

Contents

1	Classes	2
1.1	permute – permutation (symmetric) group	2
1.1.1	Permute – element of permutation group	3
1.1.1.1	setKey – change key	5
1.1.1.2	getValue – get “value”	5
1.1.1.3	getGroup – get PermGroup	5
1.1.1.4	numbering – give the index	5
1.1.1.5	order – order of the element	5
1.1.1.6	ToTranspose – represent as transpositions	6
1.1.1.7	ToCyclic – corresponding ExPermute element	6
1.1.1.8	sgn – sign of the permutation	6
1.1.1.9	types – type of cyclic representation	6
1.1.1.10	ToMatrix – permutation matrix	7
1.1.2	ExPermute – element of permutation group as cyclic representation	8
1.1.2.1	setKey – change key	10
1.1.2.2	getValue – get “value”	10
1.1.2.3	getGroup – get PermGroup	10
1.1.2.4	order – order of the element	10
1.1.2.5	ToNormal – represent as normal style	10
1.1.2.6	simplify – use simple value	11
1.1.2.7	sgn – sign of the permutation	11
1.1.3	PermGroup – permutation group	12
1.1.3.1	createElement – create an element from seed	13
1.1.3.2	identity – group identity	13
1.1.3.3	identity_c – group identity as cyclic type	13
1.1.3.4	grouporder – order as group	13
1.1.3.5	randElement – random permute element	13

Chapter 1

Classes

1.1 permute – permutation (symmetric) group

- Classes
 - **Permute**
 - **ExPermute**
 - **PermGroup**

1.1.1.1 Permute – element of permutation group

Initialize (Constructor)

Permute(value: *list/tuple*, key: *list/tuple*) → **Permute**

Permute(val_key: *dict*) → **Permute**

Permute(value: *list/tuple*, key: *int*=None) → **Permute**

Create an element of a permutation group.

An instance will be generated with “normal” way. That is, we input a **key**, which is a list of (indexed) all elements from some set, and a **value**, which is a list of all permuted elements.

Normally, you input two lists (or tuples) **value** and **key** with same length. Or you can input **val_key** as a dict whose **values()** is a list “value” and **keys()** is a list “key” in the sense of above. Also, there are some short-cut for inputting **key**:

- If key is $[1, 2, \dots, N]$, you do not have to input **key**.
- If key is $[0, 1, \dots, N - 1]$, input 0 as **key**.
- If key equals the list arranged through **value** in ascending order, input 1.
- If key equals the list arranged through **value** in descending order, input -1.

Attributes

key :
It expresses **key**.

data :
†It expresses indexed form of **value**.

Operations

operator	explanation
A==B	Check equality for A's value and B's one and A's key and B's one.
A*B	right multiplication (that is, $A \circ B$ with normal mapping way)
A/B	division (that is, $A \circ B^{-1}$)
A**B	powering
A.inverse()	inverse
A[c]	the element of value corresponding to c in key
A(1st)	permute 1st with A

Examples

```

>>> p1 = permute.Permute(['b','c','d','a','e'], ['a','b','c','d','e'])
>>> print p1
['a', 'b', 'c', 'd', 'e'] -> ['b', 'c', 'd', 'a', 'e']
>>> p2 = permute.Permute([2, 3, 0, 1, 4], 0)
>>> print p2
[0, 1, 2, 3, 4] -> [2, 3, 0, 1, 4]
>>> p3 = permute.Permute(['c','a','b','e','d'], 1)
>>> print p3
['a', 'b', 'c', 'd', 'e'] -> ['c', 'a', 'b', 'e', 'd']
>>> print p1 * p3
['a', 'b', 'c', 'd', 'e'] -> ['d', 'b', 'c', 'e', 'a']
>>> print p3 * p1
['a', 'b', 'c', 'd', 'e'] -> ['a', 'b', 'e', 'c', 'd']
>>> print p1 ** 4
['a', 'b', 'c', 'd', 'e'] -> ['a', 'b', 'c', 'd', 'e']
>>> p1['d']
'a'
>>> p2([0, 1, 2, 3, 4])
[2, 3, 0, 1, 4]

```

Methods

1.1.1.1 setKey – change key

setKey(self, key: *list/tuple*) → *Permute*

Set other key.

key must be list or tuple with same length to **key**.

1.1.1.2 getValue – get “value”

getValue(self) → *list*

Return (not data) value of self.

1.1.1.3 getGroup – get PermGroup

getGroup(self) → *PermGroup*

Return **PermGroup** to which self belongs.

1.1.1.4 numbering – give the index

numbering(self) → *int*

Number self in the permutation group. (Slow method)

The numbering is made to fit the following inductive definition for dimension of the permutation group.

If numbering of $[\sigma_1, \sigma_2, \dots, \sigma_{n-2}, \sigma_{n-1}]$ on $(n-1)$ -dimension is k , numbering of $[\sigma_1, \sigma_2, \dots, \sigma_{n-2}, \sigma_{n-1}, n]$ on n -dimension is k and numbering of $[\sigma_1, \sigma_2, \dots, \sigma_{n-2}, n, \sigma_{n-1}]$ on n -dimension is $k + (n-1)!$, and so on. (See [Room of Points And Lines, part 2, section 15, paragraph 2 \(Japanese\)](#))

1.1.1.5 order – order of the element

order(self) → *int/long*

Return order as the element of group.

This method is faster than general group method.

1.1.1.6 ToTranspose – represent as transpositions

ToTranspose(self) → ExPermute

Represent **self** as a composition of transpositions.

Return the element of **ExPermute** with transpose (2-dimensional cyclic) type. It is recursive program, and it would take more time than the method **ToCyclic**.

1.1.1.7 ToCyclic – corresponding ExPermute element

ToCyclic(self) → ExPermute

Represent **self** as a composition of cyclic representations.

Return the element of **ExPermute**. †This method decomposes **self** into coprime cyclic permutations, so each cyclic is commutative.

1.1.1.8 sgn – sign of the permutation

sgn(self) → int

Return the sign of permutation group element.

If **self** is even permutation, that is, **self** can be written as a composition of an even number of transpositions, it returns 1. Otherwise, that is, for odd permutation, it returns -1.

1.1.1.9 types – type of cyclic representation

types(self) → list

Return cyclic type defined by each cyclic permutation element length.

1.1.1.10 ToMatrix – permutation matrix

ToMatrix(self) → Matrix

Return permutation matrix.

The row and column correspond to **key**. If **self** G satisfies $G[a] = b$, then (a, b) -element of the matrix is 1. Otherwise, the element is 0.

Examples

```
>>> p = Permute([2,3,1,5,4])
>>> p.numbering()
28
>>> p.order()
6
>>> p.ToTranspose()
[(4,5)(1,3)(1,2)](5)
>>> p.sgn()
-1
>>> p.ToCyclic()
[(1,2,3)(4,5)](5)
>>> p.types()
'(2,3)type'
>>> print p.ToMatrix()
0 1 0 0 0
0 0 1 0 0
1 0 0 0 0
0 0 0 0 1
0 0 0 1 0
```

1.1.2 ExPermute – element of permutation group as cyclic representation

Initialize (Constructor)

ExPermute(dim: *int*, value: *list*, key: *list*=None) → ExPermute

Create an element of a permutation group.

An instance will be generated with “cyclic” way. That is, we input a **value**, which is a list of tuples and each tuple expresses a cyclic permutation. For example, $(\sigma_1, \sigma_2, \sigma_3, \dots, \sigma_k)$ is one-to-one mapping, $\sigma_1 \mapsto \sigma_2, \sigma_2 \mapsto \sigma_3, \dots, \sigma_k \mapsto \sigma_1$.

dim must be positive integer, that is, an instance of `int`, `long` or **Integer**. **key** should be a list whose length equals **dim**. Input a list of tuples whose elements are in **key** as **value**. Note that you can abbreviate **key** if **key** has the form $[1, 2, \dots, N]$. Also, you can input 0 as **key** if **key** has the form $[0, 1, \dots, N-1]$.

Attributes

dim :
It expresses **dim**.

key :
It expresses **key**.

data :
†It expresses indexed form of **value**.

Operations

operator	explanation
A==B	Check equality for A’s value and B’s one and A’s key and B’s one.
A*B	right multiplication (that is, $A \circ B$ with normal mapping way)
A/B	division (that is, $A \circ B^{-1}$)
A**B	powering
A.inverse()	inverse
A[c]	the element of value corresponding to c in key
A(1st)	permute 1st with A
str(A)	simple representation. use simplify .
repr(A)	representation

Examples

```
>>> p1 = permute.ExPermute(5, [('a', 'b')], ['a','b','c','d','e'])
>>> print p1
[('a', 'b')] <['a', 'b', 'c', 'd', 'e']>
>>> p2 = permute.ExPermute(5, [(0, 2), (3, 4, 1)], 0)
>>> print p2
[(0, 2), (1, 3, 4)] <[0, 1, 2, 3, 4]>
>>> p3 = permute.ExPermute(5, [('b', 'c')], ['a','b','c','d','e'])
>>> print p1 * p3
[('a', 'b'), ('b', 'c')] <['a', 'b', 'c', 'd', 'e']>
>>> print p3 * p1
[('b', 'c'), ('a', 'b')] <['a', 'b', 'c', 'd', 'e']>
>>> p1['c']
'c'
>>> p2([0, 1, 2, 3, 4])
[2, 4, 0, 1, 3]
```

Methods

1.1.2.1 `setKey` – change key

`setKey(self, key: list) → ExPermute`

Set other key.

`key` must be a list whose length equals **dim**.

1.1.2.2 `getValue` – get “value”

`getValue(self) → list`

Return (not data) value of `self`.

1.1.2.3 `getGroup` – get `PermGroup`

`getGroup(self) → PermGroup`

Return **PermGroup** to which `self` belongs.

1.1.2.4 `order` – order of the element

`order(self) → int/long`

Return order as the element of group.

This method is faster than general group method.

1.1.2.5 `ToNormal` – represent as normal style

`ToNormal(self) → Permute`

Represent `self` as an instance of **Permute**.

1.1.2.6 `simplify` – use simple value

`simplify(self) → ExPermute`

Return the more simple cyclic element.

†This method uses **ToNormal** and **ToCyclic**.

1.1.2.7 `sgn` – sign of the permutation

`sgn(self) → int`

Return the sign of permutation group element.

If `self` is even permutation, that is, `self` can be written as a composition of an even number of transpositions, it returns 1. Otherwise, that is, for odd permutation, it returns -1 .

Examples

```
>>> p = permute.ExPermute(5, [(1, 2, 3), (4, 5)])
>>> p.order()
6
>>> print p.ToNormal()
[1, 2, 3, 4, 5] -> [2, 3, 1, 5, 4]
>>> p * p
[(1, 2, 3), (4, 5), (1, 2, 3), (4, 5)] <[1, 2, 3, 4, 5]>
>>> (p * p).simplify()
[(1, 3, 2)] <[1, 2, 3, 4, 5]>
```

1.1.3 PermGroup – permutation group

Initialize (Constructor)

PermGroup(key: *int/long*) → PermGroup

PermGroup(key: *list/tuple*) → PermGroup

Create a permutation group.

Normally, input list as **key**. If you input some integer N , **key** is set as $[1, 2, \dots, N]$.

Attributes

key :
It expresses **key**.

Operations

operator	explanation
A==B	Check equality for A's value and B's one and A's key and B's one.
card(A)	same as grouporder
str(A)	simple representation
repr(A)	representation

Examples

```
>>> p1 = permute.PermGroup(['a','b','c','d','e'])
>>> print p1
['a','b','c','d','e']
>>> card(p1)
120L
```

Methods

1.1.3.1 createElement – create an element from seed

`createElement(self, seed: list/tuple/dict) → Permute`

`createElement(self, seed: list) → ExPermute`

Create new element in `self`.

`seed` must be the form of “value” on **Permute** or **ExPermute**

1.1.3.2 identity – group identity

`identity(self) → Permute`

Return the identity of `self` as normal type.

For cyclic type, use **identity_c**.

1.1.3.3 identity_c – group identity as cyclic type

`identity_c(self) → ExPermute`

Return permutation group identity as cyclic type.

For normal type, use **identity**.

1.1.3.4 grouporder – order as group

`grouporder(self) → int/long`

Compute the order of `self` as group.

1.1.3.5 randElement – random permute element

`randElement(self) → Permute`

Create random new element as normal type in `self`.

Examples

```
>>> p = permute.PermGroup(5)
>>> print p.createElement([3, 4, 5, 1, 2])
[1, 2, 3, 4, 5] -> [3, 4, 5, 1, 2]
>>> print p.createElement([(1, 2), (3, 4)])
[(1, 2), (3, 4)] <[1, 2, 3, 4, 5]>
>>> print p.identity()
[1, 2, 3, 4, 5] -> [1, 2, 3, 4, 5]
>>> print p.identity_c()
[] <[1, 2, 3, 4, 5]>
>>> p.grouporder()
120L
>>> print p.randElement()
[1, 2, 3, 4, 5] -> [3, 4, 5, 2, 1]
```