is written in pure Python with no dependencies. Just use pip (You have pip, right?) to install:

```
pip install permutation
```

CHAPTER 2

Examples

```
>>> from permutation import Permutation
>>> p = Permutation(2, 1, 4, 5, 3)
>>> p.to_cycles()
[(1, 2), (3, 4, 5)]
>>> print(p)
(1 2)(3 4 5)
>>> print(p.inverse())
(1 2)(3 5 4)
>>> p.degree
5
>>> p.order
6
>>> p.is_even
False
>>> p.lehmer(5)
27
>>> q = Permutation.cycle(1,2,3)
>>> print(p * q)
(2 4 5 3)
>>> print(q * p)
(1 3 4 5)
>>> for p in Permutation.group(3):
...     print(p)
...
1
(1 2)
(2 3)
(1 3 2)
(1 2 3)
(1 3)
```

CHAPTER 3

API

class `permutation.`**`Permutation`**`(`**img*`)`

A `Permutation` object represents a permutation of finitely many positive integers, i.e., a bijective function from some integer range $[1, n]$ to itself.

The arguments to the constructor are the elements of the permutation's word representation, i.e., the images of the integers 1 through some n under the permutation. For example, `Permutation(5, 4, 3, 6, 1, 2)` is the permutation that maps 1 to 5, 2 to 4, 3 to itself, 4 to 6, 5 to 1, and 6 to 2. `Permutation()` (with no arguments) evaluates to the identity permutation (i.e., the permutation that returns all inputs unchanged).

`Permutation`s are hashable and immutable. They can be compared for equality but not for ordering/sorting.

`__bool__``()`

A `Permutation` is true iff it is not the identity

`__call__``(`*i*`)`

Map an integer through the permutation. Values less than 1 are returned unchanged.

Parameters **`i`** (`int`) –

Returns the image of `i` under the permutation

`__mul__``(`*other*`)`

Multiplication/composition of permutations. `p * q` returns a `Permutation` `r` such that `r(x) == p(q(x))` for all integers `x`.

Parameters **`other`** (`Permutation`) –

Return type *Permutation*

`__nonzero__``()`

A `Permutation` is true iff it is not the identity

`__str__``()`

Convert a `Permutation` to cycle notation. The instance is decomposed into cycles with `to_cycles()`, each cycle is written as a parenthesized space-separated sequence of integers, and the cycles are concatenated.

`str(Permutation())` is `"1"`.

This is the inverse of *parse*.

```
>>> str(Permutation(2, 5, 4, 3, 1))
'(1 2 5)(3 4)'
```

classmethod cycle(**cyc*)

Construct a cyclic permutation from a sequence of unique positive integers. If `p = Permutation.cycle(*cyc)`, then `p(cyc[0]) == cyc[1]`, `p(cyc[1]) == cyc[2]`, etc., and `p(cyc[-1]) == cyc[0]`, with `p` returning all other values unchanged.

`Permutation.cycle()` (with no arguments) evaluates to the identity permutation.

Parameters **cyc** – zero or more unique positive integers

Returns the permutation represented by the given cycle

Raises **ValueError** –

- if `cyc` contains a value less than 1
- if `cyc` contains the same value more than once

degree

The degree of the permutation, i.e., the largest integer that it permutes (does not map to itself), or 0 if there is no such integer (i.e., if the permutation is the identity)

classmethod from_cycles(**cycles*)

Construct a *Permutation* from zero or more cyclic permutations. Each element of `cycles` is converted to a *Permutation* with *cycle*, and the results (which need not be disjoint) are multiplied together. `Permutation.from_cycles()` (with no arguments) evaluates to the identity permutation.

This is the inverse of *to_cycles*.

Parameters **cycles** – zero or more iterables of unique positive integers

Returns the *Permutation* represented by the product of the cycles

Raises **ValueError** –

- if any cycle contains a value less than 1
- if any cycle contains the same value more than once

classmethod from_left_lehmer(*x*)

Returns the permutation with the given left Lehmer code. This is the inverse of *left_lehmer()*.

Parameters **x** (*int*) – a nonnegative integer

Returns the *Permutation* with left Lehmer code `x`

Raises **ValueError** – if `x` is less than 0

classmethod from_lehmer(*x, n*)

Calculate the permutation in S_n with Lehmer code `x`. This is the permutation at index `x` (zero-based) in the list of all permutations of degree at most `n` ordered lexicographically by word representation.

This is the inverse of *lehmer*.

Parameters

- **x** (*int*) – a nonnegative integer
- **n** (*int*) – the degree of the symmetric group with respect to which `x` was calculated

Returns the *Permutation* with Lehmer code `x`

Raises **ValueError** – if `x` is less than 0 or greater than or equal to the factorial of `n`

classmethod **`group`**(*n*)
Generates all permutations in S_n, the symmetric group of degree `n`, i.e., all permutations with degree less than or equal to `n`. The permutations are yielded in ascending order of their *left Lehmer codes*.

Parameters **n** (*`int`*) – a nonnegative integer

Returns a generator of all *`Permutation`*s with degree `n` or less

Raises **`ValueError`** – if `n` is less than 0

`inverse`()
Returns the inverse of the permutation, i.e., the unique permutation that, when multiplied by the invocant on either the left or the right, produces the identity

Return type *Permutation*

`is_even`
Whether the permutation is even, i.e., can be expressed as the product of an even number of transpositions (cycles of length 2)

`is_odd`
Whether the permutation is odd, i.e., not even

`isdisjoint`(*other*)
Returns `True` iff the permutation and `other` are disjoint, i.e., iff they do not permute any of the same integers

Parameters **other** (`Permutation`) – a permutation to compare against

Return type bool

`left_lehmer`()
Encode the permutation as a nonnegative integer using a modified form of Lehmer codes that uses the left inversion count instead of the right inversion count. This modified encoding establishes a degree-independent bijection between permutations and nonnegative integers, with *`from_left_lehmer()`* converting values in the opposite direction.

Returns the permutation's left Lehmer code

Return type int

`lehmer`(*n*)
Calculate the Lehmer code of the permutation with respect to all permutations of degree at most `n`. This is the (zero-based) index of the permutation in the list of all permutations of degree at most `n` ordered lexicographically by word representation.

This is the inverse of *`from_lehmer`*.

Parameters **n** (*`int`*) –

Return type int

Raises **`ValueError`** – if `n` is less than *`degree`*

`next_permutation`()
Returns the next *`Permutation`* in *left Lehmer code* order

`order`
The order (a.k.a. period) of the permutation, i.e., the smallest positive integer n such that multiplying n copies of the permutation together produces the identity

classmethod **`parse`**(*s*)
Parse a permutation written in cycle notation. This is the inverse of *`__str__`*.

Parameters **s** (*`str`*) – a permutation written in cycle notation

Returns the permutation represented by `s`

Return type *Permutation*

Raises **ValueError** – if `s` is not valid cycle notation for a permutation

permute(*xs*)

Reorder the elements of a sequence according to the permutation; each element at index `i` is moved to index `p(i)`.

Note that `p.permute(range(1, n+1)) == p.inverse().to_image(n)` for all integers `n` greater than or equal to `degree`.

Parameters **xs** – a sequence of at least `degree` elements

Returns a permuted sequence

Return type tuple

Raises **ValueError** – if `len(xs)` is less than `degree`

prev_permutation()

Returns the previous `Permutation` in *left Lehmer code* order

Raises **ValueError** – if called on the identity `Permutation` (which has no predecessor)

sign

The sign (a.k.a. signature) of the permutation: 1 if the permutation is even, -1 if it is odd

to_cycles()

Decompose the permutation into a product of disjoint cycles. `to_cycles()` returns a list of cycles in which each cycle is a tuple of integers. Each cycle `c` is a sub-permutation that maps `c[0]` to `c[1]`, `c[1]` to `c[2]`, etc., finally mapping `c[-1]` back around to `c[0]`. The permutation is then the product of these cycles.

Each cycle is at least two elements in length and places its smallest element first. Cycles are ordered by their first elements in increasing order. No two cycles share an element.

When the permutation is the identity, `to_cycles()` returns an empty list.

This is the inverse of `from_cycles`.

Returns the cycle decomposition of the permutation

to_image(*n=None*)

Returns a tuple of the results of applying the permutation to the integers 1 through `n`, or through `degree` if `n` is unspecified. If `v = p.to_image()`, then `v[0] == p(1)`, `v[1] == p(2)`, etc.

When the permutation is the identity, `to_image` called without an argument returns an empty tuple.

This is the inverse of the constructor.

Parameters **n** (`int`) – the length of the image to return; defaults to `degree`

Returns the image of 1 through `n` under the permutation

Return type tuple of ints

Raises **ValueError** – if `n` is less than `degree`

CHAPTER 4

Indices and tables

- genindex
- search

Python Module Index

p

Index