Sider Documentation

Release 0.1.1

Hong Minhee

March 29, 2012

CONTENTS

Sider is a persistent object library based on Redis. This is being planned and heavily under development currently.

```
>>> from sider.types import Set, Integer
>>> s = session.get('my_set', Set(Integer))
>>> 3 in s  # SISMEMBER 3
True
>>> 4 in s  # SISMEMBER 4
False
>>> s2 = session.get('another_set', Set(Integer))
>>> s & s2  # SINTER my_set another_set
set([2, 3])
>>> s
<sider.set.Set {1, 2, 3}>
>>> s2
<sider.set.Set {-1, 0, 1, 2}>
>>> session.get('my_int_key', Integer)
1234
```

You can install it from PyPI:

```
$ pip install Sider  # or
$ easy_install Sider
$ python -m sider.version
0.1.1
```

What was the name 'Sider' originated from?:

```
>>> 'redis'[::-1]
'sider'
```

CHAPTER

ONE

REFERENCES

1.1 sider — Sider

1.1.1 sider.session — Sessions

What sessions mainly do are identity map and unit of work.

class `sider.session.`**`Session`**(*client*)

Session is an object which manages Python objects that represent Redis values e.g. lists, sets, hashes. It maintains identity maps between Redis values and Python objects, and deals with transactions.

Parameters **client** (`redis.client.Client`) – the Redis client

`get`(*key*, *value_type=<class 'sider.types.ByteString'>*)

Loads the value from the `key`. If `value_type` is present the value will be treated as it, or `ByteString` by default.

Parameters

- **key** (`str`) – the Redis key to load
- **value_type** (`Value`, `type`) – the type of the value to load. default is `ByteString`

Returns the loaded value

`server_version`

(`str`) Redis server version string e.g. `'2.2.11'`.

`server_version_info`

(`tuple`) Redis server version triple e.g. `(2, 2, 11)`. You can compare versions using this property.

`set`(*key*, *value*, *value_type=<class 'sider.types.ByteString'>*)

Stores the `value` into the `key`. If `value_type` is present the value will be treated as it, or `ByteString` by default.

Parameters

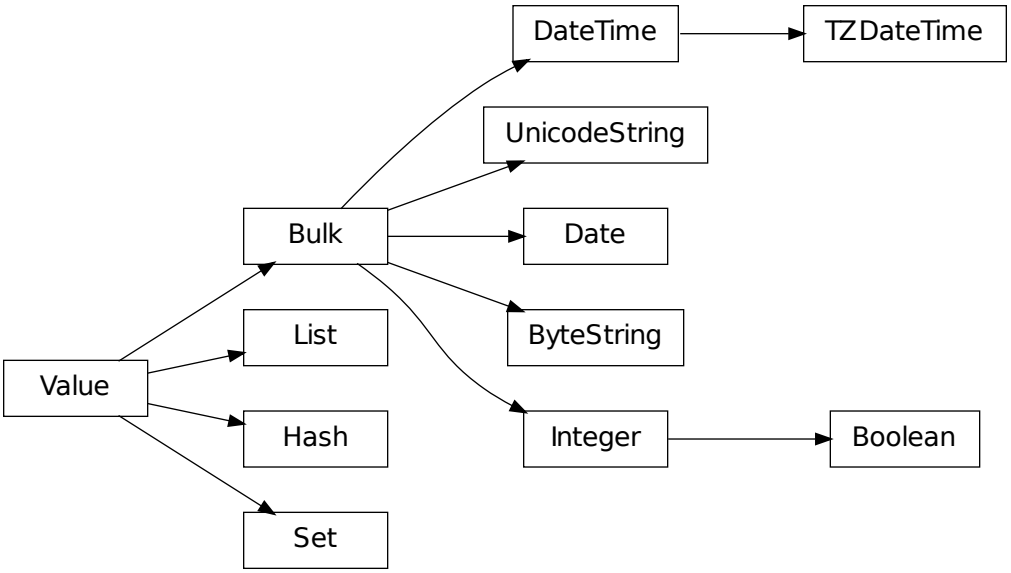
- **key** (`str`) – the Redis key to save the value into
- **value** – the value to be saved
- **value_type** (`Value`, `type`) – the type of the `value`. default is `ByteString`

Returns the Python representation of the saved value. it is equivalent to the given `value` but may not equal nor the same to

1.1.2 `sider.types` — Conversion between Python and Redis types

In Redis all data are byte strings — *bulks*. Lists are lists of byte strings, sets are sets of byte strings, and hashes consist of byte string keys and byte string values.

To store richier objects into Redis we have to encode Python values and decode Redis data. `Bulk` and its subclasses are for that, it defines two basic methods: `encode()` and `decode()`. For example, `Integer` encodes Python `int 3` into Redis bulk `"3"` and decodes Redis bulk `"3"` into Python `int 3`.



class `sider.types.`**`Boolean`**

Bases: `sider.types.Integer`

Stores `bool` values as `'1'` or `'0'`.

```
>>> boolean = Boolean()
>>> boolean.encode(True)
'1'
>>> boolean.encode(False)
'0'
```

class `sider.types.`**`Bulk`**

Bases: `sider.types.Value`

The abstract base class to be subclassed. You have to implement `encode()` and `decode()` methods in subclasses.

`decode`(*bulk*)

Decodes a Redis `bulk` to Python object. Every subclass of `Bulk` must implement this method. By default it raises `NotImplementedError`.

Parameters **bulk** (`str`) – a Redis bullk to decode into Python object

Returns a decoded Python object

encode(*value*)

Encodes a Python `value` into Redis bulk. Every subclass of `Bulk` must implement this method. By default it raises `NotImplementedError`.

Parameters **value** – a Python value to encode into Redis bulk

Returns an encoded Redis bulk

Return type `str`

Raises **exceptions.TypeError** if the type of a given value is not acceptable by this type

class `sider.types.`**ByteString**

Bases: `sider.types.Bulk`

Stores byte strings. It stores the given byte strings as these are. It works simply transparently for `str` values.

```
>>> bytestr = ByteString()
>>> bytestr.encode('annyeong')
'annyeong'
>>> bytestr.decode('sayonara')
'sayonara'
```

class `sider.types.`**Date**

Bases: `sider.types.Bulk`

Stores `datetime.date` values. Dates are internally formatted in **RFC 3339** format e.g. `2012-03-28`.

```
>>> import datetime
>>> date = Date()
>>> date.encode(datetime.date(2012, 3, 28))
'2012-03-28'
>>> date.decode(_)
datetime.date(2012, 3, 28)
```

DATE_FORMAT = '%Y-%m-%d'

(`str`) The `strftime()` format string for **RFC 3339**.

DATE_PATTERN = <_sre.SRE_Pattern object at 0x3cd99c0>

The `re` pattern that matches to **RFC 3339** formatted date string e.g. `'2012-03-28'`.

class `sider.types.`**DateTime**

Bases: `sider.types.Bulk`

Stores naive `datetime.datetime` values. Values are internally formatted in **RFC 3339** format e.g. `2012-03-28T09:21:34.638972`.

```
>>> dt = DateTime()
>>> dt.decode('2012-03-28T09:21:34.638972')
datetime.datetime(2012, 3, 28, 9, 21, 34, 638972)
>>> dt.encode(_)
'2012-03-28T09:21:34.638972'
```

It doesn't store `tzinfo` data.

```
>>> from sider.datetime import UTC
>>> decoded = dt.decode('2012-03-28T09:21:34.638972Z')
>>> decoded
datetime.datetime(2012, 3, 28, 9, 21, 34, 638972)
>>> dt.encode(decoded.replace(tzinfo=UTC))
'2012-03-28T09:21:34.638972'
```

Note: If you must be aware of time zone, use `TZDateTime` instead.

DATETIME_PATTERN = <_sre.SRE_Pattern object at 0x3e0adf0>

parse_datetime(*bulk*)

Parses a **RFC 3339** formatted string into `datetime.datetime`.

```
>>> dt = DateTime()
>>> dt.parse_datetime('2012-03-28T09:21:34.638972')
datetime.datetime(2012, 3, 28, 9, 21, 34, 638972)
```

Unlike `decode()` it is aware of `tzinfo` data if the string contains time zone notation.

```
>>> a = dt.parse_datetime('2012-03-28T09:21:34.638972Z')
>>> a
datetime.datetime(2012, 3, 28, 9, 21, 34, 638972,
                  tzinfo=sider.datetime.Utc())
>>> b = dt.parse_datetime('2012-03-28T18:21:34.638972+09:00')
>>> b
datetime.datetime(2012, 3, 28, 18, 21, 34, 638972,
                  tzinfo=sider.datetime.FixedOffset(540))
>>> a == b
True
```

Parameters **bulk** (`basestring`) – a *3339* formatted string

Returns a parsing result

Return type `datetime.datetime`

Note: It is for internal use and `decode()` method actually uses this method.

class `sider.types.`**Hash**(*key_type=None*, *value_type=None*)

Bases: `sider.types.Value`

The type object for `sider.hash.Hash` objects and other `collections.Mapping` objects.

Parameters

- **key_type** (`Bulk`, `type`) – the type of keys the hash will contain. default is `ByteString`
- **value_type** (`Bulk`, `type`) – the type of values the hash will contain. default is `ByteString`

class `sider.types.`**Integer**

Bases: `sider.types.Bulk`

Stores integers as decimal strings. For example:

```
>>> integer = Integer()
>>> integer.encode(42)
'42'
>>> integer.decode('42')
42
```

Why it doesn't store integers as binaries but decimals is that Redis provides INCR, INCRBY, DECR and DECRBY for decimal strings. You can simply add and subtract integers.

class `sider.types.`**`List`**(*value_type=None*)

Bases: `sider.types.Value`

The type object for `sider.list.List` objects and other `collections.Sequence` objects except strings. (Use `ByteString` or `UnicodeString` for strings.)

Parameters value_type (`Bulk`, `type`) – the type of values the list will contain. default is `ByteString`

class `sider.types.`**`Set`**(*value_type=None*)

Bases: `sider.types.Value`

The type object for `sider.set.Set` objects and other `collections.Set` objects.

Parameters value_type (`Bulk`, `type`) – the type of values the set will contain. default is `ByteString`

class `sider.types.`**`TZDateTime`**

Bases: `sider.types.DateTime`

Similar to `DateTime` except it accepts only tz-aware `datetime.datetime` values. All values are internally stored in `UTC`.

```
>>> from sider.datetime import FixedOffset
>>> dt = datetime.datetime(2012, 3, 28, 18, 21, 34, 638972,
...                        tzinfo=FixedOffset(540))
>>> tzdt = TZDateTime()
>>> tzdt.encode(dt)
'2012-03-28T09:21:34.638972Z'
>>> tzdt.decode(_)
datetime.datetime(2012, 3, 28, 9, 21, 34, 638972,
                  tzinfo=sider.datetime.Utc())
```

If any naive `datetime.datetime` has passed it will raise `ValueError`.

class `sider.types.`**`UnicodeString`**

Bases: `sider.types.Bulk`

Stores Unicode strings (`unicode`), not byte strings (`str`). Internally all Unicode strings are encoded into and decoded from UTF-8 byte strings.

```
>>> unistr = UnicodeString()
>>> unistr.encode(u'\uc720\ub2c8\ucf54\ub4dc')
'\xec\x9c\xa0\xeb\x8b\x88\xec\xbd\x94\xeb\x93\x9c'
>>> unistr.decode(_)
u'\uc720\ub2c8\ucf54\ub4dc'
```

class `sider.types.`**`Value`**

Bases: `object`

There are two layers behind Sider types: the lower one is this `Value` and the higher one is `Bulk`.

`Value` types can be set to Redis keys, but unlike `Bulk` it cannot be a value type of other rich `Value` types e.g. `List`, `Hash`.

In most cases you (users) don't have to subclass `Value`, and should not. Direct subclasses of `Value` aren't about encodings/decodings of Python object but simply Python-side representations of Redis types. It actually doesn't have methods like `encode()` and `decode()`. These methods appear under `Bulk` or its subtypes.

But it's about how to save Python objects into Redis keys and how to load values from associated Redis keys. There are several commands to save like SET, MSET, HSET, RPUSH and the rest in Redis and subtypes have to decide which command of those to use.

All subtypes of `Value` implement `save_value()` and `load_value()` methods. The constructor which takes no arguments have to be implemented as well.

classmethod ensure_value_type(*value_type*, *parameter=None*)

Raises a `TypeError` if the given `value_type` is not an instance of nor a subclass of the class.

```
>>> Integer.ensure_value_type(Bulk
... )
Traceback (most recent call last):
  ...
TypeError: expected a subtype of sider.types.Integer,
           but sider.types.Bulk was passed
>>> Integer.ensure_value_type(UnicodeString()
... )
Traceback (most recent call last):
  ...
TypeError: expected an instance of sider.types.Integer,
           but <sider.types.UnicodeString object at ...>
           was passed
>>> Bulk.ensure_value_type(1)
Traceback (most recent call last):
  ...
TypeError: expected a type, not 1
```

Otherwise it simply returns an instance of the given `value_type`.

```
>>> Bulk.ensure_value_type(Bulk)
<sider.types.Bulk object at ...>
>>> Bulk.ensure_value_type(ByteString)
<sider.types.ByteString object at ...>
>>> ByteString.ensure_value_type(ByteString
... )
<sider.types.ByteString object at ...>
>>> bytestr = ByteString()
>>> ByteString.ensure_value_type(bytestr)
<sider.types.ByteString object at ...>
```

If an optional `parameter` name has present, the error message becomes better.

```
>>> Integer.ensure_value_type(Bulk,
...   parameter='argname')
Traceback (most recent call last):
  ...
TypeError: argname must be a subtype of sider.types.Integer,
           but sider.types.Bulk was passed
>>> Integer.ensure_value_type(UnicodeString(),
...   parameter='argname'
... )
Traceback (most recent call last):
  ...
TypeError: argname must be an instance of sider.types.Integer,
           but <sider.types.UnicodeString object at ...>
           was passed
>>> Bulk.ensure_value_type(1, parameter='argname')
Traceback (most recent call last):
  ...
TypeError: argname must be a type, not 1
```

Parameters

- **value_type** (`Value`, `type`) – a type expected to be a subtype of the class
- **parameter** (`str`) – an optional parameter name. if present the error message becomes better

Raises **exceptions.TypeError** if the given `subtype` is not a subclass of the class

`load_value`(*session*, *key*)

How to load the value from the given Redis `key`. Subclasses have to implement it. By default it raises `NotImplementedError`.

Parameters

- **session** (`sider.session.Session`) – the session object that stores the given `key`
- **key** (`str`) – the key name to load

Returns the Python representation of the loaded value

`save_value`(*session*, *key*, *value*)

How to save the given `value` into the given Redis `key`. Subclasses have to implement it. By default it raises `NotImplementedError`.

Parameters

- **session** (`sider.session.Session`) – the session object going to store the given `key-value` pair
- **key** (`str`) – the key name to save the `value`
- **value** – the value to save into the `key`

Returns the Python representation of the saved value. it is equivalent to the given `value` but may not equal nor the same to

1.1.3 `sider.hash` — Hash objects

class `sider.hash.`**`Hash`**(*session*, *key*, *key_type=<class 'sider.types.ByteString'>*, *value_type=<class 'sider.types.ByteString'>*)

The Python-side representaion of Redis hash value. It behaves such as built-in Python `dict` object. More exactly, it implements `collections.MutableMapping` protocol.

Table 1.1: Mappings of Redis commands–Hash methods

Redis commands	`Hash` methods
DEL	`Hash.clear()`
HDEL	`del` (`Hash.__delitem__()`)
HEXISTS	`in` (`Hash.__contains__()`)
HGET	`Hash.__getitem__()`, `Hash.get()`
HGETALL	`Hash.items()`
HINCRBY	N/A
HINCRBYFLOAT	N/A
HKEYS	`iter()` (`Hash.__iter__()`), `Hash.keys()`
HLEN	`len()` (`Hash.__len__()`)
HMGET	N/A
HMSET	`Hash.update()`
HSET	= (`Hash.__setitem__()`)
HSETNX	`Hash.setdefault()`
HVALS	`Hash.values()`
N/A	`Hash.pop()`
N/A	`Hash.popitem()`

`__contains__`(*key*)

Tests whether the given `key` exists.

Parameters **key** – the key

Returns `True` if the `key` exists or `False`

Return type `bool`

Note: It is directly mapped to Redis HEXISTS command.

`__delitem__`(*key*)

Removes the `key`.

Parameters **key** – the key to delete

Raises

- **exceptions.TypeError** – if the given `key` is not acceptable by its `key_type`
- **exceptions.KeyError** – if there's no such `key`

Note: It is directly mapped to Redis HDEL command.

`__getitem__`(*key*)

Gets the value of the given `key`.

Parameters **key** – the key to get its value

Returns the value of the `key`

Raises

- **exceptions.TypeError** – if the given `key` is not acceptable by its `key_type`
- **exceptions.KeyError** – if there's no such `key`

Note: It is directly mapped to Redis HGET command.

__iter__()

Iterates over its `keys()`.

Returns the iterator which yields its keys

Return type `collections.Iterator`

Note: It is directly mapped to Redis HKEYS command.

__len__()

Gets the number of items.

Returns the number of items

Return type `numbers.Integral`

Note: It is directly mapped to Redis HLEN command.

__setitem__(*key*, *value*)

Sets the `key` with the `value`.

Parameters

- **key** – the key to set
- **value** – the value to set

Raises **exceptions.TypeError** if the given `key` is not acceptable by its `key_type` or the given `value` is not acceptable by its `value_type`

Note: It is directly mapped to Redis HSET command.

---

clear()

Removes all items from this hash.

Note: Under the hood it simply DEL the key.

---

items()

Gets its all `(key, value)` pairs. There isn't any meaningful order of pairs.

Returns the set of `(key, value)` pairs (`tuple`)

Return type `collections.ItemsView`

Note: This method is mapped to Redis HGETALL command.

---

key_type = **None**

(`sider.types.Bulk`) The type of hash keys.

keys()

Gets its all keys. Equivalent to `__iter__()` except it returns a `Set` instead of iterable. There isn't any meaningful order of keys.

Returns the set of its all keys

Return type `collections.KeysView`

Note: This method is directly mapped to Redis HKEYS command.

setdefault(*key*, *default=None*)

Sets the given `default` value to the `key` if it doesn't exist and then returns the current value of the `key`.

For example, the following code is:

```
val = hash.setdefault('key', 'set this if not exist')
```

equivalent to:

```
try:
    val = hash['key']
except KeyError:
    val = hash['key'] = 'set this if not exist'
```

except `setdefault()` method is an atomic operation.

Parameters

- **key** – the key to get or set
- **default** – the value to be set if the `key` doesn't exist

Raises **exceptions.TypeError** when the given `key` is not acceptable by its `key_type` or the given `default` value is not acceptable by its `value_type`

Note: This method internally uses Redis HSETNX command which is atomic.

update(*mapping={}*, ***keywords*)

Updates the hash from the given `mapping` and keyword arguments.

•If `mapping` has `keys()` method, it does:

```
for k in mapping:
    self[k] = mapping[k]
```

•If `mapping` lacks `keys()` method, it does:

```
for k, v in mapping:
    self[k] = v
```

•In either case, this is followed by (where `keywords` is a `dict` of keyword arguments):

```
for k, v in keywords.items():
    self[k] = v
```

Parameters

- **mapping** (`collections.Mapping`) – an iterable object of `(key, value)` pairs or a mapping object (which has `keys()` method). default is empty
- ****keywords** – the keywords to update as well. if its `key_type` doesn't accept byte strings (`str`) it raises `TypeError`

Raises

- **exceptions.TypeError** – if the `mapping` is not actually mapping or iterable, or the given keys and values to update aren't acceptable by its `key_type` and `value_type`
- **exceptions.ValueError** – if the `mapping` is an iterable object which yields non-pair values e.g. `[(1, 2, 3), (4,)]`

value_type = None
(`sider.types.Bulk`) The type of hash values.

values()
Gets its all values. It returns a `list` but there isn't any meaningful order of values.

Returns its all values

Return type `collections.ValuesView`

Note: This method is directly mapped to Redis HVALS command.

1.1.4 `sider.set` — Set objects

class `sider.set.`**Set**(*session*, *key*, *value_type=<class 'sider.types.ByteString'>*)
The Python-side representaion of Redis set value. It behaves alike built-in Python `set` object. More exactly, it implements `collections.MutableSet` protocol.

Table 1.2: Mappings of Redis commands–Set methods

Redis commands	`Set` methods
DEL	`Set.clear()`
SADD	`Set.add()`, `Set.update()`
SCARD	`len()` (`Set.__len__()`)
SDIFF	`Set.difference()`, `-` (`Set.__sub__()`)
SDIFFSTORE	`Set.difference_update()`, `-=` (`Set.__isub__()`)
SINTER	`Set.intersection()`, `&` (`Set.__and__()`)
SINTERSTORE	`Set.intersection_update()`, `&=` (`Set.__iand__()`)
SISMEMBER	`in` (`Set.__contains__()`)
SMEMBERS	`iter()` (`Set.__iter__()`)
SMOVE	N/A
SPOP	`Set.pop()`
SRANDMEMBER	N/A
SREM	`Set.discard()`, `Set.remove()`
SUNION	`Set.union()`, `|` (`Set.__or__()`)
SUNIONSTORE	`Set.update()`, `|=` (`Set.__ior__()`)
N/A	`Set.symmetric_difference()`, `^` (`Set.__xor__()`)
N/A	`Set.symmetric_difference_update()`, `^=` (`Set.__ixor__()`)

__and__(*operand*)
Bitwise and (`&`) operator. Gets the union of operands.

Mostly equivalent to `intersection()` method except it can take only one set-like operand. On the other hand `intersection()` can take zero or more iterable operands (not only set-like objects).

Parameters **operand** (`collections.Set`) – another set to get intersection

Returns the intersection

Return type `set`

__contains__(*member*)

`in` operator. Tests whether the set contains the given operand `member`.

Parameters **member** – the value to test

Returns `True` if the set contains the given operand `member`

Return type `bool`

Note: This method is directly mapped to SISMEMBER command.

__ge__(*operand*)

Greater-than or equal to (>=) operator. Tests whether the set is a superset of the given `operand`.

It's the same operation to `issuperset()` method except it can take a set-like operand only. On the other hand `issuperset()` can take an any iterable operand as well.

Parameters **operand** (`collections.Set`) – another set to test

Returns `True` if the set contains the `operand`

Return type `bool`

__gt__(*operand*)

Greater-than (>) operator. Tests whether the set is a *proper* (or *strict*) superset of the given `operand`.

To eleborate, the key difference between this greater-than (>) operator and greater-than or equal-to (>=) operator, which is equivalent to `issuperset()` method, is that it returns `False` even if two sets are exactly the same.

Let this show a simple example:

```
>>> assert isinstance(s, sider.set.Set)
>>> set(s)
set([1, 2, 3])
>>> s > set([1, 2]), s >= set([1, 2])
(True, True)
>>> s > set([1, 2, 3]), s >= set([1, 2, 3])
(False, True)
>>> s > set([1, 2, 3, 4]), s >= set([1, 2, 3, 4])
(False, False)
```

Parameters **operand** (`collections.Set`) – another set to test

Returns `True` if the set is a proper superset of `operand`

Return type `bool`

__iand__(*operand*)

Bitwise and (&=) assignment. Updates the set with the intersection of itself and the `operand`.

Mostly equivalent to `intersection_update()` method except it can take only one set-like operand. On the other hand `intersection_update()` can take zero or more iterable operands (not only set-like objects).

Parameters **operand** (`collections.Set`) – another set to intersection

Returns the set itself

Return type `Set`

__ior__(*operand*)

Bitwise or (|=) assignment. Updates the set with the union of itself and the `operand`.

Mostly equivalent to `update()` method except it can take only one set-like operand. On the other hand `update()` can take zero or more iterable operands (not only set-like objects).

Parameters **operand** (`collections.Set`) – another set to union

Returns the set itself

Return type `Set`

__isub__(*operand*)

Minus augmented assignment (−=). Removes all elements of the `operand` from this set.

Mostly equivalent to `difference_update()` method except it can take only one set-like operand. On the other hand `difference_update()` can take zero or more iterable operands (not only set-like objects).

Parameters **operand** (`collections.Set`) – another set which has elements to remove from this set

Returns the set itself

Return type `Set`

__ixor__(*operand*)

Bitwise exclusive argumented assignment (^=). Updates the set with the symmetric difference of itself and `operand`.

Mostly equivalent to `symmetric_difference_update()` method except it can take a set-like operand only. On the other hand `symmetric_difference_update()` can take an any iterable operand as well.

Parameters **operand** (`collections.Set`) – another set

Returns the set itself

Return type `Set`

__le__(*operand*)

Less-than or equal to (<=) operator. Tests whether the set is a subset of the given `operand`.

It's the same operation to `issubset()` method except it can take a set-like operand only. On the other hand `issubset()` can take an any iterable operand as well.

Parameters **operand** (`collections.Set`) – another set to test

Returns `True` if the `operand` set contains the set

Return type `bool`

__len__()

Gets the cardinality of the set.

Use this with the built-in `len()` function.

Returns the cardinality of the set

Return type `numbers.Integral`

Note: This method is directly mapped to SCARD command.

__lt__(*operand*)

Less-than (<) operator. Tests whether the set is a *proper* (or *strict*) subset of the given `operand` or not.

To eleborate, the key difference between this less-than (<) operator and less-than or equal-to (<=) operator, which is equivalent to `issubset()` method, is that it returns `False` even if two sets are exactly the same.

Let this show a simple example:

```
>>> assert isinstance(s, sider.set.Set)
>>> set(s)
set([1, 2, 3])
>>> s < set([1, 2]), s <= set([1, 2])
(False, False)
>>> s < set([1, 2, 3]), s <= set([1, 2, 3])
(False, True)
>>> s < set([1, 2, 3, 4]), s <= set([1, 2, 3, 4])
(True, True)
```

Parameters **operand** (`collections.Set`) – another set to test

Returns `True` if the set is a proper subset of `operand`

Return type `bool`

__or__(*operand*)

Bitwise or (|) operator. Gets the union of operands.

Mostly equivalent to `union()` method except it can take only one set-like operand. On the other hand `union()` can take zero or more iterable operands (not only set-like objects).

Parameters **operand** (`collections.Set`) – another set to union

Returns the union set

Return type `set`

__sub__(*operand*)

Minus (−) operator. Gets the relative complement of the `operand` in the set.

Mostly equivalent to `difference()` method except it can take a set-like operand only. On the other hand `difference()` can take an any iterable operand as well.

Parameters **operand** (`collections.Set`) – another set to get the relative complement

Returns the relative complement

Return type `set`

__xor__(*operand*)

Bitwise exclusive or (^) operator. Returns a new set with elements in either the set or the `operand` but not both.

Mostly equivalent to `symmetric_difference()` method except it can take a set-like operand only. On the other hand `symmetric_difference()` can take an any iterable operand as well.

Parameters **operand** (`collections.Set`) – other set

Returns a new set with elements in either the set or the `operand` but not both

Return type `set`

add(*element*)

Adds an `element` to the set. This has no effect if the `element` is already present.

Parameters **element** – an element to add

Note: This method is a direct mapping to SADD comamnd.

clear()

Removes all elements from this set.

Note: Under the hood it simply DEL the key.

difference(**sets*)

Returns the difference of two or more `sets` as a new `set` i.e. all elements that are in this set but not the others.

Parameters **sets** – other iterables to get the difference

Returns the relative complement

Return type `set`

Note: This method is mapped to SDIFF command.

difference_update(**sets*)

Removes all elements of other `sets` from this set.

Parameters ***sets** – other sets that have elements to remove from this set

Note: For `Set` objects of the same session it internally uses SDIFFSTORE command.

For other ordinary Python iterables, it uses SREM commands. If the version of Redis is less than 2.4, sends SREM multiple times. Because multiple operands of SREM command has been supported since Redis 2.4.

discard(*element*)

Removes an `element` from the set if it is a member. If the `element` is not a member, does nothing.

Parameters **element** – an element to remove

Note: This method is mapped to SREM command.

intersection(**sets*)

Gets the intersection of the given sets.

Parameters ***sets** – zero or more operand sets to get intersection. all these must be iterable

Returns the intersection

Return type `set`

intersection_update(**sets*)

Updates the set with the intersection of itself and other `sets`.

Parameters ***sets** – zero or more operand sets to intersection. all these must be iterable

Note: It sends a SINTERSTORE command for other `Set` objects and a SREM command for other ordinary Python iterables.

Multiple operands of SREM command has been supported since Redis 2.4.0, so it would send multiple SREM commands if the Redis version is less than 2.4.0.

Used commands: SINTERSTORE, SMEMBERS and SREM.

---

isdisjoint(*operand*)

Tests whether two sets are disjoint or not.

Parameters **operand** (`collections.Iterable`) – another set to test

Returns `True` if two sets have a null intersection

Return type `bool`

Note: It internally uses SINTER command.

---

issubset(*operand*)

Tests whether the set is a subset of the given `operand` or not. To test proper (strict) subset, use < operator instead.

Parameters **operand** (`collections.Iterable`) – another set to test

Returns `True` if the `operand` set contains the set

Return type `bool`

Note: This method consists of following Redis commands:

1. SDIFF for this set and `operand`
2. SLEN for this set
3. SLEN for `operand`

If the first SDIFF returns anything, it sends no commands of the rest and simply returns `False`.

---

issuperset(*operand*)

Tests whether the set is a superset of the given `operand`. To test proper (strict) superset, use > operator instead.

Parameters **operand** (`collections.Iterable`) – another set to test

Returns `True` if the set contains `operand`

Return type `bool`

pop()

Removes an arbitrary element from the set and returns it. Raises `KeyError` if the set is empty.

Returns a removed arbitrary element

Raises **exceptions.KeyError** if the set is empty

Note: This method is directly mapped to SPOP command.

---

symmetric_difference(*operand*)

Returns a new set with elements in either the set or the `operand` but not both.

Parameters **operand** (`collections.Iterable`) – other set

Returns a new set with elements in either the set or the `operand` but not both

Return type `set`

Note: This method consists of following two commands:

1. SUNION of this set and the `operand`
2. SINTER of this set and the `operand`

and then makes a new `set` with elements in the first result are that are not in the second result.

---

symmetric_difference_update(*operand*)

Updates the set with the symmetric difference of itself and `operand`.

Parameters **operand** (`collections.Iterable`) – another set to get symmetric difference

---

Note: This method consists of several Redis commands in a transaction: SINTER, SUNIONSTORE and SREM.

---

union(**sets*)

Gets the union of the given sets.

Parameters ***sets** – zero or more operand sets to union. all these must be iterable

Returns the union set

Return type `set`

Note: It sends a SUNION command for other `Set` objects. For other ordinary Python iterables, it unions all in the memory.

---

update(**sets*)

Updates the set with union of itself and operands.

Parameters ***sets** – zero or more operand sets to union. all these must be iterable

---

Note: It sends a SUNIONSTORE command for other `Set` objects and a SADD command for other ordinary Python iterables.

Multiple operands of SADD command has been supported since Redis 2.4.0, so it would send multiple SADD commands if the Redis version is less than 2.4.0.

1.1.5 `sider.list` — List objects

class `sider.list.`**List**(*session*, *key*, *value_type=<class 'sider.types.ByteString'>*)

The Python-side representaion of Redis list value. It behaves alike built-in Python `list` object. More exactly, it implements `collections.MutableSequence` protocol.

append(*value*)

Inserts the `value` at the tail of the list.

Parameters **value** – an value to insert

extend(*iterable*)

Extends the list with the `iterable`.

Parameters **iterable** (`collections.Iterable`) – an iterable object that extend the list with

Raises **exceptions.TypeError** if the given `iterable` is not iterable

> **Warning:** Appending multiple values is supported since Redis 2.4.0. This may send RPUSH multiple times in a transaction if Redis version is not 2.4.0 nor higher.

insert(*index*, *value*)

Inserts the `value` right after the offset `index`.

Parameters

- **index** (`numbers.Integral`) – the offset of the next before the place where `value` would be inserted to
- **value** – the value to insert

Raises **exceptions.TypeError** if the given `index` is not an integer

> **Warning:** Redis does not provide any primitive operations for random insertion. You only can prepend or append a value into lists. If index is 0 it'll send LPUSH command or if index is -1 it'll send RPUSH command, but otherwise it inefficiently LRANGE the whole list to manipulate it in offline, and then DEL the key so that empty the whole list, and then RPUSH the whole result again. Moreover all the commands execute in a transaction.
> So you should not treat this method as the same method of Python built-in `list` object. It is just for being compatible to `collections.MutableSequence` protocol.
> If it faced the case, it also will warn you `PerformanceWarning`.

pop(*index=-1*, *_stacklevel=1*)

Removes and returns an item at `index`. Negative index means `len(list) + index` (counts from the last).

Parameters

- **index** (`numbers.Integral`) – an index of an item to remove and return
- **_stacklevel** (`numbers.Integral`) – internal use only. base stacklevel for warning. default is 1

Returns the removed element

Raises

- **exceptions.IndexError** – if the given `index` doesn't exist
- **exceptions.TypeError** – if the given `index` is not an integer

> **Warning:** Redis doesn't offer any primitive operations for random deletion. You can pop only the last or the first. Other middle elements cannot be popped in a command, so it emulates the operation inefficiently.
> Internal emulation routine to pop an other index than -1 or 0 consists of three commands in a transaction:
> •LINDEX
> •LTRIM
> •RPUSH (In worst case, this command would be sent N times where N is the cardinality of elements placed after popped index. Because multiple operands for RPUSH was supported since Redis 2.4.0.)
> So you should not treat this method as the same method of Python built-in `list` object. It is just for being compatible to `collections.MutableSequence` protocol.
> If it faced the case, it also will warn you `PerformanceWarning`.

value_type = None
(sider.types.Bulk) The type of list values.

1.1.6 sider.datetime — Date and time related utilities

For minimum support of time zones, without adding any external dependencies e.g. pytz, Sider had to implement Utc class which is a subtype of datetime.tzinfo.

Because datetime module provided by the Python standard library doesn't contain UTC or any other tzinfo subtype implementations. (A funny thing is that the documentation of datetime module shows an example of how to implement UTC tzinfo.)

If you want more various time zones support use the third-party pytz package.

class sider.datetime.**FixedOffset**(*offset*, *name=None*)
Fixed offset in minutes east from UTC.

```
>>> import datetime
>>> day = FixedOffset(datetime.timedelta(days=1))
>>> day
sider.datetime.FixedOffset(1440)
>>> day.tzname(None)
'+24:00'
>>> half = FixedOffset(-720)
>>> half
sider.datetime.FixedOffset(-720)
>>> half.tzname(None)
'-12:00'
>>> half.utcoffset(None)
datetime.timedelta(-1, 43200)
>>> zero = FixedOffset(0)
>>> zero.tzname(None)
'UTC'
>>> zero.utcoffset(None)
datetime.timedelta(0)
```

Parameters

- **offset** (numbers.Integral, datetime.timedelta) – the offset integer in minutes, or timedelta (from a minute to a day)
- **name** (basestring) – an optional name. if not present, automatically generated

Raises exceptions.ValueError when offset's precision is too short or too long

MAX_PRECISION = datetime.timedelta(1)
(datetime.timedelta) The maximum precision of utcoffset().

MIN_PRECISION = datetime.timedelta(0, 60)
(datetime.timedelta) The minimum precision of utcoffset().

sider.datetime.**UTC = sider.datetime.Utc()**
(Utc) The singleton instance of Utc.

class sider.datetime.**Utc**
The datetime.tzinfo implementation of UTC.

```
>>> from datetime import datetime
>>> utc = Utc()
```

```
>>> dt = datetime(2012, 3, 15, 0, 15, 30, tzinfo=utc)
>>> dt
datetime.datetime(2012, 3, 15, 0, 15, 30, tzinfo=sider.datetime.Utc())
>>> utc.utcoffset(dt)
datetime.timedelta(0)
>>> utc.dst(dt)
datetime.timedelta(0)
>>> utc.tzname(dt)
'UTC'
```

`sider.datetime.`**`ZERO_DELTA`** **= datetime.timedelta(0)**
(`datetime.timedelta`) No difference.

`sider.datetime.`**`total_seconds`**(*timedelta*)
For Python 2.6 compatibility. Equivalent to `timedelta.total_seconds()` method which was introduced in Python 2.7.

Parameters **timedelta** (`datetime.timedelta`) – the timedelta

Returns the total number of seconds contained in the duration

`sider.datetime.`**`utcnow`**()
The current time in `UTC`. The Python standard library also provides `datetime.datetime.utcnow()` function except it returns a naive `datetime.datetime` value. This function returns tz-aware `datetime.datetime` value instead.

```
>>> import datetime
>>> datetime.datetime.utcnow()
datetime.datetime(...)
>>> utcnow()
datetime.datetime(..., tzinfo=sider.datetime.Utc())
```

Returns the tz-aware `datetime` value of the current time

Return type `datetime.datetime`

1.1.7 `sider.warnings` — Warning categories

This module defines several custom warning category classes.

exception `sider.warnings.`**`PerformanceWarning`**
The category for warnings about performance worries. Operations that warn this category would work but be inefficient.

1.1.8 `sider.lazyimport` — Lazily imported modules

Provides a `types.ModuleType`-like proxy object for submodules of the `sider` package. These are for workaround circular importing.

class `sider.lazyimport.`**`DeferredModule`**(**args*, ***kwargs*)
The deferred version of `types.ModuleType`. Under the hood it imports the actual module when it actually has to.

`sider.lazyimport.`**`session`** **= <deferred module 'sider.session'>**
(`DeferredModule`) Alias of `sider.session`.

`sider.lazyimport.`**`set`** **= <deferred module 'sider.set'>**
(`DeferredModule`) Alias of `sider.set`.

`sider.lazyimport.`**`hash`** **= <deferred module 'sider.hash'>**
(`DeferredModule`) Alias of `sider.hash`.

`sider.lazyimport.`**`version`** **= <deferred module 'sider.version'>**
(`DeferredModule`) Alias of `sider.version`.

`sider.lazyimport.`**`warnings`** **= <deferred module 'sider.warnings'>**
(`DeferredModule`) Alias of `sider.warnings`.

`sider.lazyimport.`**`list`** **= <deferred module 'sider.list'>**
(`DeferredModule`) Alias of `sider.list`.

`sider.lazyimport.`**`datetime`** **= <deferred module 'sider.datetime'>**
(`DeferredModule`) Alias of `sider.datetime`.

`sider.lazyimport.`**`types`** **= <deferred module 'sider.types'>**
(`DeferredModule`) Alias of `sider.types`.

1.1.9 `sider.version` — Version data

`sider.version.`**`VERSION`** **= '0.1.1'**
(`str`) The version string e.g. `'1.2.3'`.

`sider.version.`**`VERSION_INFO`** **= (0, 1, 1)**
(`tuple`) The triple of version numbers e.g. `(1, 2, 3)`.

CHAPTER

TWO

FURTHER READING

2.1 Documentation guides

This project use Sphinx for documentation and Read the Docs for documentation hosting. Build the documentation always before you commit — You must not miss documentation of your contributed code.

Be fluent in reStructuredText.

2.1.1 Build

Install Sphinx 1.1 or higher first. If it's been installed already, skip this.

```
$ easy_install "Sphinx>=1.1"
```

Use **make** in the `docs/` directory.

```
$ cd docs/
$ make html
```

You can find the built documentation in the `docs/_build/html/` directory.

```
$ python -m webbrowser docs/_build/html/  # in the root
```

2.1.2 Convention

- Follow styles as it was.
- Every module/package has to start with docstring like this:

  ```
  """:mod:`sider.modulename` --- Module title
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  Short description about the module.

  """
  ```

  and make reStructuredText file of the same name in the `docs/sider/` directory. Use `automodule` directive.
- All published modules, constants, functions, classes, methods and attributes (properties) have to be documented in their docstrings.
- Source code to quote is in Python, use a literal block. If the code is a Python interactive console session, don't use it (see below).

- The source code is not in Python, use a `sourcecode` directive provided by Sphinx. For example, if the code is a Python interactive console session:

  ```
  .. sourcecode:: pycon

     >>> 1 + 1
     2
  ```

  See also the list of Pygments lexers.

- Link Redis commands using `redis` role. For example:

  ```
  It may send :redis:`RPUSH` multiple times.
  ```

2.2 Design memo

2.2.1 Directions

- Do not reinvent the wheel. Use redis-py for connection pooling. It already is mature.
- Don't be implicit. Hashes aren't entities. Hash keys aren't fields. Connections aren't sessions.

2.2.2 Example

Schema

```
from sider.entity import Entity, Field
from sider.types import UnicodeString, Date, TZDateTime
from sider.datetime import now
from .password import Password


class User(Entity):
    """User entity."""

    login = Field(UnicodeString, required=True, key=True)
    password = Field(UnicodeString, required=True)
    name = Field(UnicodeString, required=True)
    url = Field(UnicodeString, unique=True)
    dob = Field(Date)
    created_at = Field(TZDateTime, required=True, default=now)

    @login.before_set
    def login(self, value):
        value = value.strip().lower()
        if 2 < len(value) < 50:
            return value
        raise ValueError('invalid login')

    @password.before_set
    def password(self, value):
        return Password.hash(self, value)

    @password.after_get
    def password(self, value):
        return Password(self, value)
```

```
    def __unicode__(self):
        return getattr(self, 'user', None) or u''
```

Query

```
>>> from redis import Redis
>>> from sider.session import Session
>>> from myapp.user import User
>>> session = Session(Redis(host='127.0.0.1', port=6379, db=0))
>>> user = session.get(User, 'hongminhee')
>>> user
<myapp.user.User 'users:hongminhee'>
>>> user.password
<myapp.password.Password user='hongminhee'>
```

2.3 To do list

2.3.1 To be added

- `sider.sortedset`

2.3.2 To be fixed

2.4 Sider Changelog

2.4.1 Version 0.1.1

To be released.

- Added `sider.types.Boolean` type.
- Added `sider.types.Date` type.
- Added `sider.datetime.FixedOffset` tzinfo subtype.
- Added `sider.types.DateTime` and `TZDateTime` types.
- Now you can check the version by this command: `python -m sider.version`.

2.4.2 Version 0.1.0

Released on March 23, 2012. Pre-alpha release.

CHAPTER

THREE

OPEN SOURCE

Sider is an open source software written in Hong Minhee. The source code is distributed under MIT license and you can find it at Bitbucket repository. Check out now:

```
$ hg clone https://bitbucket.org/dahlia/sider
```

If you find a bug, report it to the issue tracker or send pull requests.

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S