ChainMap — Search Multiple Dictionaries

The ChainMap class manages a sequence of dictionaries, and searches through them in the order they are given to find values associated with keys. A ChainMap makes a good “context” container, since it can be treated as a stack for which changes happen as the stack grows, with these changes being discarded again as the stack shrinks.

Accessing Values

The ChainMap supports the same API as a regular dictionary for accessing existing values.

```
# collections_chainmap_read.py

import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)

print('Individual Values')
print('a = {}'.format(m['a']))
print('b = {}'.format(m['b']))
print('c = {}'.format(m['c']))
print()

print('Keys = {}'.format(list(m.keys())))
print('Values = {}'.format(list(m.values())))
print()

print('Items:')
for k, v in m.items():
    print('{} = {}'.format(k, v))
print()

print('"d" in m: {}'.format(('d' in m)))
```

The child mappings are searched in the order they are passed to the constructor, so the value reported for the key 'c' comes from the a dictionary.

```
$ python3 collections_chainmap_read.py

Individual Values
a = A
b = B
c = C

Keys = ['b', 'c', 'a']
Values = ['B', 'C', 'A']

Items:
b = B
c = C
a = A

"d" in m: False
```

Reordering

The ChainMap stores the list of mappings over which it searches in a list in its maps attribute. This list is mutable, so it is possible to add new mappings directly or to change the order of the elements to control lookup and update behavior.

```
# collections_chainmap_reorder.py

import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)

print(m.maps)
print('c = {}\n'.format(m['c']))

# reverse the list
m.maps = list(reversed(m.maps))

print(m.maps)
print('c = {}'.format(m['c']))
```

When the list of mappings is reversed, the value associated with `'c'` changes.

```
$ python3 collections_chainmap_reorder.py

[{'a': 'A', 'c': 'C'}, {'b': 'B', 'c': 'D'}]
c = C

[{'b': 'B', 'c': 'D'}, {'a': 'A', 'c': 'C'}]
c = D
```

Updating Values

A `ChainMap` does not cache the values in the child mappings. Thus, if their contents are modified, the results are reflected when the `ChainMap` is accessed.

```
# collections_chainmap_update_behind.py

import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)
print('Before: {}'.format(m['c']))
a['c'] = 'E'
print('After : {}'.format(m['c']))
```

Changing the values associated with existing keys and adding new elements works the same way.

```
$ python3 collections_chainmap_update_behind.py

Before: C
After : E
```

It is also possible to set values through the `ChainMap` directly, although only the first mapping in the chain is actually modified.

```
# collections_chainmap_update_directly.py

import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m = collections.ChainMap(a, b)
print('Before:', m)
m['c'] = 'E'
print('After :', m)
print('a:', a)
```

When the new value is stored using m, the a mapping is updated.

```
$ python3 collections_chainmap_update_directly.py

Before: ChainMap({'a': 'A', 'c': 'C'}, {'b': 'B', 'c': 'D'})
After : ChainMap({'a': 'A', 'c': 'E'}, {'b': 'B', 'c': 'D'})
a: {'a': 'A', 'c': 'E'}
```

ChainMap provides a convenience method for creating a new instance with one extra mapping at the front of the maps list to make it easy to avoid modifying the existing underlying data structures.

```
# collections_chainmap_new_child.py

import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}

m1 = collections.ChainMap(a, b)
m2 = m1.new_child()

print('m1 before:', m1)
print('m2 before:', m2)

m2['c'] = 'E'

print('m1 after:', m1)
print('m2 after:', m2)
```

This stacking behavior is what makes it convenient to use ChainMap instances as template or application contexts. Specifically, it is easy to add or update values in one iteration, then discard the changes for the next iteration.

```
$ python3 collections_chainmap_new_child.py

m1 before: ChainMap({'a': 'A', 'c': 'C'}, {'b': 'B', 'c': 'D'})
m2 before: ChainMap({}, {'a': 'A', 'c': 'C'}, {'b': 'B', 'c':
'D'})
m1 after: ChainMap({'a': 'A', 'c': 'C'}, {'b': 'B', 'c': 'D'})
m2 after: ChainMap({'c': 'E'}, {'a': 'A', 'c': 'C'}, {'b': 'B',
'c': 'D'})
```

For situations where the new context is known or built in advance, it is also possible to pass a mapping to new_child().

```
# collections_chainmap_new_child_explicit.py

import collections

a = {'a': 'A', 'c': 'C'}
b = {'b': 'B', 'c': 'D'}
c = {'c': 'E'}

m1 = collections.ChainMap(a, b)
m2 = m1.new_child(c)

print('m1["c"] = {}'.format(m1['c']))
print('m2["c"] = {}'.format(m2['c']))
```

This is the equivalent of

```
m2 = collections.ChainMap(c, *m1.maps)
```

and produces

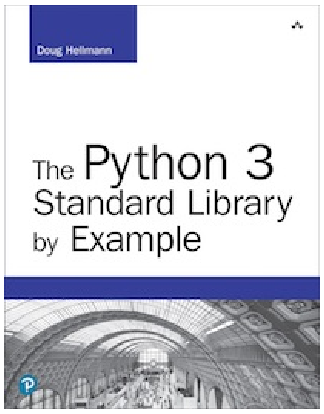
```
$ python3 collections_chainmap_new_child_explicit.py

m1["c"] = C
m2["c"] = E
```

Quick Links

This page was last updated 2018-12-09.

Navigation

collections — Container Data Types
Counter — Count Hashable Objects



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?