Combi Documentation

Release 0.1.1

Ram Rachum

January 15, 2015

Contents

Combi is a Pythonic package for combinatorics.

Combi lets you explore spaces of permutations and combinations as if they were Python sequences, but without generating all the permutations/combinations in advance. It lets you specify a lot of special conditions on these spaces. It also provides a few more classes that might be useful in combinatorics programming.

CHAPTER 1

Documentation contents

1.1 Getting started with Combi

Combi is a Pythonic package for combinatorics.

Combi lets you explore spaces of permutations and combinations as if they were Python sequences, but without generating all the permutations/combinations in advance. It lets you specify a lot of special conditions on these spaces. It also provides a few more classes that might be useful in combinatorics programming.

Let's look at the simplest example of using Combi. Check out this $5 padlock in the picture:

I use this padlock for my gym locker, so people won't steal my stuff when I'm swimming in the pool. It has 8 buttons, and to open it you have to press down a secret combination of 4 buttons. I wonder though, how easy is it to crack?

```
>>> from combi import *
>>> padlock_space = CombSpace(range(1, 9), 4)
>>> padlock_space
<CombSpace: range(1, 9), n_elements=4>
```

`padlock_space` is the space of all possible combinations for our padlock. At this point, the combinations weren't really generated; if we'll ask for a combination from the space, it'll be generated on-demand:

```
>>> padlock_space[7]
<Comb, n_elements=4: (1, 2, 4, 7)>
```

As you can see, `padlock_space` behaves like a sequence. We can get a combination by index number. We can also do other sequence-y things, like getting the index number of a combination, or slicing it, or getting the length using `len()`. This is a huge benefit because then we can explore these spaces in a declarative rather than imperative style of programming. (i.e. we don't have to think about generating the permutations, we simply assume that the permutation space exists and we're taking items from it at leisure.) Let's try looking at the length of `padlock_space`:

```
>>> len(padlock_space)
70
```

Only 70 combinations. That's pretty lame... At 3 seconds to try a combination, this means this padlock is crackable in under 4 minutes. Not very secure.

In the example above, I used `CombSpace`, which is a space of combinations. It's a thin subclass over `PermSpace`, which is a space of permutations. A combination is like a permutation, except order doesn't matter.

Now, because the permutations/combinations are generated on-demand, I can do something like this:

```
>>> huge_perm_space = PermSpace(1000)
>>> huge_perm_space
<PermSpace: 0..999>
```

This is a perm space of all permutations of the numbers between 0 and 999. It is ginormous. The number of permutations is around 10**2500 (a number that far exceeds the number of particles in the universe.) I'm not even going to show its length in the shell session because the length number alone would fill the entire page. And yet you can fetch any permutation from this space by index number in a fraction of a second:

```
>>> huge_perm_space[7]
<Perm: (0, 1, 2, 3, 4, ... 997, 996, 999, 998)>
```

Note that the permutation `huge_perm_space[7]` is a sequence by itself, where every item is a number in `range(1000)`.

Combi lets you specify a myriad of options on the the spaces that you create. For example, you can make some elements be fixed:

```
>>> fixed_perm_space = PermSpace(4, fixed_map={3: 3,})
>>> fixed_perm_space
<PermSpace: 0..3, fixed_map={3: 3}>
>>> tuple(fixed_perm_space)
(<Perm: (0, 1, 2, 3)>,
 <Perm: (0, 2, 1, 3)>,
 <Perm: (1, 0, 2, 3)>,
 <Perm: (1, 2, 0, 3)>,
 <Perm: (2, 0, 1, 3)>,
 <Perm: (2, 1, 0, 3)>)
```

This limits the space and makes it smaller. This is useful when you're making explorations on a huge `PermSpace` and want to inspect only a smaller subset of it that would be easier to handle.

There are many more variations that you could have on a `PermSpace` or a `CombSpace`. You can specify a custom domain and a custom range to a space. You can constrain it to permutations of a certain degree (e.g. `degrees=1` to limit to transformations only.) You can do k-permutations by specifying the length of the desired permutations as `n_elements`. You can have the permutation objects be of a custom subclass that you define, so you could provide extra methods on them that fit your use case. You can provide sequences that have some items appear multiple times and Combi would be smart about it and consider multiple occurrences of the same item to be interchangable. You can also toggle that behavior so it would treat them as unique. It's very customizable :) For more information on doing that, please refer to the documentation for `PermSpace` and `CombSpace`.

`PermSpace` and `CombSpace` are the flagship classes of Combi; but it also provides a few more useful features. See *features* and *documentation contents* for more info.

1.2 `PermSpace` and `Perm`

1.2.1 `PermSpace`

class `PermSpace`(*iterable_or_length*, *n_elements=None*, ***, *domain=None*, *fixed_map=None*, *degrees=None*, *is_combination=False*, *slice_=None*, *perm_type=None*)

A space of permutations on a sequence.

Each item in a PermSpace is a Perm, i.e. a permutation. This is similar to itertools.permutations(), except it offers far, far more functionality. The permutations may be accessed by index number, the permutation space can have its range and domain specified, some items can be fixed, and more.

Here is the simplest possible PermSpace:

```
>>> perm_space = PermSpace(3)
<PermSpace: 0..2>
>>> perm_space[2]
<Perm: (1, 0, 2)>
>>> tuple(perm_space)
(<Perm: (0, 1, 2)>, <Perm: (0, 2, 1)>, <Perm: (1, 0, 2)>,
 <Perm: (1, 2, 0)>, <Perm: (2, 0, 1)>, <Perm: (2, 1, 0)>)
```

The members are Perm objects, which are sequence-like objects that have extra functionality. (See documentation of Perm for more info.)

The permutations are generated on-demand, not in advance. This means you can easily create something like PermSpace(1000), which has about 10**2500 permutations in it (a number that far exceeds the number of particles in the universe), in a fraction of a second. You can then fetch by index number any permutation of the 10**2500 permutations in a fraction of a second as well.

PermSpace allows the creation of various special kinds of permutation spaces. For example, you can specify an integer to n_elements to set a permutation length that's smaller than the sequence length. (a.k.a. k-permutations.) This variation of a PermSpace is called "partial" and it's one of 8 different variations, that are listed below.

- **Rapplied (Range-applied):** Having an arbitrary sequence as a range. To make one, pass your sequence as the first argument instead of the length.
- **Dapplied (Domain-applied):** Having an arbitrary sequence as a domain. To make one, pass a sequence into the *domain* argument.
- **Recurrent:** If you provide a sequence (making the space rapplied) and that sequence has repeating items, you've made a recurrent *PermSpace*. It'll be shorter because all of the copies of same item will be considered the same item. (Though they will appear more than once, according to their count in the sequence.)
- **Fixed:** Having a specified number of indices always pointing at certain values, making the space smaller. To make one, pass a dict from each key to the value it should be fixed to as the argument fixed_map, like PermSpace('meow', fixed_map={0: 'm', 1: 'w'})
- **Sliced:** A perm space can be sliced like any Python sequence (except you can't change the step.) To make one, use slice notation on an existing perm space, e.g. perm_space[56:100].
- **Degreed:** A perm space can be limited to perms of a certain degree. (A perm's degree is the number of transformations it takes to make it.) To make one, pass into the degrees argument either a single degree (like 5) or a tuple of different degrees (like (1, 3, 7))
- **Partial:** A perm space can be partial, in which case not all elements are used in perms. E.g. you can have a perm space of a sequence of length 5 but with n_elements=3, so every perm will have only 3 items. (These are usually called k-permutations in math-land.) To make one, pass a number as the argument n_elements.
- **Combination:** If you pass in is_combination=True or use the subclass *CombSpace*, then you'll have a space of combinations (Combs) instead of perms. Combs are like Perms except there's no order to the elements. (They are always forced into canonical order.)
- **Typed:** If you pass in a perm subclass as perm_type, you'll get a typed PermSpace, meaning that the perms will use the class you provide rather than the default Perm. This is useful when you want to provide extra functionality on top of Perm that's specific to your use case.

Most of these variations can be used in conjuction with each other, but some cannot:

- A combination space can't be dapplied, degreed, or fixed.
- A partial permutation space can't be degreed.
- A recurrent permutation space must be rapplied and can't be degreed.

For each of these variations, there's a function to make a perm space have that variation and get rid of it. For example, if you want to make a normal perm space be degreed, call `get_degreed()` on it with the desired degrees. If you want to make a degreed perm space non-degreed, access its `undegreed` property. The same is true for all other variations.

A perm space that has none of these variations is called **pure**.

coerce_perm(*perm*)
Coerce a sequence to be a permutation of this space.

index(*perm*)
Get the index number of permutation `perm` in this space.

For example:

```
>>> perm_space = PermSpace(3)
>>> perm_space.index((2, 1, 0))
5
>>> perm_space[5]
<Perm: (2, 1, 0)>
```

length
The `PermSpace`'s length, i.e. the number of permutations in it.

This is also accessible as `len(perm_space)`, but since CPython doesn't support very large length numbers, it's best to access it through `perm_space.length`.

Methods and properties for adding a variation to a perm space: (A new perm space is returned, the original one does not get modified)

get_rapplied(*sequence*)
Get a rapplied version of this `PermSpace` that has a range of *sequence*.

get_dapplied(*domain*)
Get a dapplied version of this `PermSpace` that has a domain of *domain*.

(There's no `get_recurrented` method because we can't know which sequence you'd want. If you want a recurrent perm space you need to use `get_rapplied()` with a recurrent sequence.)

(There's no `get_sliced` method because slicing is done using Python's normal slice notation, e.g. `perm_space[4:-7]`.)

get_degreed(*degrees*)
Get a degreed version of this *PermSpace* restricted to certain degrees.

You may use a single integer for *degrees* to limit it to permutations of that degree, or a sequence of integers like `(1, 3, 7)`.

get_partialled(n_elements):
Get a partialled version of this `PermSpace`, i.e. a k-permutation.

combinationed:
A combination version of this perm space. (i.e. a `CombSpace`.)

get_typed(*perm_type*)
Get a typed version of this `PermSpace` where perms are of a custom type.

Properties for removing a variation from a perm space: (A new perm space is returned, the original one does not get modified)

purified
A purified version of this PermSpace, i.e. with all variations removed.

unrapplied
A version of this PermSpace without a custom range.

undapplied
A version of this PermSpace without a custom domain.

unrecurrented
A version of this PermSpace with no recurrences.

unfixed
An unfixed version of this PermSpace.

unsliced
An unsliced version of this PermSpace.

undegreed
An undegreed version of this PermSpace.

unpartialled
A non-partial version of this PermSpace.

uncombinationed
A version of this PermSpace where permutations have order.

untyped
An untyped version of this PermSpace.

1.2.2 Perm

class Perm(*perm_sequence*, *perm_space=None*)
A permutation of items from a PermSpace.

In combinatorics, a permutation is a sequence of items taken from the original sequence.

Example:

```
>>> perm_space = PermSpace('abcd')
>>> perm = Perm('dcba', perm_space)
>>> perm
<Perm: ('d', 'c', 'b', 'a')>
>>> perm_space.index(perm)
23
```

apply(*sequence*, *result_type=None*)
Apply the perm to a sequence, choosing items from it.

This can also be used as `sequence * perm`. Example:

```
>>> perm = PermSpace(5)[10]
>>> perm
<Perm: (0, 2, 4, 1, 3)>
>>> perm.apply('growl')
'golrw'
>>> 'growl' * perm
'golrw'
```

Specify `result_type` to determine the type of the result returned. If `result_type=None`, will use `tuple`, except when `other` is a `str` or `Perm`, in which case that same type would be used.

as_dictoid
A dict-like interface to a `Perm`.

classmethod coerce(*item*, *perm_space=None*)
Coerce item into a perm, optionally of a specified `PermSpace`.

degree
The permutation's degree, i.e. the number of transformations needed to produce it.

You can think of a permutation's degree like this: Imagine that you're starting with the identity permutation, and you want to make this permutation, by switching two items with each other over and over again until you get this permutation. The degree is the number of such switches you'll have to make.

domain
The permutation's domain.

For a non-dapplied permutation, this will be `range(len(sequence))`.

get_neighbors(*, *degrees=(1,)*, *perm_space=None*)
Get the neighbor permutations of this permutation.

This means, get the permutations that are close to this permutation. By default, this means permutations that are one transformation (switching a pair of items) away from this permutation. You can specify a custom sequence of integers to the `degrees` argument to get different degrees of relation. (e.g. specify `degrees=(1, 2)` to get both the closest neighbors and the second-closest neighbors.)

index(*member*)
Get the index number of `member` in the permutation.

Example:

```
>>> perm = PermSpace(5)[10]
>>> perm
<Perm: (0, 2, 4, 1, 3)>
>>> perm.index(3)
4
```

inverse
The inverse of this permutation, i.e. the permutation that we need to multiply this permutation by to get the identity permutation.

This is also accessible as `~perm`.

Example:

```
>>> perm = PermSpace(5)[10]
>>> perm
<Perm: (0, 2, 4, 1, 3)>
>>> ~perm
<Perm: (0, 3, 1, 4, 2)>
>>> perm * ~perm
<Perm: (0, 1, 2, 3, 4)>
```

items
A viewer of a perm's items, similar to `dict.items()`.

This is useful for dapplied perms; it lets you view the perm (both index access and iteration) as a sequence where each item is a 2-tuple, where the first item is from the domain and the second item is its corresponding item from the sequence.

length
The permutation's length.

n_cycles
The number of cycles in this permutation.

If item 1 points at item 7, and item 7 points at item 3, and item 3 points at item 1 again, then that's one cycle. `n_cycles` is the total number of cycles in this permutation.

unrapplied
A version of this permutation without a custom sequence. (Using `range(len(sequence))`.)

undapplied
A version of this permutation without a custom domain.

uncombinationed
A non-combination version of this permutation.

1.3 CombSpace and Comb

1.3.1 CombSpace

class CombSpace(*iterable_or_length*, *n_elements*, *, *slice_=None*, *perm_type=None*)
A space of combinations.

This is a subclass of `PermSpace`; see its documentation for more details.

Each item in a `CombSpace` is a `Comb`, i.e. a combination. This is similar to `itertools.combinations()`, except it offers far, far more functionality. The combinations may be accessed by index number, the combinations can be of a custom type, the space may be sliced, etc.

Here is the simplest possible `CombSpace`:

```
>>> comb_space = CombSpace(4, 2)
<CombSpace: 0..3, n_elements=2>
>>> comb_space[2]
<Comb, n_elements=2: (0, 3)>
>>> tuple(comb_space)
(<Comb, n_elements=2: (0, 1)>, <Comb, n_elements=2: (0, 2)>,
 <Comb, n_elements=2: (0, 3)>, <Comb, n_elements=2: (1, 2)>,
 <Comb, n_elements=2: (1, 3)>, <Comb, n_elements=2: (2, 3)>)
```

The members are `Comb` objects, which are sequence-like objects that have extra functionality. (See documentation of `Comb` and its base class `Perm` for more info.)

1.3.2 Comb

class Comb(*perm_sequence*, *perm_space=None*)
A combination of items from a `CombSpace`.

In combinatorics, a combination is like a permutation except with no order. In the `combi` package, we implement that by making the items in `Comb` be in canonical order. (This has the same effect as having no order because each combination of items can only appear once, in the canonical order, rather than many different times in many different orders like with `Perm`.)

Example:

```
>>> comb_space = CombSpace('abcde', 3)
>>> comb = Comb('bcd', comb_space)
>>> comb
<Comb, n_elements=3: ('a', 'b', 'c')>
>>> comb_space.index(comb)
6
```

1.4 Bags

> "This sort of thing is my bag, baby!"
>
> —Austin Powers

Combi provides 4 bag classes for you to choose from according to your needs:

- `Bag` - The simplest bag class
- `FrozenBag` - An immutable (and thus hashable) bag class
- `OrderedBag` - A bag class where items are ordered by insertion order
- `FrozenOrderedBag` - An immutable, ordered bag class

1.4.1 Bag

class Bag (*iterable={}*)

The `Bag` class is an implementation of the mathematical concept of a multiset; meaning something like a set, except that every item could appear multiple times, and crucially, we only save the *count* of the item in memory instead of saving multiple copies of the same item, which would be a waste of memory.

You may know the `collections.Counter` class from Python's standard library; the bag classes provided by Combi are very similar, except that they are more strictly defined as multisets, meaning that counts must be positive integers. (Zeros are allowed in input but they are weeded out.) By contrast, `collections.Counter` allows any value as an item's count.

This restriction makes the bag classes more powerful than `collections.Counter` because it allows more methods to be defined. More arithmetical operations are defined, comparison between bags is supported, and more.

When you create a bag, it will be populated with the `iterable` argument. If `iterable` is a plain sequence, its items will be added one-by-one:

```
>>> Bag('abracadabra')
Bag({'c': 1, 'b': 2, 'd': 1, 'a': 5, 'r': 2})
```

If `iterable` is a mapping, its values will be taken as item counts:

```
>>> Bag({'meow': 2, 'woof': 5,})
Bag({'meow': 2, 'woof': 5})
```

`Bag` can be accessed similarly to a `dict` or `Counter`:

```
>>> my_bag = Bag('abracadabra')
>>> my_bag['b']
2
>>> 'x' in my_bag
False
>>> my_bag['x'] = 7
```

```
>>> my_bag
Bag({'r': 2, 'x': 7, 'b': 2, 'a': 5, 'd': 1, 'c': 1})
>>> 'x' in my_bag
True
>>> my_bag['y'] # If it's not in the bag, the count is zero:
0
>>> for key, count in my_bag.items():
...     print((key, count))
...
('r', 2)
('x', 7)
('b', 2)
('a', 5)
('d', 1)
('c', 1)
```

elements
An iterator over the elements in the bag. If an item has a count of 7 in the bag, it will repeat 7 times in `bag.elements`. Example:

```
>>> bag = Bag({'a': 1, 'b': 2, 'c': 3,})
>>> for element in bag.elements:
...     print(element)
a
c
c
c
b
b
```

If you're using `OrderedBag` or `FrozenOrderedBag`, the items will be yielded by their canonical order (usually insertion order), otherwise they'll be yielded in arbitrary order, just like dicts and sets.

n_elements
The number of elements in the bag, i.e. the sum of all the counts. Example:

```
>>> bag = Bag({'a': 1, 'b': 2, 'c': 3,})
>>> bag.n_elements
6
```

clear()
Clear the bag, making it empty.

copy()
Create a shallow copy of the bag.

get(*key*, *default=None*)
Get a key's count with a default fallback, just like `dict.get()`.

keys()
Get an iterator over the keys in a bag:

```
>>> bag = Bag({'a': 1, 'b': 2, 'c': 3,})
>>> tuple(bag.keys())
('b', 'c', 'a')
```

values()
Get an iterator over the counts in a bag:

```
>>> bag = Bag({'a': 1, 'b': 2, 'c': 3,})
>>> tuple(bag.values())
(2, 3, 1)
```

items()
Get an iterator over the items in a bag, i.e. keys with counts:

```
>>> bag = Bag({'a': 1, 'b': 2, 'c': 3,})
>>> tuple(bag.items())
(('b', 2), ('c', 3), ('a', 1))
```

most_common([*n*])
Get a tuple of the `n` most common elements and their counts from the most common to the least. If n is not specified, `most_common()` returns all elements in the bag. Elements with equal counts are ordered arbitrarily:

```
>>> Bag('abracadabra').most_common(3)
(('a', 5), ('r', 2), ('b', 2))
```

pop(*key*[, *default*])
Get the count of `key` and remove it from the bag, optionally with a default fallback.

popitem()
Remove a key from the bag, and get a tuple `(key, count)`.

setdefault(*key*, *default=None*)
Get the count of *key*, optionally with a default fallback. If *key* is missing, its count in the bag will be set to the default.

update(*mapping*)
Update the bag with a mapping of key to count.

get_contained_bags()
Get all bags that are subsets of this bag.

This means all bags that have counts identical or smaller for each key.

Example:

```
>>> bag = Bag({'a': 1, 'b': 2, 'c': 3,})
>>> contained_bags = bag.get_contained_bags()
>>> len(contained_bags)
24
>>> contained_bags[7]
Bag({'c': 2, 'b': 1})
>>> contained_bags[7] < bag # Contained bag is contained.
True
```

1.4.2 FrozenBag

class FrozenBag(*iterable={}*)
`FrozenBag` is a multiset just like `Bag`, except it's immutable. After it's created, it can't be modified. It's a subclass over `Bag`.

In which cases would you want to use `FrozenBag` rather than `Bag`?

- If you want to have your bag as a key in a `set`, `dict`, or any other kind of hashtable. (A normal bag can't be used for this because it's mutable and thus not hashable.)
- If you want to make it clear in your program that a certain bag should never be changed after it's created.

get_mutable()
Get a mutable version of this frozen bag. Example:

```
>>> frozen_bag = FrozenBag('abracadabra')
>>> frozen_bag
FrozenBag({'d': 1, 'c': 1, 'b': 2, 'a': 5, 'r': 2})
>>> frozen_bag.get_mutable()
Bag({'d': 1, 'c': 1, 'b': 2, 'a': 5, 'r': 2})
```

1.4.3 OrderedBag

class OrderedBag(*iterable={}*)
OrderedBag is a multiset just like Bag, except it's also ordered. Items have a defined order, which is by default the order in which they were inserted. OrderedBag is a subclass over Bag.

Another way to think of it is that OrderedBag is to Bag what collections.OrderedDict is to dict.

Example:

```
>>> ordered_bag = OrderedBag('abbcccdddd')
>>> tuple(ordered_bag.elements) # Items ordered by insertion order:
('a', 'b', 'b', 'c', 'c', 'c', 'd', 'd', 'd', 'd')
>>> tuple(ordered_bag.items())
(('a', 1), ('b', 2), ('c', 3), ('d', 4))
>>> ordered_bag.index('b')
1
```

reversed
Get a version of this ordered bag with order reversed. Example:

```
>>> ordered_bag = OrderedBag('abbcccdddd')
>>> ordered_bag
OrderedBag(OrderedDict([('a', 1), ('b', 2), ('c', 3), ('d', 4)]))
>>> ordered_bag.reversed
OrderedBag(OrderedDict([('d', 4), ('c', 3), ('b', 2), ('a', 1)]))
```

This does *not* change the existing bag, but creates a new one.

index(*key*)
Get the index number (i.e. position) of key in the ordered bag.

move_to_end(*key*, *last=True*)
Move a key to the end of the order. Specify *last=False* to move it to the start instead.

sort(*key=None*, *reverse=False*)
Sort the keys, changing the order in-place.

The optional key argument, (not to be confused with the bag keys,) is a key function. If it's not passed in, default Python ordering will be used.

If reverse=True is used, the keys will be sorted in reverse order.

1.4.4 FrozenOrderedBag

class FrozenOrderedBag(*iterable={}*)
FrozenOrderedBag is a multiset just like Bag, except it's immutable *and* ordered. You can think of it as a combination of FrozenBag and OrderedBag.

1.4.5 Comparisons between bags

One of the advantages of `Bag` over `collections.Counter` is that `Bag` provides comparison methods between bags, that act similarly to comparisons between Python's built-in sets. This makes it easy to see whether one bag is contained by another. Example:

```
>>> sandwich = Bag({'bread': 2, 'cheese': 1, 'tomato': 2, 'burger': 1,})
>>> vegan_sandwich = Bag({'bread': 2, 'tomato': 2,})
>>> vegan_sandwich < sandwich
True
>>> salad = Bag({'tomato': 2, 'bell pepper': 1,})
>>> salad < sandwich # False because there's no bell pepper in our sandwich
False
```

A bag is smaller-or-equal to another bag if it's "contained" in it, meaning that every key in the contained bag also exists in the containing bag, and with a count that's bigger or equal to its count in the contained bag.

A bag is strictly smaller than another bag if the above holds, and there's at least one key for which the count in the containing bag is strictly bigger than its count in the contained bag.

Please note that unlike most comparisons in Python, this is a partial order rather than a total one, meaning that for some pairs of bags, neither `x >= y` nor `y >= x` holds true. This is similar to set comparison in Python.

1.4.6 Arithmetic operations on bags

Another advantage of `Bag` over `collections.Counter` is that `Bag` provides a *wealth* of arithmetic operations (addition, subtraction, etc.) between bags, and between bags and integers.

The basic arithmetic operations do what you expect them to, operating on the counts of items:

```
>>> sandwich = Bag({'bread': 2, 'cheese': 1, 'tomato': 2, 'burger': 1,})
>>> salad = Bag({'tomato': 2, 'bell pepper': 1,})
>>> single_tomato = Bag({'tomato': 1,})
>>> sandwich + single_tomato # Addition
Bag({'cheese': 1, 'bread': 2, 'tomato': 3, 'burger': 1})
>>> sandwich - single_tomato # Subtraction
Bag({'cheese': 1, 'bread': 2, 'tomato': 1, 'burger': 1})
>>> sandwich * 3 # Multiplication
Bag({'cheese': 3, 'bread': 6, 'tomato': 6, 'burger': 3})
```

As for division: You can divide one bag by another to get an integer saying how many times the second bag would go into the first. You can also divide a bag by an integer, which will divide all the counts by that integer, rounding down. All bag division is done with the floor-division operator `//`, because it's always rounded-down. Examples:

```
>>> sandwich // single_tomato # Floor-division by another bag
2
>>> sandwich // 2 # Floor-division by integer
Bag({'bread': 1, 'tomato': 1})
```

The more advanced operations are also supported:

```
>>> sandwich % salad # Modulo by bag
Bag({'bread': 2, 'cheese': 1, 'burger': 1, 'tomato': 2})
>>> divmod(sandwich, salad) # Divmod by bag
(0, Bag({'tomato': 2, 'cheese': 1, 'bread': 2, 'burger': 1}))
>>> salad % 2 # Modulo by integer
Bag({'bell pepper': 1})
>>> divmod(salad, 2) # Divmod by integer
(Bag({'tomato': 1}), Bag({'bell pepper': 1}))
```

```
>>> salad ** 3 # Exponentiation
Bag({'tomato': 8, 'bell pepper': 1})
>>> pow(salad, 3, 5) # Exponentiation with modulo
Bag({'tomato': 3, 'bell pepper': 1})
```

And... Did someone say logical operations? Like in Python sets, these do union and intersection:

```
>>> sandwich | salad # Logical or, a.k.a. set union
Bag({'bread': 2, 'cheese': 1, 'burger': 1, 'tomato': 2, 'bell pepper': 1})
>>> sandwich & salad # Logical and, a.k.a. set intersection
Bag({'tomato': 2})
```

As a final note about arithmetic operations, augmented assignment is supported for all operations, so you can elegantly mutate bags in-place like this:

```
>>> sandwich += Bag({'bacon': 2, 'egg': 1,})
>>> sandwich
Bag({'cheese': 1, 'bacon': 2, 'bread': 2, 'egg': 1, 'burger': 1, 'tomato': 2})
>>> sandwich **= 2
>>> sandwich
Bag({'cheese': 1, 'bacon': 4, 'bread': 4, 'egg': 1, 'burger': 1, 'tomato': 4})
```

1.5 Other classes

These are smaller, simpler classes that are included in `combi`.

1.5.1 MapSpace

class MapSpace (*function*, *sequence*)

A space of a function applied to a sequence.

This is similar to Python's built-in `map()`, except that it behaves like a sequence rather than an iterable. (Though it's also iterable.) You can access any item by its index number.

Example:

```
>>> map_space = MapSpace(lambda x: x ** 2, range(7))
>>> map_space
MapSpace(<function <lambda> at 0x00000000030C1510>, range(0, 7))
>>> len(map_space)
7
>>> map_space[3]
9
>>> tuple(map_space)
(0, 1, 4, 9, 16, 25, 36)
```

1.5.2 ProductSpace

class ProductSpace (*sequences*)

A product space between sequences.

This is similar to Python's `itertools.product()`, except that it behaves like a sequence rather than an iterable. (Though it's also iterable.) You can access any item by its index number.

Example:

```
>>> product_space = ProductSpace(('abc', range(4)))
>>> product_space
<ProductSpace: 3 * 4>
>>> product_space.length
12
>>> product_space[10]
('c', 2)
>>> tuple(product_space)
(('a', 0), ('a', 1), ('a', 2), ('a', 3), ('b', 0), ('b', 1), ('b', 2),
 ('b', 3), ('c', 0), ('c', 1), ('c', 2), ('c', 3))
```

index(*given_sequence*)
Get the index number of `given_sequence` in this product space.

1.5.3 ChainSpace

class ChainSpace(*sequences*)
A space of sequences chained together.

This is similar to Python's `itertools.chain()`, except that items can be fetched by index number rather than just iteration.

Example:

```
>>> chain_space = ChainSpace(('abc', (1, 2, 3)))
>>> chain_space
<ChainSpace: 3+3>
>>> chain_space[4]
2
>>> tuple(chain_space)
('a', 'b', 'c', 1, 2, 3)
>>> chain_space.index(2)
4
```

index(*item*)
Get the index number of `item` in this space.

1.5.4 SelectionSpace

class SelectionSpace(*sequence*)
Space of possible selections of any number of items from `sequence`.

For example:

```
>>> tuple(SelectionSpace(range(2)))
(set(), {1}, {0}, {0, 1})
```

The selections (which are sets) can be for any number of items, from zero to the length of the sequence.

Of course, this is a smart object that doesn't really create all these sets in advance, but rather on demand. So you can create a `SelectionSpace` like this:

```
>>> selection_space = SelectionSpace(range(10**4))
```

And take a random selection from it:

```
>>> selection_space.take_random()
{0, 3, 4, ..., 9996, 9997}
```

Even though the length of this space is around 10 ** 3010, which is much bigger than the number of particles in the universe.

index(*selection*)
Find the index number of set `selection` in this `SelectionSpace`.

CHAPTER 2

Basic usage

Use `PermSpace` to create a permutation space:

```
>>> from combi import *
>>> perm_space = PermSpace('meow')
```

It behaves like a sequence:

```
>>> len(perm_space)
24
>>> perm_space[7]
<Perm: ('e', 'm', 'w', 'o')>
>>> perm_space.index('mowe')
3
```

And yet the permutations are created on-demand rather than in advance.

Use `CombSpace` to create a combination space, where order doesn't matter:

```
>>> comb_space = CombSpace(('vanilla', 'chocolate', 'strawberry'), 2)
>>> comb_space
<CombSpace: ('vanilla', 'chocolate', 'strawberry'), n_elements=2>
>>> comb_space[2]
<Comb, n_elements=2: ('chocolate', 'strawberry')>
>>> len(comb_space)
3
```

For more details, *try the tutorial* or see the *documentation contents*.

CHAPTER 3

Features

- `PermSpace` lets you explore a space of permutations as if it was a Python sequence.
  - Permutations are generated on-demand, so huge permutation spaces can be created easily without big memory footprint.
  - `PermSpace` will notice if you have repeating elements in your sequence, and treat all occurences of the same value as interchangable rather than create redundant permutations.
  - A custom domain can be specified instead of just using index numbers.
  - You may specify some elements to be fixed, so they'll point to the same value in all permutations. (Useful for limiting an experiment to a subset of the original permutation space.)
  - Permutation spaces may be limited to a certain degree of permutations. (A permutation's degree is the number of transformations it takes to make it.)
  - k-permutations are supported.
  - You may specify a custom type for the generated permutations, so you could implement your own functionality on them.
- `CombSpace` lets you explore a space of combinations as if it was a Python sequence.
- `MapSpace` is like Python's built-in `map()`, except it's a sequence that allows index access.
- `ProductSpace` is like Python's `itertools.product()`, except it's a sequence that allows index access.
- `ChainSpace` is like Python's `itertools.chain()`, except it's a sequence that allows index access.
- `SelectionSpace` is a space of all selections from a sequence, of all possible lengths.
- The `Bag` class is a multiset like Python's `collections.Counter`, except it offers far more functionality, like more *arithmetic operations between bags*, *comparison between bags*, and more. (It can do that because unlike Python's `collections.Counter`, it only allows natural numbers as keys.)
- Classes `FrozenBag`, `OrderedBag` and `FrozenOrderedBag` are provided, which are variations on `Bag`.

CHAPTER 4

Requirements

- Python, version 2.7 or 3.3 or above. If you're new to Python, download the newest version from here.
- Setuptools.

CHAPTER 5

Installation

Use pip to install Combi:

```
$ pip install combi
```

CHAPTER 6

Community

Combi on GitHub: https://github.com/cool-RR/combi Feel free to fork and send pull requests!

There are three Combi groups, a.k.a. mailing lists:

- If you need help with Combi, post a message on the combi-users Google Group.
- If you want to help on the development of Combi itself, come say hello on the combi-dev Google Group.
- If you want to be informed on new releases of Combi, sign up for the low-traffic combi-announce Google Group.

CHAPTER 7

Roadmap

Combi is currently at a version 0.1.1. It's in a very early phase, and currently backward compatibility will not be maintained, to allow for freedom in changing the API. After more feedback and revisions to the API, backward compatibility will start being maintained.

Combi has an extensive test suite.

Changelog.

S

U

V