Tarantool python client library Documentation

Release 0.6.6

Konstantin Cherkasoff

Oct 07, 2022

Contents

Version 0.6.6

Download

- PyPI
- GitHub

Install

```
$ pip install tarantool
```

Tarantool is a damn fast key/value data store originally designed by Mail.Ru and released under the terms of BSD license.

CHAPTER 1

Documentation

1.1 Quick start

1.1.1 Connecting to the server

Create connection to the server:

```
>>> import tarantool
>>> server = tarantool.connect("localhost", 33013)
```

1.1.2 Creating a space instance

Instance of *Space* is a named object to access the key space.

Create ‘‘ demo ‘‘ object which will be used to access the space ‘‘ 0 ‘‘

```
>>> demo = server.space(0)
```

All subsequent operations with space `0` performed using methods of the `demo`.

1.1.3 Data Manipulation

Select

Select one single record with id `'AAAA'` from the space `demo` using primary key (index zero):

```
>>> demo.select('AAAA')
```

Select several records using primary index:

```
>>> demo.select(['AAAA', 'BBBB', 'CCCC'])
[('AAAA', 'Alpha'), ('BBBB', 'Bravo'), ('CCCC', 'Charlie')]
```

Insert

Insert tuple `('DDDD', 'Delta')` into the space `demo`:

```
>>> demo.insert(('DDDD', 'Delta'))
```

The first element is the primary key for the tuple.

Update

Update the record with id `'DDDD'` placing the value `'Denver'` into the field `1`:

```
>>> demo.update('DDDD', [(1, '=', 'Denver')])
[('DDDD', 'Denver')]
```

To find the record *update()* always uses the primary index. Fields numbers are starting from zero. So field `0` is the first element in the tuple.

Delete

Delete single record identified by id `'DDDD'`:

```
>>> demo.delete('DDDD')
[('DDDD', 'Denver')]
```

To find the record *delete()* always uses the primary index.

1.1.4 Call server-side functions

To call stored function method *Connection.call()* can be used:

```
>>> server.call("box.select_range", (0, 0, 2, 'AAAA'))
[('AAAA', 'Alpha'), ('BBBB', 'Bravo')]
```

The same can be done using *Space.call()* method:

```
>>> demo = server.space(0)
>>> demo.call("box.select_range", (0, 0, 2, 'AAAA'))
[('AAAA', 'Alpha'), ('BBBB', 'Bravo')]
```

Method *Space.call()* is just an alias for *Connection.call()*

1.2 Developer's guide

1.2.1 Basic concepts

Spaces

Spaces is a collections of tuples. Usually, tuples in one space represent objects of the same type, although this is not necessary.

Note: The analogue of spaces is tables in traditional (SQL) databases.

Spaces have integer identifiers defined in the server configuration. To access the space as a named object it is possible to use the method *Connection.space()* and an instance of *Space*.

Example:

```
>>> customer = connection.space(0)
>>> customer.insert(('FFFF', 'Foxtrot'))
```

Field types

Three field types are supported in Tarantool: `STR`, `NUM` and `NUM64`. These types are used only for index configuration but not saved in tuple's data and not transferred between the client and server. Thus, from the client point of view, fields are raw byte arrays without explicitly definde types.

It is much easier to use native types for python developer: `int`, `long`, `unicode` (`int` and `str` for Python 3.x). For raw binary data `bytes` should be used (in this case the type casting is not performed).

Tarantool data types corresponds to the following Python types:

- `RAW` - `bytes`
- `STR` - `unicode` (`str` for Python 3.x)
- `NUM` - `int`
- `NUM64` - `int` or `long` (`int` for Python 3.x)

Please define spaces schema to enable automatic type casting:

```
>>> import tarantool
>>> schema = {
        0: { # Space description
            'name': 'users', # Space name
            'default_type': tarantool.STR, # Type that used to decode fields that are␣
↪not listed below
            'fields': {
                0: ('numfield', tarantool.NUM), # (field name, field type)
                1: ('num64field', tarantool.NUM64),
                2: ('strfield', tarantool.STR),
                #2: { 'name': 'strfield', 'type': tarantool.STR }, # Alternative␣
↪syntax
                #2: tarantool.STR # Alternative syntax
            },
            'indexes': {
                0: ('pk', [0]), # (name, [field_no])
                #0: { 'name': 'pk', 'fields': [0]}, # Alternative syntax
                #0: [0], # Alternative syntax
            }
        }
    }
```

(continues on next page)

(continued from previous page)

```
>>> connection = tarantool.connect(host = 'localhost', port=33013, schema = schema)
>>> demo = connection.space('users')
>>> demo.insert((0, 12, u'this is unicode string'))
>>> demo.select(0)
[(0, 12, u'this is unicode string')]
```

As you can see, original "raw" fields were casted to native types as defined in the schema.

Tarantool's tuple can contain any number of fields. If some fields are not defined then `default_type` will be used.

To prevent implicit type casting for strings use `RAW` type. Raw byte fields should be used if the application uses binary data (eg, images or python objects packed with `picke`).

You can also specify schema for CALL results:

```
>>> ...
# Copy schema decription from 'users' space
>>> connection.call("box.select", '0', '0', 0L, space_name='users');
[(0, 12, u'this is unicode string')]
# Provide schema description explicitly
>>> field_defs = [('numfield', tarantool.NUM), ('num64field', tarantool.NUM)]
>>> connection.call("box.select", '0', '1', 184L, field_defs = field_defs, default_
↪type = tarantool.STR);
[(0, 12, u'this is unicode string')]
```

Note: Python 2.6 adds `bytes` as a synonym for the `str` type, and it also supports the `b''` notation.

Note: **utf-8** allways used for type conversion between `unicode` and `bytes`

Request response

Requests (*`insert()`*, *`delete()`*, *`update()`*, *`select()`*) return a *`Response`* instance.

Class *`Response`* inherited from *list*, so in fact response can be used as a list of a tuples.

In addition *`Response`* instance has the `rowcount` attribute. The value of `rowcount` equals to the number of records affected by the request. For example for *`delete()`* request `rowcount` is equals to `1` if record was deleted.

1.2.2 Connect to the server

To connect to the server it is required to use *`tarantool.connect()`* method. It returns an *`Connection`* instance.

Example:

```
>>> import tarantool
>>> connection = tarantool.connect("localhost", 33013)
>>> type(connection)
<class 'tarantool.connection.Connection'>
```

1.2.3 Data manipulation

There are four basic operations supported by Tarantool: **insert**, **delete**, **update** and **select**.

Note:

Inserting and replacing records

To insert or replace records *Space.insert()* method should be used:

```
>>> user.insert((user_id, email, int(time.time())))
```

The first element of the tuple is always its unique primary key.

If an entry with the same key already exists, it will be replaced without any warning or error message.

Note: In case of `insert` request `Response.rowcount` is always equals to `1`

Deleting Records

To delete records *Space.delete()* method should be used:

```
>>> user.delete(primary_key)
```

Note: If the record was deleted `Response.rowcount` equals to `1`. If the record was not found `Response.rowcount` equals to `0`.

Updating Records

Update request in Tarantool allows to simultaneous and atomic update multiple fields of a tuple.

To update records *Space.update()* method should be used.

Example:

```
>>> user.update(1001, [('=', 1, 'John'), ('=', 2, 'Smith')])
```

In this example new values for fields `1` and `2` are assigned.

Space.update() method allows to change multiple fields of the tuple at a time.

The following update operations are supported by Tarantool:

- `'='` – assign new value to the field
- `'+'` – add argument to the field (*both arguments are treated as signed 32-bit ints*)
- `'^'` – bitwise AND (*only for 32-bit integers*)
- `'|'` – bitwise XOR (*only for 32-bit integers*)
- `'&'` – bitwise OR (*only for 32-bit integers*)

- `'splice'` – implementation of Perl splice function

Note: The zero (i.e. [0]) field of the tuple can not be updated, because it is the primary key

See also:

See *`Space.update()`* documentation for details

Warning: `'splice'` operation is not implemented yet

Selecting Records

To select records *`Space.select()`* method should be used. *SELECT* query can return one or many records.

Select by primary key

Select a record using its primary key `3800`:

```
>>> world.select(3800)
[(3800, u'USA', u'Texas', u'Dallas', 1188580)]
```

Select using secondary index

```
>>> world.select('USA', index=1)
[(3796, u'USA', u'Texas', u'Houston', 1953631),
 (3801, u'USA', u'Texas', u'Huston', 10000),
 (3802, u'USA', u'California', u'Los Angeles', 10000),
 (3805, u'USA', u'California', u'San Francisco', 776733),
 (3800, u'USA', u'Texas', u'Dallas', 1188580),
 (3794, u'USA', u'California', u'Los Angeles', 3694820)]
```

Argument `index = 1` indicates that secondary index (`1`) should be used. The primary key (`index=0`) is used by default.

Note: Secondary indexes must be explicitly declared in the server configuration

Select records using several keys

Note: This conforms to `where key in (k1, k2, k3...)`

Select records with primary key values `3800`, `3805` and `3796`:

```
>>> world.select([3800, 3805, 3796])
[(3800, u'USA', u'Texas', u'Dallas', 1188580),
 (3805, u'USA', u'California', u'San Francisco', 776733),
 (3796, u'USA', u'Texas', u'Houston', 1953631)]
```

Retrieve a record by using a composite index

Select data on cities in Texas:

```
>>> world.select([('USA', 'Texas')], index=1)
[(3800, u'USA', u'Texas', u'Dallas', 1188580), (3796, u'USA', u'Texas', u'Houston',␣
↪1953631)]
```

Select records explicitly specifying field types

Tarantool has no strict schema so all fields are raw binary byte arrays. You can specify field types in the `schema` parameter to a connection.

1.2.4 Call server-side functions

A server-side function written in Lua can select and modify data, access configuration and perform administrative tasks.

To call stored function *`Connection.call()`* method should be used. Also, this method has an alias *`Space.call()`*.

Example:

```
>>> server.call("box.select_range", (1, 3, 2, 'AAAA'))
[(3800, u'USA', u'Texas', u'Dallas', 1188580), (3794, u'USA', u'California', u'Los␣
↪Angeles', 3694820)]
```

See also:

Tarantool/Box User Guide » Writing stored procedures in Lua

See also:

Tarantool/Box User Guide

CHAPTER 2

API Reference

2.1 module tarantool

tarantool.**connect**(*host='localhost'*, *port=33013*, *user=None*, *password=None*, *encoding=None*)
Create a connection to the Tarantool server.

Parameters

- **host** (*str*) – Server hostname or IP-address
- **port** (*int*) – Server port

Return type *Connection*

Raise *NetworkError*

class tarantool.**Connection**(*host*, *port*, *user=None*, *password=None*, *socket_timeout=None*, *reconnect_max_attempts=10*, *reconnect_delay=0.1*, *connect_now=True*, *encoding=None*, *call_16=False*, *connection_timeout=None*)
Represents connection to the Tarantool server.

This class is responsible for connection and network exchange with the server. Also this class provides low-level interface to data manipulation (insert/delete/update/select).

Initialize a connection to the server.

Parameters

- **host** (*str*) – Server hostname or IP-address
- **port** (*int*) – Server port
- **connect_now** (*bool*) – if True (default) than __init__() actually

creates network connection. if False than you have to call connect() manualy.

exception DatabaseError
Error related to the database engine

Error = <module 'tarantool.error' from '/home/docs/checkouts/readthedocs.org/user_buil

exception InterfaceError
Error related to the database interface rather than the database itself

exception NetworkError(*orig_exception=None*, **args*)
Error related to network

exception SchemaError(*value*)

authenticate(*user*, *password*)
Execute AUTHENTICATE request.

Parameters

- **user** (*string*) – user to authenticate with
- **password** (*string*) – password for the user

Return type *Response* instance

call(*func_name*, **args*)
Execute CALL request. Call stored Lua function.

Parameters

- **func_name** (*str*) – stored Lua function name
- **args** (*list or tuple*) – list of function arguments

Return type *Response* instance

close()
Close connection to the server

connect()
Create connection to the host and port specified in __init__(). Usually there is no need to call this method directly, since it is called when you create an *Connection* instance.

Raise *NetworkError*

connect_basic()

connect_tcp()
Create connection to the host and port specified in __init__(). :raise: *NetworkError*

connect_unix()
Create connection to the host and port specified in __init__(). :raise: *NetworkError*

delete(*space_name*, *key*, ***kwargs*)
Execute DELETE request. Delete single record identified by *key*. If you're using secondary index, it must be unique.

Parameters

- **space_name** (*int or name*) – space number or name to delete a record
- **key** (*int or str*) – key that identifies a record

Return type *Response* instance

eval(*expr*, **args*)
Execute EVAL request. Eval Lua expression.

Parameters

- **expr** (*str*) – Lua expression

- **args** (*list or tuple*) – list of function arguments

Return type *Response* instance

flush_schema()

generate_sync()
Need override for async io connection

handshake()

insert(*space_name*, *values*)
Execute INSERT request. It will throw error if there's tuple with same PK exists.

Parameters

- **space_name** (*int or str*) – space id to insert a record
- **values** (*tuple*) – record to be inserted. The tuple must contain only scalar (integer or strings) values

Return type *Response* instance

join(*server_uuid*)

load_schema()

ping(*notime=False*)
Execute PING request. Send empty request and receive empty response from server.

Returns response time in seconds

Return type float

replace(*space_name*, *values*)
Execute REPLACE request. It won't throw error if there's no tuple with this PK exists

Parameters

- **space_name** (*int or str*) – space id to insert a record
- **values** (*tuple*) – record to be inserted. The tuple must contain only scalar (integer or strings) values

Return type *Response* instance

select(*space_name*, *key=None*, ***kwargs*)
Execute SELECT request. Select and retrieve data from the database.

Parameters

- **space_name** (*int or str*) – specifies which space to query
- **values** (*list, tuple, set, frozenset of tuples*) – values to search over the index
- **index** (*int or str*) – specifies which index to use (default is **0** which means that the **primary index** will be used)
- **offset** (*int*) – offset in the resulting tuple set
- **limit** (*int*) – limits the total number of returned tuples

Return type *Response* instance

You may use names for index/space. Matching id's -> names connector will get from server.

Select one single record (from space=0 and using index=0) >>> select(0, 1)

Select single record from space=0 (with name='space') using composite index=1 (with name '_name'). >>> select(0, [1,'2'], index=1) # OR >>> select(0, [1,'2'], index='_name') # OR >>> select('space', [1,'2'], index='_name') # OR >>> select('space', [1,'2'], index=1)

Select all records >>> select(0) # OR >>> select(0, [])

space (*space_name*)
Create *Space* instance for particular space

Space instance encapsulates the identifier of the space and provides more convenient syntax for accessing the database space.

Parameters **space_name** (*int or str*) – identifier of the space

Return type *Space* instance

subscribe (*cluster_uuid*, *server_uuid*, *vclock=None*)

update (*space_name*, *key*, *op_list*, ***kwargs*)
Execute UPDATE request.

The *update* function supports operations on fields — assignment, arithmetic (if the field is unsigned numeric), cutting and pasting fragments of a field, deleting or inserting a field. Multiple operations can be combined in a single update request, and in this case they are performed atomically and sequentially. Each operation requires specification of a field number. When multiple operations are present, the field number for each operation is assumed to be relative to the most recent state of the tuple, that is, as if all previous operations in a multi-operation update have already been applied. In other words, it is always safe to merge multiple update invocations into a single invocation, with no change in semantics.

Update single record identified by *key*.

List of operations allows to update individual fields.

Allowed operations:

(For every operation you must provide field number, to apply this operation to)

- + for addition (values must be numeric)
- - for subtraction (values must be numeric)
- *&* for bitwise AND (values must be unsigned numeric)
- | for bitwise OR (values must be unsigned numeric)
- ^ for bitwise XOR (values must be unsigned numeric)
- *:* for string splice (you must provide *offset*, *count* and *value* for this operation)
- *!* for insertion (before) (provide any element to insert)
- = for assignment (provide any element to assign)
- # for deletion (provide count of fields to delete)

Parameters

- **space_name** (*int or str*) – space number or name to update a record
- **index** (*int or str*) – index number or name to update a record
- **key** (*int or str*) – key that identifies a record
- **op_list** (*a list of the form [(symbol_1, field_1, arg_1), (symbol_2, field_2, arg_2_1, arg_2_2, arg_2_3), ..]*) – list of operations. Each operation is tuple of three (or more) values

Return type `Response` instance

Operation examples:

```
# 'ADD' 55 to second field
# Assign 'x' to third field
[('+', 2, 55), ('=', 3, 'x')]
# 'OR' third field with '1'
# Cut three symbols starting from second and replace them with '!!'
# Insert 'hello, world' field before fifth element of tuple
[('|', 3, 1), (':', 2, 2, 3, '!!'), ('!', 5, 'hello, world')]
# Delete two fields starting with second field
[('#', 2, 2)]
```

update_schema(*schema_version*)

upsert(*space_name*, *tuple_value*, *op_list*, ***kwargs*)

Execute UPSERT request.

If there is an existing tuple which matches the key fields of *tuple_value*, then the request has the same effect as UPDATE and the [(field_1, symbol_1, arg_1), . . .] parameter is used.

If there is no existing tuple which matches the key fields of *tuple_value*, then the request has the same effect as INSERT and the *tuple_value* parameter is used. However, unlike insert or update, upsert will not read a tuple and perform error checks before returning – this is a design feature which enhances throughput but requires more caution on the part of the user.

If you're using secondary index, it must be unique.

List of operations allows to update individual fields.

Allowed operations:

(For every operation you must provide field number, to apply this operation to)

- + for addition (values must be numeric)
- - for subtraction (values must be numeric)
- *&* for bitwise AND (values must be unsigned numeric)
- | for bitwise OR (values must be unsigned numeric)
- ^ for bitwise XOR (values must be unsigned numeric)
- *:* for string splice (you must provide *offset*, *count* and *value* for this operation)
- *!* for insertion (provide any element to insert)
- = for assignment (provide any element to assign)
- *#* for deletion (provide count of fields to delete)

Parameters

- **space_name** (*int or str*) – space number or name to update a record
- **index** (*int or str*) – index number or name to update a record
- **tuple_value** – tuple, that
- **op_list** (*a list of the form [(symbol_1, field_1, arg_1), (symbol_2, field_2, arg_2_1, arg_2_2, arg_2_3),..]*) – list of operations. Each operation is tuple of three (or more) values

Return type *Response* instance

Operation examples:

```
# 'ADD' 55 to second field
# Assign 'x' to third field
[('+', 2, 55), ('=', 3, 'x')]
# 'OR' third field with '1'
# Cut three symbols starting from second and replace them with '!!'
# Insert 'hello, world' field before fifth element of tuple
[('|', 3, 1), (':', 2, 2, 3, '!!'), ('!', 5, 'hello, world')]
# Delete two fields starting with second field
[('#', 2, 2)]
```

tarantool.**connectmesh**(*addrs=({'host': 'localhost', 'port': 3301},)*, *user=None*, *password=None*, *encoding=None*)

Create a connection to the mesh of Tarantool servers.

Parameters **addrs** (*list*) – A list of maps: {'host':(HOSTNAME|IP_ADDR), 'port':PORT}.

Return type *MeshConnection*

Raise *NetworkError*

class tarantool.**MeshConnection**(*host=None*, *port=None*, *user=None*, *password=None*, *socket_timeout=None*, *reconnect_max_attempts=10*, *reconnect_delay=0.1*, *connect_now=True*, *encoding=None*, *call_16=False*, *connection_timeout=None*, *addrs=None*, *strategy_class=<class 'tarantool.mesh_connection.RoundRobinStrategy'>*, *cluster_discovery_function=None*, *cluster_discovery_delay=60*)

Represents a connection to a cluster of Tarantool servers.

This class uses Connection to connect to one of the nodes of the cluster. The initial list of nodes is passed to the constructor in 'addrs' parameter. The class set in 'strategy_class' parameter is used to select a node from the list and switch nodes in case of unavailability of the current node.

'cluster_discovery_function' param of the constructor sets the name of a stored Lua function used to refresh the list of available nodes. The function takes no parameters and returns a list of strings in format 'host:port'. A generic function for getting the list of nodes looks like this:

```
function get_cluster_nodes()
    return {
        '192.168.0.1:3301',
        '192.168.0.2:3302',
        -- ...
    }
end
```

You may put in this list whatever you need depending on your cluster topology. Chances are you'll want to make the list of nodes from nodes' replication config. Here is an example for it:

```
local uri_lib = require('uri')

function get_cluster_nodes()
    local nodes = {}

    local replicas = box.cfg.replication

```

(continues on next page)

(continued from previous page)

```
    for i = 1, #replicas do
        local uri = uri_lib.parse(replicas[i])

        if uri.host and uri.service then
            table.insert(nodes, uri.host .. ':' .. uri.service)
        end
    end

    -- if your replication config doesn't contain the current node
    -- you have to add it manually like this:
    table.insert(nodes, '192.168.0.1:3301')

    return nodes
end
```

connect()
Create connection to the host and port specified in __init__(). Usually there is no need to call this method directly, since it is called when you create an *Connection* instance.

Raise *NetworkError*

class tarantool.**Schema**(*con*)

fetch_index(*space_object*, *index*)

fetch_index_all()

fetch_index_from(*space*, *index*)

fetch_space(*space*)

fetch_space_all()

fetch_space_from(*space*)

flush()

get_field(*space*, *field*)

get_index(*space*, *index*)

get_space(*space*)

exception tarantool.**Error**
Base class for error exceptions

exception tarantool.**DatabaseError**
Error related to the database engine

exception tarantool.**NetworkError**(*orig_exception=None*, **args*)
Error related to network

exception tarantool.**NetworkWarning**
Warning related to network

exception tarantool.**SchemaError**(*value*)

2.2 class Connection

class tarantool.connection.**Connection**(*host*, *port*, *user=None*, *password=None*, *socket_timeout=None*, *reconnect_max_attempts=10*, *reconnect_delay=0.1*, *connect_now=True*, *encoding=None*, *call_16=False*, *connection_timeout=None*)

Represents connection to the Tarantool server.

This class is responsible for connection and network exchange with the server. Also this class provides low-level interface to data manipulation (insert/delete/update/select).

Initialize a connection to the server.

Parameters

- **host** (*str*) – Server hostname or IP-address
- **port** (*int*) – Server port
- **connect_now** (*bool*) – if True (default) than __init__() actually

creates network connection. if False than you have to call connect() manualy.

call(*func_name*, **args*)

Execute CALL request. Call stored Lua function.

Parameters

- **func_name** (*str*) – stored Lua function name
- **args** (*list or tuple*) – list of function arguments

Return type *Response* instance

close()

Close connection to the server

ping(*notime=False*)

Execute PING request. Send empty request and receive empty response from server.

Returns response time in seconds

Return type float

space(*space_name*)

Create *Space* instance for particular space

Space instance encapsulates the identifier of the space and provides more convenient syntax for accessing the database space.

Parameters **space_name** (*int or str*) – identifier of the space

Return type *Space* instance

2.3 class MeshConnection

class tarantool.mesh_connection.**MeshConnection**(*host=None*, *port=None*, *user=None*, *password=None*, *socket_timeout=None*, *reconnect_max_attempts=10*, *reconnect_delay=0.1*, *connect_now=True*, *encoding=None*, *call_16=False*, *connection_timeout=None*, *addrs=None*, *strategy_class=<class 'tarantool.mesh_connection.RoundRobinStrategy'>*, *cluster_discovery_function=None*, *cluster_discovery_delay=60*)

Represents a connection to a cluster of Tarantool servers.

This class uses Connection to connect to one of the nodes of the cluster. The initial list of nodes is passed to the constructor in 'addrs' parameter. The class set in 'strategy_class' parameter is used to select a node from the list and switch nodes in case of unavailability of the current node.

'cluster_discovery_function' param of the constructor sets the name of a stored Lua function used to refresh the list of available nodes. The function takes no parameters and returns a list of strings in format 'host:port'. A generic function for getting the list of nodes looks like this:

```
function get_cluster_nodes()
    return {
        '192.168.0.1:3301',
        '192.168.0.2:3302',
        -- ...
    }
end
```

You may put in this list whatever you need depending on your cluster topology. Chances are you'll want to make the list of nodes from nodes' replication config. Here is an example for it:

```
local uri_lib = require('uri')

function get_cluster_nodes()
    local nodes = {}

    local replicas = box.cfg.replication

    for i = 1, #replicas do
        local uri = uri_lib.parse(replicas[i])

        if uri.host and uri.service then
            table.insert(nodes, uri.host .. ':' .. uri.service)
        end
    end

    -- if your replication config doesn't contain the current node
    -- you have to add it manually like this:
    table.insert(nodes, '192.168.0.1:3301')

    return nodes
end
```

2.4 class Space

class tarantool.space.**Space**(*connection*, *space_name*)

Object-oriented wrapper for accessing a particular space. Encapsulates the identifier of the space and provides more convenient syntax for database operations.

Create Space instance.

Parameters

- **connection** (*Connection* instance) – Object representing connection to the server
- **space_name** (*int or str*) – space no or name to insert a record

call(*func_name*, **args*, ***kwargs*)

Execute CALL request. Call stored Lua function.

It's deprecated, use *~tarantool.connection.call* instead

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

Execute DELETE request.

See *~tarantool.connection.delete* for more information

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

Execute INSERT request.

See *~tarantool.connection.insert* for more information

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

Execute REPLACE request.

See *~tarantool.connection.replace* for more information

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

Execute SELECT request.

See *~tarantool.connection.select* for more information

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

Execute UPDATE request.

See *~tarantool.connection.update* for more information

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

Execute UPDATE request.

See *~tarantool.connection.upsert* for more information

2.5 class Response

class tarantool.response.**Response**(*conn*, *response*)

Represents a single response from the server in compliance with the Tarantool protocol. Responsible for data encapsulation (i.e. received list of tuples) and parses binary packet received from the server.

Create an instance of *Response* using data received from the server.

__init__() itself reads data from the socket, parses response body and sets appropriate instance attributes.

Parameters **body** (*array of bytes*) – body of the response

body

Type dict

Required field in the server response. Contains raw response body.

code

Type int

Required field in the server response. Contains response type id.

count(*value*) → integer – return number of occurrences of value

data

Type object

Required field in the server response. Contains list of tuples of SELECT, REPLACE and DELETE requests and arbitrary data for CALL.

index(*value*) → integer – return first index of value.
Raises ValueError if the value is not present.

return_code

Type int

Required field in the server response. Value of *return_code* can be 0 if request was sucessfull or contains an error code. If *return_code* is non-zero than *return_message* contains an error message.

return_message

Type str

The error message returned by the server in case of *return_code* is non-zero.

rowcount

Type int

Number of rows affected or returned by a query.

schema_version

Type int

Current schema version of request.

strerror

Type str

It may be ER_OK if request was successful, or contain error code string.

sync

Type int

Required field in the server response. Contains response header IPROTO_SYNC.

Python Module Index

t

Index

N

P

R

S

T

U