Sider Documentation

Release 0.2.0

Hong Minhee

April 29, 2012

CONTENTS

Sider is a persistent object library based on Redis. This is heavily under development currently, but but you can check the future roadmap if you want.

```
>>> 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
0.2.0
```

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.StrictRedis) – the Redis client

current_transaction

(Transaction) The current transaction. It could be None when it's not on any transaction.

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

mark_manipulative(*keys=frozenset([])*)

Marks it is manipulative.

Parameters **keys** (collections.Iterable) – optional set of keys to watch

---

Note: This method is for internal use.

---

mark_query(*keys=frozenset([])*)

Marks it is querying.

Parameters **keys** (collections.Iterable) – optional set of keys to watch

Raises sider.exceptions.CommitError when it is tried during commit phase

---

Note: This method is for internal use.

---

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

transaction
(sider.transaction.Transaction) The transaction object for the session.

Transaction objects are callable and so you can use this transaction property as like a method:

```
def block(trial, transaction):
    list_[0] = list[0].upper()
session.transaction(block)
```

Or you can use it in a for loop:

```
for trial in session.transaction:
    list_[0] = list[0].upper()
```

See Also:

Method sider.transaction.Transaction.__call__() Executes a given block in the transaction.

Method sider.transaction.Transaction.__iter__() More explicit way to execute a routine in the transaction than Transaction.__call__()

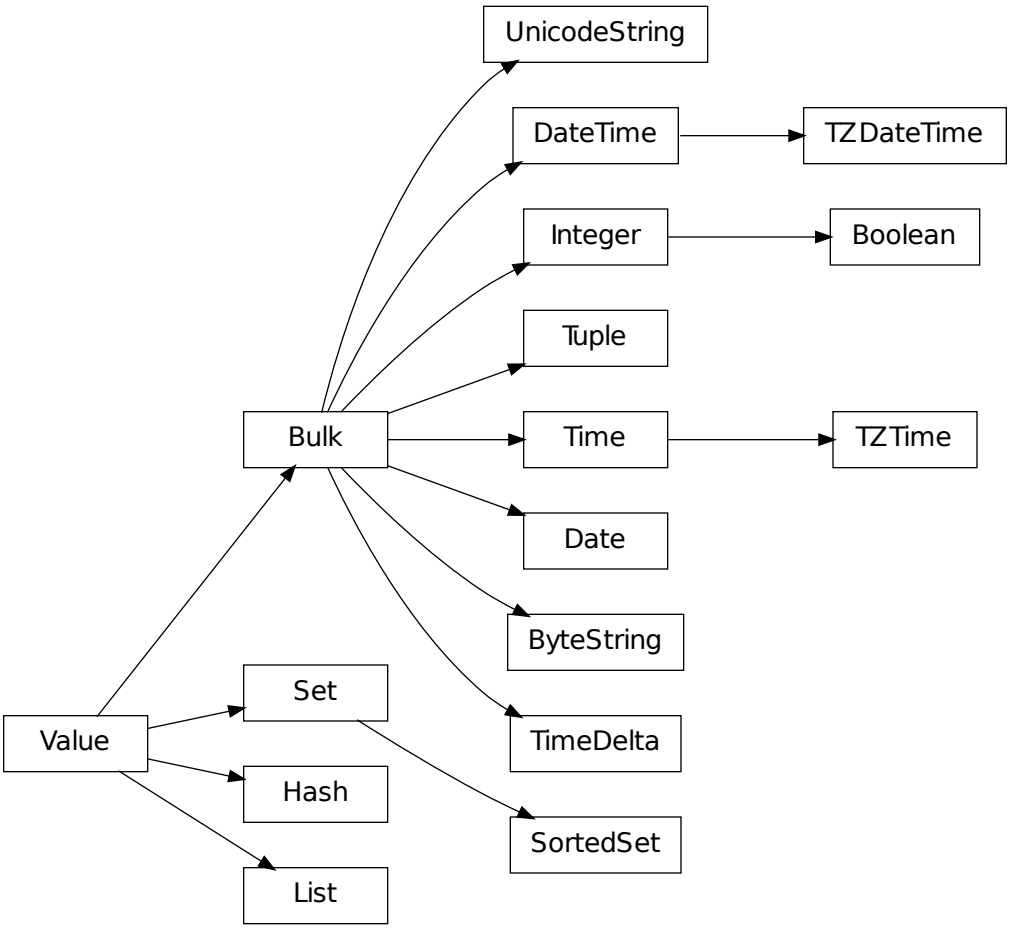
verbose_transaction_error = None
(bool) If it is set to True, error messages raised by transactions will contain tracebacks where they started query/commit phase.

It is mostly for debug purpose, and you can set this to True if it's needed.

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 0x368eda0>
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 0x36b9ea0>

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 **RFC 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.**SortedSet**(*value_type=None*)
Bases: sider.types.Set

The type object for sider.sortedset.SortedSet objects.

Parameters **value_type** (Bulk, type) – the type of values the sorted 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.**TZTime**
Bases: sider.types.Time

Similar to Time except it accepts only tz-aware datetime.time values.

```
>>> from sider.datetime import FixedOffset
>>> time = datetime.time(18, 21, 34, 638972,
...                      tzinfo=FixedOffset(540))
>>> tztime = TZTime()
>>> tztime.encode(time)
'18:21:34.638972+09:00'
>>> tztime.decode(_)
datetime.time(18, 21, 34, 638972,
              tzinfo=sider.datetime.FixedOffset(540))
>>> utctime = datetime.time(9, 21, 34, 638972, tzinfo=UTC)
>>> tztime.encode(utctime)
'09:21:34.638972Z'
```

```
>>> tztime.decode(_)
datetime.time(9, 21, 34, 638972, tzinfo=sider.datetime.Utc())
```

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

class `sider.types.`**Time**
Bases: `sider.types.Bulk`

Stores naive `datetime.time` values.

```
>>> time = Time()
>>> time.decode('09:21:34.638972')
datetime.time(9, 21, 34, 638972)
>>> time.encode(_)
'09:21:34.638972'
```

It doesn't store `tzinfo` data.

```
>>> from sider.datetime import UTC
>>> time = Time()
>>> decoded = time.decode('09:21:34.638972Z')
>>> decoded
datetime.time(9, 21, 34, 638972)
>>> time.encode(decoded.replace(tzinfo=UTC))
'09:21:34.638972'
```

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

TIME_PATTERN = <_sre.SRE_Pattern object at 0x36bc650>

parse_time(*bulk*, *drop_tzinfo*)
Parses an encoded `datetime.time`.

Parameters **bulk** (`basestring`) – an encoded time

Returns a parsed time

Return type `datetime.time`

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

---

class `sider.types.`**TimeDelta**
Bases: `sider.types.Bulk`

Stores `datetime.timedelta` values.

```
>>> import datetime
>>> td = TimeDelta()
>>> delta = datetime.timedelta(days=3, seconds=53, microseconds=123123)
>>> td.encode(delta)
'3,53,123123'
>>> td.decode(_)
datetime.timedelta(3, 53, 123123)
```

TIMEDELTA_PATTERN = <_sre.SRE_Pattern object at 0x36903c0>

class `sider.types.`**Tuple**(**field_types*)
Bases: `sider.types.Bulk`

Stores tuples of fixed fields. It can be used for compositing multiple fields into one field in ad-hoc way. For example, if you want to store 3D point value without defining new `Type`:

```
Tuple(Integer, Integer, Integer)
```

The above type will store three integers in a field.

```
>>> int_str_int = Tuple(Integer, ByteString, Integer)
>>> int_str_int.encode((123, 'abc\ndef', 456))
'3,7,3\n123\nabc\ndef\n456'
>>> int_str_int.decode(_)
(123, 'abc\ndef', 456)
```

Encoded values become a bulk bytes. It consists of a header line and other lines that contain field values. The first header line is a comma-separated integers that represent each byte size of encoded field values.

```
tuple   ::=   header (newline field)*
header  ::=   [size (“,” size)*]
size    ::=   digit+
digit   ::=   “0” | “1” | “2” | “3” | “4” |
              “5” | “6” | “7” | “8” | “9”
```

Parameters ***field_types** – the variable number of field types

field_types = None
(`tuple`) The tuple of field types.

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

See Also:

Redis Data Types The Redis documentation that explains about its data types: strings, lists, sets, sorted sets and hashes.

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__(**args*, ***kwargs*)
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__(**args*, ***kwargs*)
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__(**args*, ***kwargs*)
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__(**args*, ***kwargs*)
Gets the number of items.

Returns the number of items

Return type `numbers.Integral`

Note: It is directly mapped to Redis HLEN command.

__setitem__(**args*, ***kwargs*)
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(**args*, ***kwargs*)
Removes all items from this hash.

Note: Under the hood it simply DEL the key.

---

items(**args*, ***kwargs*)
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(**args*, ***kwargs*)

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(**args*, ***kwargs*)

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.list` — List objects

See Also:

Redis Data Types The Redis documentation that explains about its data types: strings, lists, sets, sorted sets and hashes.

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.

Table 1.2: Mappings of Redis commands–List methods

Redis commands	`List` methods
LLEN	`len()` (`List.__len__()`)
LPUSH	`List.insert()`
LPUSHX	N/A
LPOP	`List.pop()`
RPUSH	`List.append()`, `List.extend()`
RPUSHX	N/A
RPOP	`List.pop()`
RPOPLPUSH	N/A
LINDEX	`List.__getitem__()`,
LINSERT	N/A
LRANGE	`iter()` (`List.__iter__()`), `List.__getitem__()`,
LREM	N/A
LTRIM	`del` (`List.__delitem__()`)
DEL	`del` (`List.__delitem__()`)
LSET	`=` (`List.__setitem__()`)
BLPOP	N/A
BRPOP	N/A
BRPOPLPUSH	N/A

`append`(**args*, ***kwargs*)

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, 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.

1.1.5 `sider.set` — Set objects

See Also:

Redis Data Types The Redis documentation that explains about its data types: strings, lists, sets, sorted sets and hashes.

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 1.3: 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__(**args*, ***kwargs*)

`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__(**args*, ***kwargs*)
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(**args*, ***kwargs*)
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(**args*, ***kwargs*)
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(**args*, ***kwargs*)

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.

1.1.6 `sider.sortedset` — Sorted sets

See Also:

Redis Data Types The Redis documentation that explains about its data types: strings, lists, sets, sorted sets and hashes.

class `sider.sortedset.`**SortedSet**(*session*, *key*, *value_type=<class 'sider.types.ByteString'>*)
The Python-sider representation of Redis sorted set value. It behaves in similar way to `collections.Counter` object which became a part of standard library since Python 2.7.

It implements `collections.MutableMapping` and `collections.MutableSet` protocols.

Table 1.4: Mappings of Redis commands–SortedSet methods

Redis commands	`SortedSet` methods
DEL	`SortedSet.clear()`
ZADD	= (`SortedSet.__setitem__()`)
ZCARD	`len()` (`SortedSet.__len__()`)
ZINCRBY	`SortedSet.add()`, `SortedSet.discard()`, `SortedSet.update()`
ZRANGE	`iter()` (`SortedSet.__iter__()`)
ZRANGE WITHSCORES	`SortedSet.items()`, `SortedSet.most_common()`, `SortedSet.least_common()`
ZREM	`del` (`SortedSet.__delitem__()`), `SortedSet.discard()`
ZSCORE	`SortedSet.__getitem__()`, `in` (`SortedSet.__contains__()`)
ZUNIONSTORE	`SortedSet.update()`
N/A	`SortedSet.setdefault()`
N/A	`SortedSet.pop()`
N/A	`SortedSet.popitem()`

__contains__(**args*, ***kwargs*)
`in` operator. Tests whether the set contains the given operand `member`.

Parameters **member** – the value to test

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

Return type `bool`

Note: This method internally uses ZSCORE command.

__delitem__(*member*)
Removes the `member`.

Parameters **member** – the member to delete

Raises

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

Note: It is directly mapped to Redis ZREM command when it's not on transaction. If it's used with transaction, it internally uses ZSCORE and ZREM commands.

__getitem__(**args*, ***kwargs*)
Gets the score of the given `member`.

Parameters **member** – the member to get its score

Returns the score of the `member`

Return type `numbers.Real`

Raises

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

Note: It is directly mapped to Redis ZSCORE command.

__len__(**args*, ***kwargs*)
Gets the cardinality of the sorted set.

Returns the cardinality (the number of elements) of the sorted set

Return type `numbers.Integral`

Note: It is directly mapped to Redis ZCARD command.

__setitem__(**args*, ***kwargs*)
Sets the `score` of the `member`. Adds the `member` if it doesn't exist.

Parameters

- **member** – the member to set its `score`
- **scorew** – the score to set of the `member`

Raises **exceptions.TypeError** if the given `member` is not acceptable by its `value_type` or the given `score` is not a `numbers.Real` object

---

Note: It is directly mapped to Redis ZADD command.

---

add(**args*, ***kwargs*)
Adds a new `member` or increases its `score` (default is 1).

Parameters

- **member** – the member to add or increase its score

- **score** (numbers.Real) – the amount to increase the score. default is 1

Note: This method is directly mapped to ZINCRBY command.

---

clear(**args*, ***kwargs*)
Removes all values from this sorted set.

Note: Under the hood it simply DEL the key.

---

discard(*member*, *score=1*, *remove=0*)
Opposite operation of add(). It decreases its score (default is 1). When its score get the remove number (default is 0) or less, it will be removed.

If you don't want to remove it but only decrease its score, pass None into remove parameter.

If you want to remove member, not only decrease its score, use __delitem__() instead:

```
del sortedset[member]
```

Parameters

- **member** – the member to decreases its score
- **score** (numbers.Real) – the amount to decrease the score. default is 1
- **remove** (numbers.Real) – the threshold score to be removed. if None is passed, it doesn't remove the member but only decreases its score (it makes score argument meaningless). default is 0.

Note: This method is directly mapped to ZINCRBY command when remove is None.

Otherwise, it internally uses ZSCORE plus ZINCRBY or *:redis:'ZREM* (total two commands) within a transaction.

---

items(**args*, ***kwargs*)
Returns an ordered of pairs of elements and these scores.

Parameters **reverse** (bool) – order result descendingly if it's True. default is False which means ascending order

Returns an ordered list of pairs. every pair looks like (element, score)

Return type collections.Sequence

Note: This method is directly mapped to ZRANGE command and WITHSCORES option.

---

keys(*reverse=False*)
Gets its all elements. Equivalent to __iter__() except it returns an ordered Sequence instead of iterable.

Parameters **reverse** (bool) – order result descendingly if it's True. default is False which means ascending order

Returns the ordered list of its all keys

Return type collections.Sequence

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

---

least_common(**args*, ***kwargs*)

Returns a list of the `n` least common (exactly, lowly scored) members and their counts (scores) from the least common to the most. If `n` is not specified, it returns *all* members in the set. Members with equal scores are ordered arbitarily.

Parameters **n** (`numbers.Integral`) – the number of members to get

Returns an ordered list of pairs. every pair looks like (element, score)

Return type `collections.Sequence`

Note: This method is directly mapped to ZREVRANGE command and `WITHSCORES` option.

---

most_common(*n=None*, *reverse=False*)

Returns a list of the `n` most common (exactly, highly scored) members and their counts (scores) from the most common to the least. If `n` is not specified, it returns *all* members in the set. Members with equal scores are ordered arbitarily.

Parameters **n** (`numbers.Integral`) – the number of members to get

Returns an ordered list of pairs. every pair looks like (element, score)

Return type `collections.Sequence`

Note: This method is directly mapped to ZRANGE command and `WITHSCORES` option.

---

pop(**args*, ***kwargs*)

Populates a member of the set.

If `key` keyword argument or one or more positional arguments have present, it behaves like `dict.pop()` method:

Parameters

- **key** – the member to populate. it will be removed if it exists
- **default** – the default value returned instead of the member (`key`) when it doesn't exist. default is `None`

Returns the score of the member before the operation has been committed

Return type `numbers.Real`

Note: It internally uses ZSCORE, ZREM or ZINCRBY (total 2 commands) in a transaction.

If no positional arguments or no `key` keyword argument, it behaves like `set.pop()` method. Basically it does the same thing with `popitem()` except it returns just a popped value (while `popitem()` returns a pair of popped value and its score).

Parameters **desc** (`bool`) – keyword only. by default, it populates the member of the lowest score, but if you pass `True` to this it will populates the highest instead. default is `False`

Returns the populated member. it will be the lowest scored member or the highest scored member if `desc` is `True`

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

Note: It internally uses ZRANGE or ZREVRANGE, ZREM or ZINCRBY (total 2 commands) in a transaction.

See Also:

Method `popitem()`

If any case there are common keyword-only parameters:

Parameters

- **score** (`numbers.Real`) – keyword only. the amount to decrease the score. default is 1
- **remove** (`numbers.Real`) – keyword only. the threshold score to be removed. if `None` is passed, it doesn't remove the member but only decreases its score (it makes `score` argument meaningless). default is 0.

popitem(*desc=False*, *score=1*, *remove=0*)

Populates the lowest scored member (or the highest if `desc` is `True`) and its score.

It returns a pair of the populated member and its score. The score is a value before the operation has been committed.

Parameters

- **desc** (`bool`) – by default, it populates the member of the lowest score, but if you pass `True` to this parameter it will populates the highest instead. default is `False`
- **score** (`numbers.Real`) – the amount to decrease the score. default is 1
- **remove** (`numbers.Real`) – the threshold score to be removed. if `None` is passed, it doesn't remove the member but only decreases its score (it makes `score` argument meaningless). default is 0.

Returns a pair of the populated member and its score. the first part of a pair will be the lowest scored member or the highest scored member if `desc` is `True`. the second part of a pair will be the score before the operation has been committed

Return type `tuple`

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

Note: It internally uses ZRANGE or ZREVRANGE, ZREM or ZINCRBY (total 2 commands) in a transaction.

See Also:

Method `pop()`

setdefault(*key*, *default=1*)

Gets the score of the given `key` if it exists or adds `key` with `default` score.

Parameters

- **key** – the member to get its score
- **default** (`numbers.Real`) – the score to be set if the `key` doesn't exist. default is 1

Returns the score of the `key` after the operation has been committed

Return type `numbers.Real`

Note: It internally uses one or two commands. At first it sends ZSCORE command to check whether the key exists and get its score if it exists. If it doesn't exist yet, it sends one more command: ZADD. It is atomically committed in a transaction.

---

update(**sets*, ***keywords*)

Merge with passed sets and keywords. It's behavior is almost equivalent to `dict.update()` and `set.update()` except it's aware of scores.

For example, assume the initial elements and their scores of the set is (in notation of dictionary):

```
{'c': 1, 'a': 2, 'b': 3}
```

and you has updated it:

```
sortedset.update(set('acd'))
```

then it becomes (in notation of dictionary):

```
{'d': 1, 'c': 2, 'a': 3, 'b': 3}
```

You can pass mapping objects or keywords instead to specify scores to increment:

```
sortedset.update({'a': 1, 'b': 2})
sortedset.update(a=1, b=2)
sortedset.update(set('ab'), set('cd'),
                 {'a': 1, 'b': 2}, {'c': 1, 'd': 2},
                 a=1, b=2, c=1, d=2)
```

Parameters

- ***sets** – sets or mapping objects to merge with. mapping objects can specify scores by values
- ****keywords** – if `value_type` takes byte strings you can specify elements and its scores by keyword arguments

---

Note: There's an incompatibility with `dict.update()`. It always treats iterable of pairs as set of pairs, not mapping pairs, unlike `dict.update()`. It is for resolving ambiguity (remember `value_type` can take tuples or such things).

Note: Under the hood it uses multiple ZINCRBY commands and ZUNIONSTORE if there are one or more `SortedSet` objects in operands.

---

value_type = None

(`sider.types.Bulk`) The type of set elements.

values(**args*, ***kwargs*)

Returns an ordered list of scores.

Parameters **reverse** (`bool`) – order result descendingly if it's `True`. default is `False` which means ascending order

Returns an ordered list of scores

Return type `collections.Sequence`

Note: This method internally uses ZRANGE command and `WITHSCORES` option.

1.1.7 `sider.transaction` — Transaction handling

Every Persist object provided by Sider can be used within transactions. You can atomically commit multiple operations.

Under the hood, transaction blocks are simply looped until objects the transaction deals with haven't been faced any conflicts with other sessions/transactions. If there are no concurrent touches to `names` in the following transaction:

```
def block(trial, transaction):
    names.append(new_name)
session.transaction(block)
```

it will be successfully committed. Otherwise, it retries the whole transaction `block`. You can easily prove this by just printing `trial` (the first argument of the block function) inside the transaction block. It will print one or more retrial counting numbers.

This means you shouldn't do I/O in the transaction block. Your I/O could be executed two or more times. Do I/O after or before transaction blocks instead.

There are two properties of every operation: `query()` or `manipulative()` or both. For example, `Hash.get()` method is a query operation. On the other hand, `Set.add()` method is manipulative. There is a rule of transaction: query operations can't be used after manipulative operations. For example, the following transaction block has no problem:

```
# Atomically wraps an existing string value of the specific
# key of a hash.
hash_ = session.get('my_hash', Hash)
def block(trial, transaction):
    current_value = hash_['my_key']  # [query operation]
    updated_value = '(' + current_value + ')'
    hash_['my_key'] = updated_value  # [manipulative operation]
session.transaction(block)
```

while the following raises a `CommitError`:

```
hash_ = session.get('my_hash', Hash)
def block(trial, transaction):
    current_value = hash_['my_key']   # [query operation]
    updated_value = '(' + current_value + ')'
    hash_['my_key'] = updated_value   # [manipulative operation]
    # The following statement raises CommitError because
    # it contains a query operation.
    current_value2 = hash_['my_key2'] # [query operation]
    updated_value2 = '(' + current_value2 + ')'
    hash_['my_key'] = updated_value2  # [manipulative operation]
session.transaction(block)
```

See Also:

Redis Transactions The Redis documentation that explains about its transactions.

class `sider.transaction.`**`Transaction`**(*session*, *keys=frozenset([])*)

Transaction block.

Parameters

- **session** (`Session`) – a session object
- **keys** (`collections.Iterable`) – the list of keys

__call__(*block*, *keys=frozenset([])*, *ignore_double=False*)

Executes a `block` in the transaction:

```
def block(trial, transaction):
    list_[0] = list_[0].upper()
transaction(block)
```

Parameters

- **block** (`collections.Callable`) – a function to execute in a transaction. see the signature explained in the below: `block()`
- **keys** (`collections.Iterable`) – a list of keys to watch
- **ignore_double** (`bool`) – don't raise any error even if any transaction has already being executed for a session. default is `False`

Raises sider.exceptions.DoubleTransactionError when any transaction has already being executed for a session and `ignore_double` is `False`

block(*trial*, *transaction*)

Parameters

- **trial** (`numbers.Integral`) – the number of trial count. starts from 0
- **transaction** (`Transaction`) – the current transaction object

__iter__()

You can more explictly execute (and retry) a routine in the transaction than using `__call__()`.

It returns a generator that yields an integer which represents its (re)trial count (from 0) until the transaction doesn't face `ConflictError`.

For example:

```
for trial in transaction:
    list_[0] = list_[0].upper()
```

Raises sider.exceptions.DoubleTransactionError when any transaction has already being executed for a session

__weakref__

list of weak references to the object (if defined)

begin_commit(*_stacklevel=1*)

Explicitly marks the transaction beginning to commit from this. From this to end of a transaction, any query operations will raise `CommitError`.

format_commit_stack(*indent=4*, *title='Traceback of previous begin_commit() call:'*)

Makes `commit_stack` text readable. If its `session.verbose_transaction_error` is not `True`, it will simply return an empty string.

Parameters

- **indent** (`numbers.Integral`) – the number of space character for indentation. default is 4
- **title** (`basestring`) – the title string of the formatted traceback. optional

Returns the formatted `commit_stack` text

Return type basestring

Note: It's totally for internal use.

---

format_enter_stack(*indent=4*, *title='Traceback where the transaction entered:'*)
Makes enter_stack text readable. If its session.verbose_transaction_error is not True, it will simply return an empty string.

Parameters

- **indent** (numbers.Integral) – the number of space character for indentation. default is 4
- **title** (basestring) – the title string of the formatted traceback. optional

Returns the formatted enter_stack text

Return type basestring

Note: It's totally for internal use.

---

watch(*keys*, *initialize=False*)
Watches more keys.

Parameters

- **keys** (collections.Iterable) – a set of keys to watch more
- **initialize** (bool) – initializes the set of watched keys if it is True. default is False. option only for internal use

sider.transaction.**manipulative**(*function*)
The decorator that marks the method manipulative.

Parameters **function** (collections.Callable) – the method to mark

Returns the marked method

Return type collections.Callable

sider.transaction.**query**(*function*)
The decorator that marks the method query.

Parameters **function** (collections.Callable) – the method to mark

Returns the marked method

Return type collections.Callable

1.1.8 sider.threadlocal — Thread locals

This module provides a small thread/greenlet local object. Why we don't simply use the built-in threading.local is there's the case of using greenlet, the non-standard but de facto standard coroutine module for Python, in real world. (For example, widely used networking libraries like gevent or eventlet heavily use greenlet.)

LocalDict which this module offers isn't aware of only threads but including greenlets.

Note: This module is inspired by werkzeug.local module but only things we actually need are left.

---

sider.threadlocal.**get_ident**()
Gets the object that can identify of the current thread/greenlet. It can return an object that can be used as dictionary keys.

Returns a something that can identify of the current thread or greenlet

---

Note: Under the hood it is an alias of `greenlet.getcurrent()` function if it is present. Or it will be aliased to `thread.get_ident()` or `dummy_thread.get_ident()` that both are a part of standard if `greenlet` module is not present.

However that's all only an implementation detail and so it may changed in the future. Client codes that use `sider.threadlocal.get_ident()` have to be written on only assumptions that it guarantees: *it returns an object that identify of the current thread/greenlet and be used as dictionary keys.*

---

class sider.threadlocal.**LocalDict**(*mapping*=[], ***keywords*)
A thread/greenlet-local dictionary. It implements `collections.MutableMapping` protocol and so behaves almost like built-in `dict` objects.

Parameters

- **mapping** (`collections.Mapping`, `collections.Iterable`) – the initial items. all locals have the same this initial items
- ****keywords** – the initial keywords. all locals have the same this initial items

1.1.9 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.

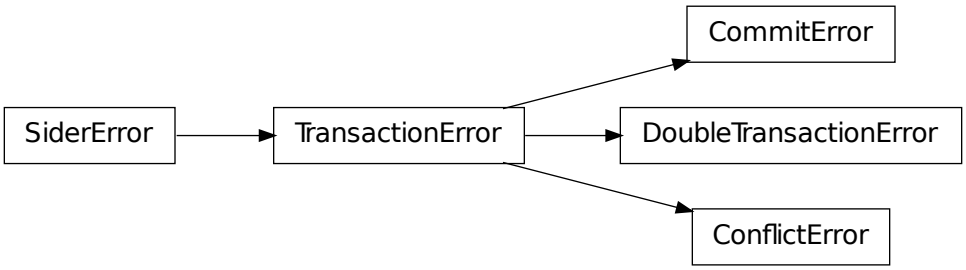
```
>>> import datetime
>>> datetime.datetime.utcnow()
datetime.datetime(...)
>>> utcnow()
datetime.datetime(..., tzinfo=sider.datetime.Utc())
```

Returns the tz-aware datetime value of the current time

Return type datetime.datetime

1.1.10 `sider.exceptions` — Exception classes

This module defines several custom exception classes.



exception `sider.exceptions.`**`CommitError`**
Bases: `sider.exceptions.TransactionError`

Error raised when any query operations are tried during commit phase.

exception `sider.exceptions.`**`ConflictError`**
Bases: `sider.exceptions.TransactionError`

Error rasied when the transaction has met conflicts.

exception `sider.exceptions.`**`DoubleTransactionError`**
Bases: `sider.exceptions.TransactionError`

Error raised when transactions are doubly tried for a session.

exception `sider.exceptions.`**`SiderError`**
Bases: `exceptions.Exception`

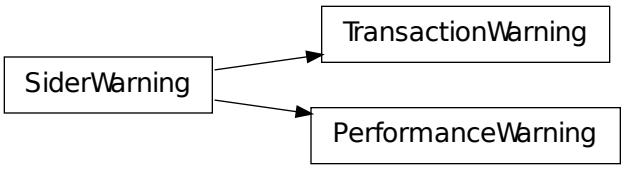
All exception classes Sider raises extend this base class.

exception `sider.exceptions.`**`TransactionError`**
Bases: `sider.exceptions.SiderError`

Transaction-related error.

1.1.11 `sider.warnings` — Warning categories

This module defines several custom warning category classes.



exception sider.warnings.**PerformanceWarning**
Bases: sider.warnings.SiderWarning, exceptions.RuntimeWarning

The category for warnings about performance worries. Operations that warn this category would work but be inefficient.

exception sider.warnings.**SiderWarning**
Bases: exceptions.Warning

All warning classes used by Sider extend this base class.

exception sider.warnings.**TransactionWarning**
Bases: sider.warnings.SiderWarning, exceptions.RuntimeWarning

The category for warnings about transactions.

1.1.12 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.**transaction = <deferred module 'sider.transaction'>**
(DeferredModule) Alias of sider.transaction.

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.**sortedset = <deferred module 'sider.sortedset'>**
(DeferredModule) Alias of sider.sortedset.

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

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

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

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

1.1.13 sider.ext — Extensions

This package is a *virtual* namespace package that forwards sider.ext.mycontrib to sider_mycontrib.

If you are writing a user-contributed module for Sider, simply name your module/package like sider_modulename and then it becomes importable by sider.ext.modulename.

1.1.14 sider.version — Version data

sider.version.**VERSION** = '0.2.0'
(str) The version string e.g. '1.2.3'.

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

CHAPTER

TWO

FURTHER READING

2.1 Examples

2.1.1 `sider.ext.wsgi_referer_stat` — Collecting referers using sorted sets

This tutorial will show you a basic example using sorted sets. We will build a small WSGI middleware that simply collects all *Referer*s of the given WSGI web application.

WSGI and middlewares

WSGI is a standard interface between web servers and Python web applications or frameworks to promote web application portability across a variety of web servers. (If you are from Java, think servlet. If you are from Ruby, think Rack.)

WSGI applications can be deployed into WSGI containers (server implementations). There are a lot of production-ready WSGI containers. Some of these are super fast, and some of others are very reliable. Check Green Unicorn, uWSGI, mod_wsgi, and so forth.

WSGI middleware is somewhat like decorator pattern for WSGI applications. Usually they are implemented using nested higher-order functions or classes with `__call__()` special method.

See Also:

To learn more details about WSGI, read **PEP 333** and other related resources. This tutorial doesn't deal with WSGI.

PEP 333 — Python Web Server Gateway Interface v1.0 This document specifies a proposed standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers.

Getting Started with WSGI by Armin Ronacher Armin Ronacher, the author of Flask, Werkzeug and Jinja, wrote this WSGI tutorial.

A Do-It-Yourself Framework by Ian Bicking Ian Bicking, the author of Paste, WebOb, lxml.html and FormEncode, explains about WSGI apps and middlewares.

Simple idea

The simple idea we'll implement here is to collect all *Referer* and store it into a persistent storage. We will use Redis as its persistent store. We want to increment the count for each *Referer*.

Stored data will be like:

Referer	Count
http://dahlia.kr/	1
https://bitbucket.org/dahlia/sider	3
https://twitter.com/hongminhee	6

We could use a hash here, but sorted set seems more suitable. Sorted sets are a data structure provided by Redis that is basically a set but able to represent duplications as its scores (ZINCRBY).

We can list a sorted set in asceding (ZRANGE) or descending order (ZREVRANGE) as well.

See Also:

Redis Data Types The Redis documentation that explains about its data types: strings, lists, sets, sorted sets and hashes.

Prototyping with using in-memory dictionary

First of all, we can implement a proof-of-concept prototype without Redis. Python has no sorted sets, so we will use `dict` instead.

```
class RefererStatMiddleware(object):
    '''A simple WSGI middleware that collects :mailheader:`Referer`
    headers.

    '''

    def __init__(self, application):
        assert callable(application)
        self.application = application
        self.referer_set = {}

    def __call__(self, environ, start_response):
        try:
            referer = environ['HTTP_REFERER']
        except KeyError:
            pass
        else:
            try:
                self.referer_set[referer] += 1
            except KeyError:
                self.referer_set[referer] = 1
        return self.application(environ, start_response)
```

It has some problems yet. What are that problems?

1. WSGI applications can be deployed into multiple server nodes, or forked to multiple processes as well. That means: `RefererStatMiddleware.referer_set` attribute can be split and not shared.
2. Increments of duplication counts aren't atomic.
3. Data will be lost when server process is terminated.

We can solve those problems by using Redis sorted sets instead of Python in-memorty `dict`.

Sider and persistent objects

It's a simple job, so we can deal with Redis commands by our hands. However it's a tutrial example of Sider. :-) We will use Sider's sorted set abstraction here instead. It's more abstracted away and easier to use!

Before touch our middleware code, the following session in Python interactive shell can make you understand basic of how to use Sider:

```
>>> from redis.client import StrictRedis
>>> from sider.session import Session
>>> from sider.types import SortedSet
>>> session = Session(StrictRedis())
>>> my_sorted_set = session.get('my_sorted_set', SortedSet)
>>> my_sorted_set
<sider.sortedset.SortedSet ('my_sorted_set') {}>
```

Note: Did you face `ImportError`?

```
>>> from redis.client import StrictRedis
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ImportError: No module named redis
```

You probably didn't install Python redis client library. You can install it through **pip**:

```
$ pip install redis
```

Or **easy_install**:

```
$ easy_install redis
```

Okay, here's an empty set: `my_sorted_set`. Let's add something to it.

```
>>> my_sorted_set
<sider.sortedset.SortedSet ('my_sorted_set') {}>
>>> my_sorted_set.add('http://dahlia.kr/')  # ZINCRBY
>>> my_sorted_set
<sider.sortedset.SortedSet ('my_sorted_set') {'http://dahlia.kr/'}>
```

Unlike Python's in-memory `set` or `dict`, it's a persistent object. In other words, `my_sorted_set` still contains `'kimchi'` even if you quit this session of Python interactive shell. Try yourself: type `exit()` to quit the session and enter **python** again. And then...

```
>>> my_sorted_set
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: global name 'my_sorted_set' is not defined
```

I didn't lie! You need to load the Sider session first.

```
>>> from redis.client import StrictRedis
>>> from sider.session import Session
>>> from sider.types import SortedSet
>>> client = StrictRedis()
>>> session = Session(client)
>>> my_sorted_set = session.get('my_sorted_set', SortedSet)
```

Then:

```
>>> my_sorted_set
<sider.sortedset.SortedSet ('my_sorted_set') {'http://dahlia.kr/'}>
```

Yeah!

Note that the following line:

```
>>> client = StrictRedis()
```

tries to connect to Redis server on localhost:6379 by default. There are `host` and `port` parameters to configure it.

```
>>> client = StrictRedis(host='localhost', port=6379)
```

Sorted sets

You can `update()` multiple values at a time:

```
>>> my_sorted_set.update(['https://bitbucket.org/dahlia/sider',
...                       'https://twitter.com/hongminhee'])  # ZINCRBY
>>> my_sorted_set
<sider.sortedset.SortedSet ('my_sorted_set')
 {'https://bitbucket.org/dahlia/sider', 'https://twitter.com/hongminhee',
  'http://dahlia.kr/'}>
>>> my_sorted_set.update(['http://dahlia.kr/',
...                       'https://twitter.com/hongminhee'])  # ZINCRBY
>>> my_sorted_set
<sider.sortedset.SortedSet ('my_sorted_set')
 {'https://bitbucket.org/dahlia/sider', 'https://twitter.com/hongminhee': 2.0,
  'http://dahlia.kr/': 2.0}>
>>> my_sorted_set['http://dahlia.kr/']  # ZSCORE
2.0
>>> my_sorted_set.add('http://dahlia.kr/')
>>> my_sorted_set['http://dahlia.kr/']  # ZSCORE
3.0
```

As you can see, doubly added members get double scores. This property is what we will use in the middleware.

You can list values and these scores the sorted set contains. Similar to `dict` there's `items()` method.

```
>>> my_sorted_set.items()  # ZRANGE
[('https://bitbucket.org/dahlia/sider', 1.0),
 ('https://twitter.com/hongminhee', 2.0),
 ('http://dahlia.kr/', 2.0)]
>>> my_sorted_set.items(reverse=True)  # ZREVRANGE
[('http://dahlia.kr/', 2.0),
 ('https://twitter.com/hongminhee', 2.0),
 ('https://bitbucket.org/dahlia/sider', 1.0)]
```

There are other many features to `SortedSet` type, but it's all we need to know to implement the middleware. So we stop introduction of the type to step forward.

Replace `dict` with `SortedSet`

To replace `dict` with `SortedSet`, look `RefererStatMiddleware.__init__()` method first:

```
def __init__(self, application):
    self.application = application
    self.referer_set = {}
```

Note: The following codes implictly assumes that it imports:

```
from redis.client import StrictRedis
from sider.session import Session
from sider.types import SortedSet
```

The above code can be easily changed to:

```
def __init__(self, application):
    assert callable(application)
    self.application = application
    client = StrictRedis()
    session = Session(client)
    self.referer_set = session.get('wsgi_referer_set', SortedSet)
```

It should be more configurable by users. Redis key is currently hard-coded as `wsgi_referer_set`. It can be parameterized, right?

```
def __init__(self, set_key, application):
    assert callable(application)
    self.application = application
    client = StrictRedis()
    session = Session(client)
    self.referer_set = session.get(str(set_key), SortedSet)
```

It still lacks configurability. Users can't set address of Redis server to connect. Parameterize `session` as well:

```
def __init__(self, session, set_key, application):
    assert isinstance(session, Session)
    assert callable(application)
    self.application = application
    self.referer_set = session.get(str(set_key), SortedSet)
```

Okay, it's enough flexible to environments. Our first and third problems have just solved. Its data become shared and don't be split anymore. No data loss even if process has terminated.

Next, we have to make increment atomic. See a part of `RefererStatMiddleware.__call__()` method:

```
try:
    self.referer_set[referer] += 1
except KeyError:
    self.referer_set[referer] = 1
```

Redis sorted set offers a simple atomic way to increase its score: ZINCRBY. Sider maps ZINCRBY command to `SortedSet.add()` method. So, those lines can be replaced by the following line:

```
self.referer_set.add(referer)
```

and it will be committed atomically.

Referer list page

Lastly, let's add an additional page for listing collected referers. This page simply shows you list of referers and counts. Referers are ordered by these counts (descendingly).

To deal with HTML this example will use Jinja template engine. Its syntax is similar to Django template language, but more expressive. You can install it through **pip** or **easy_install**:

```
$ pip install Jinja2  # or:
$ easy_install Jinja2
```

Here is a HTML template code using Jinja:

```
<h1>Referer List</h1>
<table>
  <thead>
    <tr>
      <th>URL</th>
      <th>Count</th>
    </tr>
  </thead>
  <tbody>
    {% for url, count in referers %}
      <tr>
        <th><a href="{{ url|escape }}" rel="noreferrer">
              {{- url|escape }}</a></th>
        <td>{{ count|int }}</td>
      </tr>
    {% endfor %}
  </tbody>
</table>
```

Save this template source to the file named `templates/stat.html`. Remember we used an undefined variable in the above template code: `referers`. So we have to pass this variable from the WSGI middleware code.

To load this template file, Jinja environment object has to be set in the web application code. Append the following lines to `RefererStatMiddleware.__init__()` method:

```
loader = PackageLoader(__name__)
environment = Environment(loader=loader)
```

And then we now can load the template using `Environment.get_template()` method. Append the following line to `RefererStatMiddleware.__init__()` method:

```
self.template = environment.get_template('stat.html')
```

When `RefererStatMiddleware` is initialized its template will be loaded together.

Next, let's add a new `stat_application()` method, going to serve the list page, into the middleware class. This method has to be a WSGI application as well:

```
def stat_application(self, environ, start_response):
    content_type = 'text/html; charset=utf-8'
    start_response('200 OK', [('Content-Type', content_type)])
    referers = self.referer_set.items(reverse=True)
    return self.template.render(referers=referers).encode('utf-8'),
```

`Template.render()` method takes variables to pass as keywords and returns a rendered result as `unicode` string. We have passed the `referers` variable from this line. Its value is made by `SortedSet.items()` method with `reverse=True` option which means descending order.

To connect this modular WSGI application into the main application, we should add the following conditional routine into the first of `RefererStatMiddleware.__call__()` method:

```
path = environ['PATH_INFO']
if path == '/__stat__' or path.startswith('/__stat__/'):
    return self.stat_application(environ, start_response)
```

It will delegate its responsibility of responding to `stat_application()` application if a request is to the path `/__stat__` or its subpath.

Now go to `/__stat__` page and then your browser will show a table like this:

Source code

The complete source code of this example can be found in `examples/wsgi-referer-stat/` directory of the repository.

https://bitbucket.org/dahlia/sider/src/tip/examples/wsgi-referer-stat

It's public domain, feel free!

Final API

class `sider_wsgi_referer_stat`.**RefererStatMiddleware**(*session*, *set_key*, *application*, *stat_path='/__stat__'*)

A simple WSGI middleware that collects *`Referer`* headers and stores it into a Redis sorted set.

You can see the list of referrers ordered by duplication count in `/__stat__` page (or you can configure the `stat_path` argument).

Parameters

- **session** (`sider.session.Session`) – sider session object
- **set_key** (`basestring`) – the key name of Redis sorted set to store data
- **application** (`collections.Callable`) – wsgi app to wrap
- **stat_path** (`basestring`) – path to see the collected data. default is `'/__stat__'`. if it's `None` the data cannot be accessed from outside

referer_set = None

(`sider.sortedset.SortedSet`) The set of collected *`Referer`* strings.

stat_application(*environ*, *start_response*)

WSGI application that lists its collected referers.

2.2 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.

2.2.1 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
```

2.2.2 Convention

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

```
"""`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.
```

2.2.3 Tips

- You can link Redis commands through `redis` role. For example:

```
Linking :redis:`ZRANGEBYSCORE` command.
```

- You can link `issue`, `commit` and `branch`. For example:

```
- Linking :issue:`1`.
- Linking :commit:`a78ac7eb7332`.
- Linking :branch:`docs`.
```

  It becomes:

  - Linking issue #1.
  - Linking a78ac7eb7332.
  - Linking docs.

2.3 To do list

2.3.1 To be added

- `sider.sortedset`

2.3.2 To be fixed

2.4 Roadmap

Sider is planning to provide a lot of things able to be done with Redis. It will be a long-running project, and planned features have their priority.

2.4.1 Version 0.3

Entity mapping (`sider.entity`) The main feature Sider 0.3 ships will be an entity mapper inspired by SQLAlchemy's manual mapper. In this version, entity mapper doesn't support any declarative interface yet.

It has been being developed in the branch entity-mapping.

Key templates (`sider.key`) You can organize keys by grouped values instead of raw vanilla string keys.

The branch name for this will be key.

Channels (`sider.channel`) By using Redis' pub/sub channels you will be able to use Redis as your simple message queue.

The branch name for this will be channel.

Extension namespace (`sider.ext`) User-contributed modules can be plugged inside the namespace `sider.ext`. If you write an extension module for Sider and name it `sider_something` it will be imported by `sider.ext.something`.

It has been being developed in the branch ext.

2.4.2 Version 0.4

Declarative entity mapper (`sider.entity.declarative`) Inspired by SQLAlchemy's declarative mapper, by using metaclasses, Sider will provide the easier mapping interface to use built on top of the manual mapper.

It will be developed in the branch entity-mapping.

Indices (`sider.entity.index`) While Redis hashes don't have any indices Sider's entity mapper will provide indices for arbitrary expressions by generating materialized views and you can search entities by indexed fields.

It will be developed in the branch entity-index.

Simple distributed task queue (`sider.ext.task`) By using `sider.channel` Sider will offer the simple distributed task queue. It will have very subset features of Celery (while Celery supports various AMQP implementations other than Redis e.g. RabbitMQ).

It will be developed in the branch ext-task.

2.4.3 Any other features?

Isn't there the feature what you're looking for? So write the feature request in our issue tracker.

2.5 Sider Changelog

2.5.1 Version 0.2.0

To be released.

- Added `sider.transaction` module.
- Added `sider.sortedset` module.
- Added `sider.types.SortedSet` type.
- Added `sider.types.Time` and `sider.types.TZTime` types.
- Added `sider.types.TimeDelta` type.
- Introduced `sider.types.Tuple` type for ad-hoc composition of multiple types.
- The extensible namespace package `sider.ext` was introduced.
- Added `sider.threadlocal` module.
- Added `sider.session.Session.verbose_transaction_error` option.

2.5.2 Version 0.1.3

Released on April 21, 2012. Pre-alpha release.

- Now `sider.hash.Hash` objects show their contents for `repr()`.
- Now persist objects show their key name for `repr()`.
- Added `sider.lazyimport.exceptions` deferred module.

2.5.3 Version 0.1.2

Released on April 11, 2012. Pre-alpha release.

- Now `sider.session.Session` takes `redis.client.StrictRedis` object instead of `redis.client.Redis` which is deprecated.
- Added `sider.exceptions` module.
- Added `sider.warnings.SiderWarning` base class.
- Fixed a bug of `sider.list.List.insert()` for index -1. Previously it simply appends an element to the list (and that is an incorrect behavior), but now it inserts an element into the right before of its last element.

2.5.4 Version 0.1.1

Released on March 29, 2012. Pre-alpha release.

- 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`.

2.5.5 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