MIDict Documentation

Release 0.1

Shenggao Zhu

May 18, 2016

Contents

`MIDict` is an ordered "dictionary" with multiple indices where any index can serve as "keys" or "values", capable of assessing multiple values via its powerful indexing syntax, and suitable as a bidirectional/inverse dict (a drop-in replacement for dict/OrderedDict in Python 2 & 3).

CHAPTER 1

Features

- Multiple indices
- Multi-value indexing syntax
- Convenient indexing shortcuts
- Bidirectional/inverse dict
- Compatible with normal dict in Python 2 & 3
- Accessing keys via attributes
- Extended methods for multi-indices
- Additional APIs to handle indices
- Duplicate keys/values handling

CHAPTER 2

Quickstart

name	uid	ip
jack	1	192.1
tony	2	192.2

The above table-like data set (with multiple columns/indices) can be represented using a `MIDict`:

```
user = MIDict([['jack', 1, '192.1'], # list of items (rows of data)
               ['tony', 2, '192.2']],
              ['name', 'uid', 'ip']) # a list of index names
```

Access a key and get a value or a list of values (similar to a normal `dict`):

```
user['jack'] == [1, '192.1']
```

Any index (column) can be used as the "keys" or "values" via the advanced "multi-indexing" syntax `d[index_key:key, index_value]`. Both `index_key` and `index_value` can be a normal index name or an `int` (the order the index), and `index_value` can also be a `tuple`, `list` or `slice` object to specify multiple values, e.g.:

```
user['name':'jack', 'uid'] == 1
user['ip':'192.1', 'name'] == 'jack'

user['name':'jack', ('uid', 'ip')] == [1, '192.1']
user[0:'jack', [1, 2]]             == [1, '192.1']
user['name':'jack', 'uid':]        == [1, '192.1']
```

The "multi-indexing" syntax also has convenient shortcuts:

```
user['jack'] == [1, '192.1']
user[:'192.1'] == ['jack', 1]
user['jack', :] == ['jack', 1, '192.1']
```

A `MIDict` with 2 indices can be used as a bidirectional/inverse dict:

```
mi_dict = MIDict(jack=1, tony=2)

mi_dict['jack'] == 1 # forward indexing: d[key] -> value
mi_dict[:1]     == 'jack' # backward/inverse indexing: d[:value] -> key
```

CHAPTER 3

Documentation

See https://midict.readthedocs.io

CHAPTER 4

Installation

```
pip install midict
```

PyPI repository: https://pypi.python.org/pypi/midict

CHAPTER 5

Development

Source code: https://github.com/ShenggaoZhu/midict

Report issues: https://github.com/ShenggaoZhu/midict/issues/new

5.1 Testing

```
python tests/tests.py
```

Tested with both Python 2.7 and Python 3,3, 3.4, 3.5.

CHAPTER 6

Table of contents

6.1 MIDict Tutorial

MIDict is an ordered "dictionary" with multiple indices where any index can serve as "keys" or "values", capable of assessing multiple values via its powerful indexing syntax, and suitable as a bidirectional/inverse dict (a drop-in replacement for dict/OrderedDict in Python 2 & 3).

Features:

- Multiple indices
- Multi-value indexing syntax
- Convenient indexing shortcuts
- Bidirectional/inverse dict
- Compatible with normal dict in Python 2 & 3
- Accessing keys via attributes
- Extended methods for multi-indices
- Additional APIs to handle indices
- Duplicate keys/values handling

6.1.1 Multiple indices

Consider a table-like data set (e.g., a user table):

name	uid	ip
jack	1	192.1
tony	2	192.2

In each index (i.e., column), elements are unique and hashable (suitable for dict keys). Here, a "super dict" is wanted to represent this table which allows any index (column) to be used as the "keys" to index the table. Such a "super dict" is called a multi-index dictionary (MIDict).

A multi-index dictionary `user` can be constructed with two arguments: a list of items (rows of data), and a list of index names:

```
user = MIDict([['jack', 1, '192.1'],
               ['tony', 2, '192.2']],
              ['name', 'uid', 'ip'])
```

Index names are for easy human understanding and indexing, and thus must be a string. The index names and items are ordered in the dictionary. Compatible with a normal `dict`, the first index (column) is the primary index to lookup/index a key, while the rest index or indices contain the corresponding key's value or list of values:

```
user['jack'] -> [1, '192.1']
```

To use any index (column) as the "keys", and other one or more indices as the "values", just specify the indices via the advanced "multi-indexing" syntax `d[index_key:key, index_value]`, e.g.:

```
user['name':'jack', 'uid'] -> 1
user['ip':'192.1', 'name'] -> 'jack'
```

Here, `index_key` is the single column used as the "keys", and `key` is an element in `index_key` to locate the row of record (e.g., `['jack', 1, '192.1']`) in the table. `index_value` can be one or more columns to specify the value(s) from the row of record.

6.1.2 Multi-value indexing syntax

For a multi-column data set, it's useful to be able to access multiple values/columns at the same time.

In the indexing syntax `d[index_key:key, index_value]`, both `index_key` and `index_value` can be a normal index name or an `int` (the order the index), and `index_value` can also be a `tuple`, `list` or `slice` object to specify multiple values/columns, with the following meanings:

type	meaning	corresponding values
int	the index of a key in d.keys()	the value of the key
tuple/list	multiple keys or indices of keys	list of values
slice	a range of keys (key_start:key_stop:step)	list of values

The elements in the tuple/list or `key_start`/`key_stop` in the slice syntax can be a normal index name or an `int`.

See *`midict.IndexDict`* for more details.

Using the above `user` example:

```
user['name':'jack', ['uid', 'ip']] -> [1, '192.1']
<==> user['name':'jack', [1, 2]]
<==> user['name':'jack', 'uid':]
<==> user[0:'jack', 1:]
```

6.1.3 Convenient indexing shortcuts

Full syntax: `d[index_key:key, index_value]`

Short syntax:

```
d[key] <==> d[first_index:key, all_indice_except_first_index]
d[:key] <==> d[None:key] <==> d[last_index:key, all_indice_except_last_index]
d[key, index_value] <==> d[first_index:key, index_value] # only when ``index_value`` is a list or sli
d[index_key:key, index_value_1, index_value_2, ...] <==> d[index_key:key, (index_value_1, index_value
```

Examples:

```
user['jack'] -> [1, '192.1']
user[:'192.1'] -> ['jack', 1]
user['jack', :] -> ['jack', 1, '192.1']
user['jack', ['uid', 'ip']] -> [1, '192.1']
user[0:'jack', 'uid', 'ip'] -> [1, '192.1']
```

6.1.4 Bidirectional/inverse dict

With the advanced "multi-indexing" syntax, a MIDict with 2 indices can be used as a normal dict, as well as a convenient **bidirectional dict** to index using either a key or a value:

```
mi_dict = MIDict(jack=1, tony=2)
```

- Forward indexing like a normal dict (`d[key] -> value`):

```
    mi_dict['jack'] -> 1
    <==> mi_dict[0:'jack', 1]
```

- Backward/inverse indexing using the slice syntax (`d[:value] -> key`):

```
    mi_dict[:1] -> 'jack'
    <==> mi_dict[-1:1, 0]
```

6.1.5 Compatible with normal dict in Python 2 & 3

A `MIDict` with 2 indices is fully compatible with the normal dict or OrderedDict, and can be used as a drop-in replacement of the latter:

```
normal_dict = dict(jack=1, tony=2)
mi_dict = MIDict(jack=1, tony=2)
```

The following equality checks all return `True`:

```
mi_dict == normal_dict
normal_dict['jack'] == mi_dict['jack'] == 1
normal_dict.keys() == mi_dict.keys() == ['tony', 'jack']
normal_dict.values() == mi_dict.values() == [2, 1]
```

Conversion between `MIDict` and `dict` is supported in both directions:

```
mi_dict == MIDict(normal_dict) # True
normal_dict == dict(mi_dict) # True
normal_dict == mi_dict.todict() # True
```

The `MIDict` API also matches the `dict` API in Python 2 & 3. For example, in Python 2, `MIDict` has methods `keys()`, `values()` and `items()` that return lists. In Python 3, those methods return dictionary views, just like `dict`.

6.1.6 Accessing keys via attributes

Use the attribute syntax to access a key in MIDict if it is a valid Python identifier (`d.key <==> d['key']`):

```
mi_dict.jack <==> mi_dict['jack']
```

This feature is supported by *`midict.AttrDict`*.

Note that it treats an attribute as a dictionary key only when it can not find a normal attribute with that name. Thus, it is the programmer's responsibility to choose the correct syntax while writing the code.

6.1.7 Extended methods for multi-indices

A series of methods are extended to accept an optional agrument to specify which index/indices to use, including `keys()`, `values()`, `items()`, `iterkeys()`, `itervalues()`, `iteritems()`, `viewkeys()`, `viewvalues()`,`viewitems()`,`__iter__()` and `__reversed__()`:

```
user = MIDict([['jack', 1, '192.1'],
               ['tony', 2, '192.2']],
              ['name', 'uid', 'ip'])

user.keys() <==> user.keys(0) <==> user.keys('name') -> ['jack', 'tony']
user.keys('uid') <==> user.keys(1) -> [1, 2]

user.values() <==> user.values(['uid', 'ip']) -> [[1, '192.1'], [2, '192.2']]
user.values('uid') -> [1, 2]
user.values(['name','ip']) -> [['jack', '192.1'], ['tony', '192.2']]

user.items() <==> user.values(['name', 'uid', 'ip'])
                    -> [['jack', 1, '192.1'], ['tony', 2, '192.2']]
user.items(['name','ip']) -> [['jack', '192.1'], ['tony', '192.2']]
```

MIDict also provides two handy methods `d.viewdict(index_key, index_value)` and `d.todict(dict_type, index_key, index_value)` to view it as a normal dict or convert it to a specific type of dict using specified indices as keys and values.

6.1.8 Additional APIs to handle indices

MIDict provides special methods (`d.reorder_indices()`, `d.rename_index()`, `d.add_index()`, `d.remove_index()`) to handle the indices:

```
d = MIDict([['jack', 1], ['tony', 2]], ['name', 'uid'])

d.reorder_indices(['uid', 'name'])
d -> MIDict([[1, 'jack'], [2, 'tony']], ['uid', 'name'])

d.reorder_indices(['name', 'uid']) # change back indices

d.rename_index('uid', 'userid') # rename one index
<==> d.rename_index(['name', 'userid']) # rename all indices
d -> MIDict([['jack', 1], ['tony', 2]], ['name', 'userid'])

d.add_index(values=['192.1', '192.2'], name='ip')
d -> MIDict([['jack', 1, '192.1'], ['tony', 2, '192.2']],
            ['name', 'userid', 'ip'])

d.remove_index('userid')
d -> MIDict([['jack', '192.1'], ['tony', '192.2']], ['name', 'ip'])
d.remove_index(['name', 'ip']) # remove multiple indices
d -> MIDict() # empty
```

6.1.9 Duplicate keys/values handling

The elements in each index of MIDict should be unique.

When setting an item using syntax `d[index_key:key, index_value] = value2`, if `key` already exists in `index_key`, the item of `key` will be updated according to `index_value` and `value2` (similar to updat-

ing the value of a key in a normal `dict`). However, if any value of `value2` already exists in `index_value`, a `ValueExistsError` will be raised.

When constructing a MIDict or updating it with `d.update()`, duplicate keys/values are handled in the same way as above with the first index treated as `index_key` and the rest indices treated as `index_value`:

```
d = MIDict(jack=1, tony=2)

d['jack'] = 10 # replace value of key 'jack'
d['tom'] = 3 # add new key/value
d['jack'] = 2 # raise ValueExistsError
d['alice'] = 2 # raise ValueExistsError
d[:2] = 'jack' # raise ValueExistsError
d['jack', :] = ['tony', 22] # raise ValueExistsError
d['jack', :] = ['jack2', 11] # replace key 'jack' to a new key 'jack2' and value to 11

d.update([['alice', 2]]) # raise ValueExistsError
d.update(alice=2) # raise ValueExistsError
d.update(alice=4) # add new key/value

MIDict([['jack',1]], jack=2) # {'jack': 2}
MIDict([['jack',1], ['jack',2]]) # {'jack': 2}
MIDict([['jack',1], ['tony',1]]) # raise ValueExistsError
MIDict([['jack',1]], tony=1) # raise ValueExistsError
```

6.1.10 Internal data struture

Essentially `MIDict` is a `Mapping` type, and it stores the data in the form of `{key: value}` for 2 indices (identical to a normal `dict`) or `{key: list_of_values}` for more than 2 indices.

Additionally, MIDict uses a special attribute `d.indices` to store the indices, which is an `IdxOrdDict` instance with the index names as keys (the value of the first index is the `MIDict` instance itself, and the value of each other index is an `AttrOrdDict` instance which maps each element in that index to its corresponding element in the first index):

```
d = MIDict([['jack', 1], ['tony', 2]], ['name', 'uid'])

d.indices ->

    IdxOrdDict([
        ('name', MIDict([('jack', 1), ('tony', 2)], ['name', 'uid'])),
        ('uid', AttrOrdDict([(1, 'jack'), (2, 'tony')])),
    ])
```

Thus, `d.indices` also presents an interface to access the indices and items.

For example, access index names:

```
'name' in d.indices -> True
list(d.indices) -> ['name', 'uid']
d.indices.keys() -> ['name', 'uid']
```

Access items in an index:

```
'jack' in d.indices['name'] -> True
1 in d.indices['uid'] -> True
list(d.indices['name']) -> ['jack', 'tony']
list(d.indices['uid']) -> [1, 2]
```

```
d.indices['name'].keys() -> ['jack', 'tony']
d.indices['uid'].keys() -> [1, 2]
```

`d.indices` also supports the attribute syntax:

```
d.indices.name -> MIDict([('jack', 1), ('tony', 2)], ['name', 'uid'])
d.indices.uid -> AttrOrdDict([(1, 'jack'), (2, 'tony')])
```

However, the keys/values in `d.indices` should not be directly changed, otherwise the structure or the references may be broken. Use the methods of `d` rather than `d.indices` to operate the data.

6.1.11 More examples of advanced indexing

- Example of two indices (compatible with normal dict):

```
    color = MIDict([['red', '#FF0000'], ['green', '#00FF00']],
                   ['name', 'hex'])

    # flexible indexing of short and long versions:

    color.red # -> '#FF0000'
    <==> color['red']
    <==> color['name':'red']
    <==> color[0:'red'] <==> color[-2:'red']
    <==> color['name':'red', 'hex']
    <==> color[0:'red', 'hex'] <==> color[-2:'red', 1]

    color[:'#FF0000'] # -> 'red'
    <==> color['hex':'#FF0000']
    <==> color[1:'#FF0000'] <==> color[-1:'#FF0000']
    <==> color['hex':'#FF0000', 'name'] <==> color[1:'#FF0000', 0]


    # setting an item using different indices/keys:

    color.blue = '#0000FF'
    <==> color['blue'] = '#0000FF'
    <==> color['name':'blue'] = '#0000FF'
    <==> color['name':'blue', 'hex'] = '#0000FF'
    <==> color[0:'blue', 1] = '#0000FF'

    <==> color[:'#0000FF'] = 'blue'
    <==> color[-1:'#0000FF'] = 'blue'
    <==> color['hex':'#0000FF'] = 'blue'
    <==> color['hex':'#0000FF', 'name'] = 'blue'
    <==> color[1:'#0000FF', 0] = 'blue'

    # result:
    # color -> MIDict([['red', '#FF0000'],
                       ['green', '#00FF00'],
                       ['blue', '#0000FF']],
                      ['name', 'hex'])
```

- Example of three indices:

```
    user = MIDict([[1, 'jack', '192.1'],
                   [2, 'tony', '192.2']],
                  ['uid', 'name', 'ip'])
```

```
    user[1]                     -> ['jack', '192.1']
    user['name':'jack']         -> [1, '192.1']
    user['uid':1, 'ip']         -> '192.1'
    user[1, ['name','ip']]      -> ['jack', '192.1']
    user[1, ['name',-1]]        -> ['jack', '192.1']
    user[1, [1,1,0,0,2,2]]      -> ['jack', 'jack', 1, 1, '192.1', '192.1']
    user[1, :]                  -> [1, 'jack', '192.1']
    user[1, ::2]                -> [1, '192.1']
    user[1, 'name':]            -> ['jack', '192.1']
    user[1, 0:-1]               -> [1, 'jack']
    user[1, 'name':-1]          -> ['jack']
    user['uid':1, 'name','ip']  -> ['jack', '192.1']
    user[0:3, ['name','ip']] = ['tom', '192.3'] # set a new item explictly
    <==> user[0:3] = ['tom', '192.3'] # set a new item implicitly
    # result:
    # user -> MIDict([[1, 'jack', '192.1'],
                      [2, 'tony', '192.2'],
                      [3, 'tom', '192.3']],
                     ['uid', 'name', 'ip'])
```

6.1.12 More classes and functions

Check midict package API for more classes and functions, such as *midict.FrozenMIDict*, *midict.AttrDict*, *midict.IndexDict*, *midict.MIDictView*, etc.

6.1.13 Related libraries

- bidict provides bidirectional/inverse mapping via the `d.inv` attribute.
- multi_key_dict maps multiple keys to the same value.
- orderedmultidict stores multiple values for the same key.
- mdict retrieves and stores values in nested python dicts.
- pandas has a hierarchical/multi-level indexing feature (pandas.MultiIndex).

6.2 midict package API

6.2.1 midict.AttrDict

class `midict.`**`AttrDict`**`(`**args*, ***kw*`)`

Bases: `dict`

A dictionary that can get/set/delete a key using the attribute syntax if it is a valid Python identifier. (`d.key` <==> `d['key']`)

Note that it treats an attribute as a dictionary key only when it can not find a normal attribute with that name. Thus, it is the programmer's responsibility to choose the correct syntax while writing the code.

Be aware that besides all the inherited attributes, AttrDict has an additional internal attribute "_AttrDict__attr2item".

Examples:

```
d = AttrDict(__init__='value for key "__init__"')
d.__init__ -> <bound method AttrDict.__init__>
d["__init__"] -> 'value for key "__init__"'
```

__init__(**args*, ***kw*)
Init the dict using the same arguments for `dict`.

set any attributes here (or in subclass) - before __init__() so that these remain as normal attributes

__getattr__(*item*)
Maps values to attributes. Only called if there *isn't* an attribute with this name

__setattr__(*item*, *value*)
Maps attributes to values. Only if initialized and there *isn't* an attribute with this name

__delattr__(*item*)
Maps attributes to values. Only if there *isn't* an attribute with this name

class midict.**AttrOrdDict**(**args*, ***kw*)
Bases: *midict.AttrDict*, collections.OrderedDict

AttrDict + OrderedDict

6.2.2 midict.IndexDict

class midict.**IndexDict**(**args*, ***kw*)
Bases: dict

A dictionary that supports flexible indexing (get/set/delete) of multiple keys via an int, tuple, list or slice object.

The type of a valid key in IndexDict should not be int, tuple, or NoneType.

To index one or more items, use a proper `item` argument with the bracket syntax: `d[item]`. The possible types and contents of `item` as well as the corresponding values are summarized as follows:

type	content of the `item` argument	corresponding values
int	the index of a key in d.keys()	the value of the key
tuple/list	multiple keys or indices of keys	list of values
slice	"key_start : key_stop : step"	list of values
other types	a normal key	the value of the key

The tuple/list syntax can mix keys with indices of keys.

The slice syntax means a range of keys (like the normal list slicing), and the `key_start` and `key_stop` parameter can be a key, the index of a key, or None (which can be omitted).

When setting items, the slice and int syntax (including int in the tuple/list syntax) can only be used to change values of existing keys, rather than set values for new keys.

Examples:

```
d = IndexDict(a=1,b=2,c=3)

d -> {'a': 1, 'c': 3, 'b': 2}
d.keys() -> ['a', 'c', 'b']

d['a'] -> 1
d[0] -> 1
d['a','b'] <==> d[('a','b')] <==> d[['a','b']] -> [1, 2]
d[:] -> [1,3,2]
d['a':'b'] <==> d[0:2] <==> d['a':2] <==> d['a':-1] -> [1, 3]
```

```
    d[0::2] -> [1, 2]

    d[0] = 10 # d -> {'a': 10, 'c': 3, 'b': 2}
    d['a':-1] = [10, 30] # d -> {'a': 10, 'c': 30, 'b': 2}

    d[5] = 10 -> KeyError: 'Index out of range of keys: 5'
```

__init__(**args*, ***kw*)
check key is valid

__getitem__(*item*)
Get one or more items using flexible indexing.

__setitem__(*item*, *value*)
Set one or more items using flexible indexing.

The slice and int syntax (including int in the tuple/list syntax) can only be used to change values of existing keys, rather than set values for new keys.

__delitem__(*item*)
Delete one or more items using flexible indexing.

__contains__(*item*)
Check if the dictionary contains one or more items using flexible indexing.

class midict.**IdxOrdDict**(**args*, ***kw*)
Bases: *midict.IndexDict*, *midict.AttrDict*, collections.OrderedDict

IndexDict + AttrDict + OrderedDict

6.2.3 midict.MIMapping

class midict.**MIMapping**(**args*, ***kw*)
Bases: *midict.AttrOrdDict*

Base class for all provided multi-index dictionary (MIDict) types.

Mutable and immutable MIDict types extend this class, which implements all the shared logic. Users will typically only interact with subclasses of this class.

__init__(**args*, ***kw*)
Init dictionary with items and index names:

```
        (items, names, **kw)
        (dict, names, **kw)
        (MIDict, names, **kw)
```

`names` and `kw` are optional.

`names` must all be str or unicode type. When `names` not present, index names default to: 'index_0', 'index_1', etc. When keyword arguments present, only two indices allowed (like a normal dict)

Examples:

```
        index_names = ['uid', 'name', 'ip']
        rows_of_data = [[1, 'jack', '192.1'],
                        [2, 'tony', '192.2']]

        user = MIDict(rows_of_data, index_names)
```

```
user = MIDict(rows_of_data)
<==> user = MIDict(rows_of_data, ['index_0', 'index_1', 'index_2'])
```

Construct from normal dict:

```
normal_dict = {'jack':1, 'tony':2}
user = MIDict(normal_dict.items(), ['name', 'uid'])
# user -> MIDict([['tony', 2], ['jack', 1]], ['name', 'uid'])
```

__getitem__(*args*)
get values via multi-indexing

__setitem__(*args*, *value*)
set values via multi-indexing

__delitem__(*args*)
delete a key (and the whole item) via multi-indexing

__eq__(*other*)
Test for equality with `other`.

if `other` is a regular mapping/dict, compare only order-insensitive keys/values. if `other` is also a OrderedDict, also compare the order of keys. if `other` is also a MIDict, also compare the index names.

__lt__(*other*)
Check if `self < other`

If `other` is not a Mapping type, return NotImplemented.

If `other` is a Mapping type, compare in the following order:

- convert `self` to an OrderedDict or a dict (depends on the type of `other`) and compare it with `other`
- index names (only if `other` is a MIMapping)

__le__(*other*)
self <= other

__gt__(*other*)
self > other

__ge__(*other*)
self >= other

__repr__(*_repr_running={}*)
repr as "MIDict(items, names)"

__reduce__()
Return state information for pickling

copy()
a shallow copy

clear(*clear_indices=False*)
Remove all items. index names are removed if `clear_indices==True`.

classmethod fromkeys(*keys*, *value=None*, *names=None*)
Create a new dictionary with keys from `keys` and values set to `value`.

fromkeys() is a class method that returns a new dictionary. `value` defaults to None.

Length of `keys` must not exceed one because no duplicate values are allowed.

Optional `names` can be provided for index names (of length 2).

get(*key*, *default=None*)
Return the value for `key` if `key` is in the dictionary, else `default`. If `default` is not given, it defaults to None, so that this method never raises a `KeyError`.

Support "multi-indexing" keys

__contains__(*key*)
Test for the presence of `key` in the dictionary.

Support "multi-indexing" keys

__iter__(*index=None*)
Iterate through keys in the `index` (defaults to the first index)

__reversed__(*index=None*)
Iterate in reversed order through keys in the `index` (defaults to the first index)

iterkeys(*index=None*)
Iterate through keys in the `index` (defaults to the first index)

keys(*index=None*)
a set-like object providing a view on the keys in `index` (defaults to the first index)

itervalues(*index=None*)
Iterate through values in the `index` (defaults to all indices except the first index).

When `index is None`, yielded values depend on the length of indices (`N`):

- if N <= 1: return
- if N == 2: yield values in the 2nd index
- if N > 2: yield values in all indices except the first index (each value is a list of `N-1` elements)

values(*index=None*)
a set-like object providing a view on the values in `index` (defaults to all indices except the first index)

iteritems(*indices=None*)
Iterate through items in the `indices` (defaults to all indices)

items(*index=None*)
a set-like object providing a view on the items in `index` (defaults to all indices)

update(**args*, ***kw*)
Update the dictionary

viewdict(*index_key=None*, *index_value=None*)
a dict-like object providing a view on the keys in `index_key` (defaults to the first index) and values in `index_value` (defaults to the last index)

todict(*dict_type=<class 'dict'>*, *index_key=0*, *index_value=-1*)
convert to a specific type of dict using `index_key` as keys and `index_value` as values (discarding index names)

6.2.4 midict.MIDict

class `midict.`**MIDict**(**args*, ***kw*)
Bases: *midict.MIMapping*

MIDict is an ordered "dictionary" with multiple indices where any index can serve as "keys" or "values", capable of assessing multiple values via its powerful indexing syntax, and suitable as a bidirectional/inverse dict (a drop-in replacement for dict/OrderedDict in Python 2 & 3).

Features:

- Multiple indices
- Multi-value indexing syntax
- Convenient indexing shortcuts
- Bidirectional/inverse dict
- Compatible with normal dict in Python 2 & 3
- Accessing keys via attributes
- Extended methods for multi-indices
- Additional APIs to handle indices
- Duplicate keys/values handling

__setitem__(*args*, *value*)
set values via multi-indexing

If `d.indices` is empty (i.e., no index names and no items are set), index names can be created when setting a new item with specified names (`index1` and `index2` can not be int or slice):

```
d = MIDict()
d['uid':1, 'name'] = 'jack'
# d -> MIDict([[1, 'jack']], ['uid', 'name'])

d = MIDict()
d[1] = 'jack' # using default index names
<==> d[:'jack'] = 1
# d -> MIDict([(1, 'jack')], ['index_1', 'index_2'])
```

If `d.indices` is not empty, when setting a new item, all indices of the item must be specified via `index1` and `index2` (implicitly or explicitly):

```
d = MIDict([['jack', 1, '192.1']], ['name', 'uid', 'ip'])
d['tony'] = [2, '192.2']
<==> d['name':'tony',['uid', 'ip']] = [2, '192.2']
# the following will not work:
d['alice', ['uid']] = [3] # raise ValueError
```

More examles:

```
d = MIDict(jack=1, tony=2)

d['jack'] = 10 # replace value of key 'jack'
d['tom'] = 3 # add new key/value
d['jack'] = 2 # raise ValueExistsError
d['alice'] = 2 # raise ValueExistsError
d[:2] = 'jack' # raise ValueExistsError
d['jack', :] = ['tony', 22] # raise ValueExistsError
d['jack', :] = ['jack2', 11] # replace item of key 'jack'
```

__delitem__(*args*)
delete a key (and the whole item) via multi-indexing

clear(*clear_indices=False*)
Remove all items. index names are removed if `clear_indices==True`.

update(**args*, ***kw*)
Update the dictionary with items and names:

```
(items, names, **kw)
(dict, names, **kw)
(MIDict, names, **kw)
```

Optional positional argument `names` is only allowed when `self.indices` is empty (no indices are set yet).

rename_index(**args*)
change the index name(s).

- **call with one argument:**
  1. list of new index names (to replace all old names)
- **call with two arguments:**
  1. old index name(s) (or index/indices)
  2. new index name(s)

reorder_indices(*indices_order*)
reorder all the indices

add_index(*values*, *name=None*)
add an index of `name` with the list of `values`

remove_index(*index*)
remove one or more indices

6.2.5 midict.FrozenMIDict

class `midict.`**FrozenMIDict**(**args*, ***kw*)
Bases: *midict.MIMapping*, `collections.abc.Hashable`

An immutable, hashable multi-index dictionary (similar to `MIDict`).

__hash__()
Return the hash of this bidict.

6.2.6 Exceptions

exception `midict.`**MIMappingError**
Bases: `Exception`

Base class for MIDict exceptions

exception `midict.`**ValueExistsError**
Bases: `KeyError`, *midict.MIMappingError*

Value already exists in an index and can not be used as a key.

Usage:

```
ValueExistsException(value, index_order, index_name)
```

__str__()
Get a string representation of this exception for use with str.

6.2.7 Dict views

class midict.**MIKeysView**(*mapping*, *index=None*)
Bases: collections.abc.KeysView

a set-like object providing a view on the keys in index (defaults to the first index)

class midict.**MIValuesView**(*mapping*, *index=None*)
Bases: collections.abc.ValuesView

a set-like object providing a view on the values in index (defaults to all indices except the first index)

class midict.**MIItemsView**(*mapping*, *index=None*)
Bases: collections.abc.ItemsView

a set-like object providing a view on the items in index (defaults to all indices)

class midict.**MIDictView**(*mapping*, *index_key=None*, *index_value=None*)
Bases: collections.abc.KeysView

a dict-like object providing a view on the keys in index_key (defaults to the first index) and values in index_value (defaults to the last index)

6.2.8 Auxiliary functions

midict.**_MI_init**(*self*, **args*, ***kw*)
Separate __init__ function of MIMapping

midict.**_MI_setitem**(*self*, *args*, *value*)
Separate __setitem__ function of MIMapping

midict.**force_list**(*a*)
convert an iterable a into a list if it is not a list.

midict.**cvt_iter**(*a*)
Convert an iterator/generator to a tuple so that it can be iterated again.

E.g., convert zip in PY3.

midict.**convert_dict**(*d*, *cls=<class 'midict.AttrDict'>*)
recursively convert a normal Mapping *d* and it's values to a specified type (defaults to AttrDict)

midict.**convert_key_to_index**(*keys*, *key*)
convert key of various types to int or list of int

return index, single

midict.**convert_index_to_keys**(*d*, *item*)
Convert item in various types (int, tuple/list, slice, or a normal key) to a single key or a list of keys.

midict.**IndexDict_check_key_type**(*key*)
raise TypeError if key is int, tuple or NoneType

midict.**MI_check_index_name**(*name*)
Check if index name is a valid str or unicode

midict.**get_unique_name**(*name=''*, *collection=()*)
Generate a unique name (str type) by appending a sequence number to the original name so that it is not contained in the collection. collection has a __contains__ method (tuple, list, dict, etc.)

midict.**get_value_len**(*value*)
Get length of value. If value (eg, iterator) has no len(), convert it to list first.

return both length and converted value.

midict.**MI_parse_args**(*self*, *args*, *ingore_index2=False*, *allow_new=False*)
Parse the arguments for indexing in MIDict.

Full syntax: `d[index1:key, index2]`.

`index2` can be flexible indexing (int, list, slice etc.) as in `IndexDict`.

Short syntax:

•d[key] <==> d[key,] <==> d[first_index:key, all_indice_except_first]

•d[:key] <==> d[:key,] <==> d[None:key] <==> d[last_index:key, all_indice_except_last]

•d[key, index2] <==> d[first_index:key, index2] # this is valid # only when index2 is a list or slice object

•d[index1:key, index2_1, index2_2, ...] <==> d[index1:key, (index2_1, index2_2, ...)]

midict.**mget_list**(*item*, *index*)
get mulitple items via index of int, slice or list

midict.**mset_list**(*item*, *index*, *value*)
set mulitple items via index of int, slice or list

midict.**MI_get_item**(*self*, *key*, *index=0*)
return list of item

midict.**od_replace_key**(*od*, *key*, *new_key*, **args*, ***kw*)
Replace key(s) in OrderedDict `od` by new key(s) in-place (i.e., preserving the order(s) of the key(s))

Optional new value(s) for new key(s) can be provided as a positional argument (otherwise the old value(s) will be used):

od_replace_key(od, key, new_key, new_value)

To replace multiple keys, pass argument `key` as a list instance, or explicitly pass a keyword argument `multi=True`:

od_replace_key(od, keys, new_keys, [new_values,] multi=True)

midict.**od_reorder_keys**(*od*, *keys_in_new_order*)
Reorder the keys in an OrderedDict `od` in-place.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

m

Symbols

A

C

F

G

I

K

M

O

R

T

U

V