permutation

Release 0.5.0.dev1

John T. Wodder II

2024 May 02

CONTENTS

GitHub | PyPI | Documentation | Issues | Changelog

`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 ONE

INSTALLATION

`permutation` requires Python 3.8 or higher. Just use pip for Python 3 (You have pip, right?) to install:

```
python3 -m pip install permutation
```

CHAPTER

TWO

EXAMPLES

```
>>> from permutation import Permutation
>>> p = Permutation(2, 1, 4, 5, 3)
>>> p(1)
2
>>> p(3)
4
>>> p(42)
42
>>> 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

THREE

API

class permutation.**Permutation**(**img: int*)

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

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

Construction:

__init__(*img)	Construct a permutation from a word representation.
parse(s)	Parse a permutation written in cycle notation.
cycle(*cyc)	Construct a cyclic permutation.
from_cycles(*cycles)	Construct the product of cyclic permutations.
from_lehmer(x, n)	Calculate the permutation in S_n with Lehmer code `x`.
from_left_lehmer(x)	Calculate the permutation with the given left Lehmer code.
group(n)	Generates all permutations in S_n.
next_permutation()	Returns the next *Permutation* in *left Lehmer code* order
prev_permutation()	Returns the previous *Permutation* in *left Lehmer code* order

Operations:

__call__(i)	Map an integer through the permutation.
__mul__(other)	Multiplication/composition of permutations.
__pow__(n)	Power/repeated composition of permutations.
permute(xs)	Returns the elements of `xs` reordered according to the permutation.

Properties:

__str__()	Convert a *Permutation* to cycle notation.
__bool__()	A *Permutation* is true iff it is not the identity.
inverse()	Returns the inverse of the permutation.
degree	The degree of the permutation.
order	The order/period of the permutation.
is_even	Whether the permutation is even.
is_odd	Whether the permutation is odd, i.e., not even
sign	The sign/signature of the permutation.
isdisjoint(other)	Tests whether the permutation is disjoint from `other`.
to_image([n])	Returns the images of 1 through `n` under the permutation.
to_cycles()	Decompose the permutation into a product of disjoint cycles.
right_inversion_count([n])	Calculate the right inversion count through degree `n`.
inversions()	Calculate the inversion number of the permutation.
lehmer(n)	Calculate a Lehmer code for the permutation.
left_lehmer()	Calculate the "left Lehmer code" for the permutation.

__bool__() → bool

A *Permutation* is true iff it is not the identity.

__call__(*i: int*) → int

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

Parameters
`i` (*int*)

Returns
the image of `i` under the permutation

__init__(**img: int*) → None

Construct a permutation from a word representation. The arguments are the images of the integers 1 through some n under the permutation to construct.

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).

__mul__(*other: Permutation*) → *Permutation*

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

__pow__(*n: int*) → *Permutation*

Power/repeated composition of permutations.

- `p ** 0 == Permutation()`
- `p ** n == p ** (n - 1) * p`
- `p ** -n == p.inverse() ** n`

Parameters
n (*int*) – exponent

Return type
Permutation

__str__() → 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: int*) → *Permutation*

Construct a cyclic permutation. 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 distinct 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

property degree: int

The degree of the permutation. This is 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: Iterable[int]*) → *Permutation*

Construct the product of 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 distinct 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: int*) → *Permutation*

Calculate 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: int*, *n: int*) → *Permutation*

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: int*) → Iterator[*Permutation*]

Generates all permutations in S_n. This is 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`() → *Permutation*

Returns the inverse of the permutation. This is the unique permutation that, when multiplied by the invocant on either the left or the right, produces the identity.

Return type
Permutation

inversions() → int

Calculate the inversion number of the permutation. This is the number of pairs of numbers which are in the opposite order after applying the permutation. This is also the Kendall tau distance from the identity permutation. This is also the sum of the terms in the Lehmer code when in factorial base.

Added in version 0.2.0.

Returns
the number of inversions in the permutation

Return type
int

property is_even: bool

Whether the permutation is even. That is, whether it can be expressed as the product of an even number of transpositions (cycles of length 2).

property is_odd: bool

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

isdisjoint(*other:* Permutation) → bool

Tests whether the permutation is disjoint from `other`. This returns `True` iff the two permutations do not permute any of the same integers.

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

Return type
bool

left_lehmer() → int

Calculate the "left Lehmer code" for the permutation. This uses 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: int*) → int

Calculate a Lehmer code for the permutation. The Lehmer code is computed with respect to all permutations of degree at most `n` and evaluates to the zero-based index of the permutation in the list of all such permutations when 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() → *Permutation*

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

property order: int

The order/period of the permutation. This is the smallest positive integer n such that multiplying n copies of the permutation together produces the identity

classmethod parse(*s: str*) → *Permutation*

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: Iterable[T]*) → list[T]

Returns the elements of `xs` reordered 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`.

Changed in version 0.5.0: This method now accepts iterables of any element type and returns a list. (Previously, it only accepted iterables of `int`s and returned a tuple.)

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

Returns
a permuted sequence

Return type
list

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

prev_permutation() → *Permutation*

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

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

right_inversion_count(*n: int* | *None* = *None*) → list[int]

Calculate the right inversion count through degree `n`. The result is a list of `n` elements in which the element at index `i` corresponds to the number of right inversions for `i+1`, i.e., the number of values `x > i+1` for which `p(x) < p(i+1)`.

Setting `n` larger than `degree` causes the resulting list to have trailing zeroes, which become relevant when converting to & from Lehmer codes and factorial base.

Added in version 0.2.0.

Parameters
`n` (`Optional[int]`) – defaults to `degree`

Return type
list[int]

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

property sign: int

The sign/signature of the permutation. This is 1 if the permutation is even, -1 if it is odd.

to_cycles() → list[tuple[int, ...]]

Decompose the permutation into a product of disjoint cycles. `to_cycles()` returns a list of cycles, each one represented as 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 product of these cycles is then the original permutation.

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: int* | *None* = *None*) → tuple[int, ...]

Returns the images of 1 through `n` under the permutation. 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[int, . . .]

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

CHAPTER FOUR

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

p

INDEX