



# Python Driver 2.5 for Apache Cassandra Documentation

`${ds.localized.time}`

# Contents

- About the Python driver..... 3**
  
- Architectural overview.....4**
  - The driver and its dependencies..... 4
  - Optional non-Python dependencies..... 5
  
- Installation..... 7**
  
- Writing your first client.....9**
  - Connecting to a Cassandra cluster.....9
  - Using a session to execute CQL statements..... 11
  - Using prepared statements..... 15
  
- Python driver reference..... 18**
  - Asynchronous I/O..... 18
  - Automatic failover..... 19
  - CQL data types to Python types.....20
  - Debugging.....22
    - Enabling debug-level logging..... 22
  - Mapping user-defined types..... 23
  - Metadata refresh controls.....25
  - Named parameters..... 25
  - Object-mapping package.....26
  - Passing parameters to CQL queries.....26
  - Tuple types..... 28
  
- FAQ.....29**
  - Is there a distinction between different hosts?..... 29
  - Are there plans to allow for a retry policy option for the execute() or execute\_async() methods?.... 29
  
- API reference.....30**
  
- Using the docs.....31**

## About the Python driver

Use this driver in production applications to pass CQL statements from the client to a cluster and retrieve, manipulate, or remove data.

The Python driver is a modern, feature-rich and highly tunable Java client library for Apache Cassandra (1.2+) and DataStax Enterprise (3.1+) using exclusively Cassandra's binary protocol and Cassandra Query Language v3.

Use this driver in production applications to pass CQL statements from the client to a cluster and retrieve, manipulate, or remove data. Cassandra Query Language (CQL) is the primary language for communicating with the Cassandra database. Documentation for CQL is available in [CQL for Cassandra 2.x](#). DataStax also provides [DataStax DevCenter](#), which is a free graphical tool for creating and running CQL statements against Apache Cassandra and DataStax Enterprise. Other administrative tasks can be accomplished using [OpsCenter](#).

Here are the new and noteworthy features of the Python driver.

### What's new in 2.5

- Integrated [cqlengine object-mapping](#) package
- Support new [date and time data types](#)
- [Utility functions](#) for converting CQL `timeuuid` data type to `datetime` type
- [Randomized metadata fetch windows with configuration](#)

### What's new in 2.1

- **Cassandra 2.1 support**
  - [User-defined types](#) (UDT)
  - When using the version 3 protocol, only one connection is opened per-host, and throughput is improved due to reduced pooling overhead and lock contention.
  - [Tuple type](#) (limited usage Cassandra 2.0.9, full usage in Cassandra 2.1)
  - Protocol-level client-side timestamps (see [Session.use\\_client\\_timestamp](#))
- **New features**
  - Overridable type encoding for non-prepared statements (see [Session.encoder](#))
  - Configurable serial consistency levels for batch statements
  - Use [io.BytesIO](#) for reduced CPU consumption
  - Support Twisted as a reactor. Note that a Twisted-compatible API is not exposed (so no `Deferreds`), this is just a reactor implementation.

### What's new in 2.0

- Support v2 of Cassandra's native protocol, which includes the following new features: automatic query paging support, protocol-level batch statements, and lightweight transactions
- Support for Python 3.3 and 3.4
- Allow a default query timeout to be set per-Session
- Support v2 protocol authentication

# Architectural overview

Information about the driver's architecture.

The Python Driver 2.5 for Apache Cassandra works exclusively with the Cassandra Query Language version 3 (CQL3) and Cassandra's binary protocol.

The driver architecture is a layered one. At the bottom lies the driver core. This core handles everything related to the connections to a Cassandra cluster (for example, connection pool, discovering new nodes, etc.) and exposes a simple, relatively low-level API on top of which a higher level layer can be built.

The driver has the following features:

- **Asynchronous:** the driver uses the new CQL binary protocol asynchronous capabilities. Only a relatively low number of connections per nodes needs to be maintained open to achieve good performance.
- **Nodes discovery:** the driver automatically discovers and uses all nodes of the Cassandra cluster, including newly bootstrapped ones.
- **Configurable load balancing:** the driver allows for custom routing and load balancing of queries to Cassandra nodes. Out of the box, round robin is provided with optional data-center awareness (only nodes from the local data-center are queried (and have connections maintained to)) and optional token awareness (that is, the ability to prefer a replica for the query as coordinator).
- **Transparent failover:** if Cassandra nodes fail or become unreachable, the driver automatically and transparently tries other nodes and schedules reconnection to the dead nodes in the background.
- **Cassandra trace handling:** tracing can be set on a per-query basis and the driver provides a convenient API to retrieve the trace.
- **Convenient schema access:** the driver exposes a Cassandra schema in a usable way.
- **Configurable retry policy:** a retry policy can be set to define a precise behavior to adopt on query execution exceptions (for example, timeouts, unavailability). This avoids polluting client code with retry-related code.
- **Tunability:** the default behavior of the driver can be changed or fine tuned by using tuning policies and connection options.

Queries can be executed synchronously or asynchronously and prepared statements are supported.

## The driver and its dependencies

The Python driver only supports the Cassandra Binary Protocol and CQL3

### Cassandra binary protocol

The driver uses the binary protocol that was introduced in Cassandra 1.2. It only works with a version of Cassandra greater than or equal to 1.2. Furthermore, the binary protocol server is not started with the default configuration file in Cassandra 1.2. You must edit the `cassandra.yaml` file for each node:

```
start_native_transport: true
```

Then restart the node.

### Cassandra compatibility

The 2.0 version of the driver handles a single version of the Cassandra native protocol for the sake of simplicity. Cassandra does the multiple version handling. This makes it possible to do a rolling upgrade of a Cassandra cluster from 1.2 to 2.0 and then to upgrade the drivers in the application layer from 1.0 to 2.0. Because the application code needs to be changed anyway to leverage the new features of Cassandra 2.0, this small constraint appear to be fair.

	Python# driver 1.0.x	Python driver 2.0.x
Cassandra 1.2.x	Compatible	Compatible
Cassandra 2.0.x	Compatible for Cassandra 1.0 API and commands	Compatible

## Optional non-Python dependencies

The driver has several optional features that have non-Python dependencies.

### C extensions

By default, two C extensions are compiled: one that adds support for token-aware routing with the Murmur3Partitioner, and one that allows you to use libev for the event loop, which improves performance.

When installing manually through setup.py, you can disable both with the --no-extensions option, or selectively disable one or the other with --no-murmur3 and --no-libev.

To compile the extensions, ensure that GCC and the Python headers are available.

On Ubuntu and Debian, this can be accomplished by running:

```
$ sudo apt-get install gcc python-dev
```

On RedHat and RedHat-based systems like CentOS and Fedora:

```
$ sudo yum install gcc python-devel
```

On Mac OS X, homebrew installations of Python should provide the necessary headers.

### libev support

The driver currently uses Python's `asyncore` module for its default event loop. For better performance, libev is also supported through a C extension.

If you're on Linux, you should be able to install libev through a package manager. For example, on Debian/Ubuntu:

```
$ sudo apt-get install libev4 libev-dev
```

On On RedHat and RedHat-based systems like CentOS and Fedora:

```
$ sudo yum install libev libev-devel
```

If you're on Mac OS X, you should be able to install libev through [Homebrew](#). For example, on Mac OS X:

```
$ brew install libev
```

If successful, you should be able to build and install the extension (just using `setup.py build` or `setup.py install`) and then use the libev event loop by doing the following:

```
>>> from cassandra.io.libevreactor import LibevConnection
>>> from cassandra.cluster import Cluster
>>> cluster = Cluster()
>>> cluster.connection_class = LibevConnection
>>> session = cluster.connect()
```

### Compression support

Compression can optionally be used for communication between the driver and Cassandra. There are currently two supported compression algorithms: Snappy (in Cassandra 1.2+) and LZ4 (only in Cassandra 2.0+). If either is available for the driver and Cassandra also supports it, it will be used automatically.

For LZ4 support:

## Architectural overview

```
$ pip install lz4
```

For Snappy support:

```
$ pip install python-snappy
```

If using a Debian Linux derivative such as Ubuntu, it may be easier to just run

```
$ apt-get install python-snappy
```

.

# Installation

Different ways to install the Python driver.

## Supported platforms

Python 2.6, 2.7, 3.3, and 3.4 are supported. Both CPython (the standard Python implementation) and PyPy are supported and tested against.

Linux, Mac OS X, and Windows are supported.

## Installation with pip

`pip` is the suggested tool for installing packages. It handles installing all Python dependencies for the driver at the same time as the driver itself. To install the driver:

```
$ pip install cassandra-driver
```

You can use `pip install --pre cassandra-driver` if you need to install a beta version.

## Manual installation

When installing manually, ensure that the dependencies listed in the `requirements.txt` file are already installed.

Once the dependencies are installed, simply run:

```
$ python setup.py install
```

## Compression support (optional)

Compression can optionally be used for communication between the driver and Cassandra. There are currently two supported compression algorithms: snappy (in Cassandra 1.2+) and LZ4 (only in Cassandra 2.0+). If either is available for the driver and Cassandra also supports it, it will be used automatically.

### LZ4

```
$ pip install lz4
```

### Snappy

```
$ pip install python-snappy
```

If using a Debian Linux derivative such as Ubuntu, it may be easier to just run .

```
$ apt-get install python-snappy
```

## Metrics support (optional)

The driver has built-in support for capturing `Cluster.metrics` about the queries you run, however, the `scales` library is required to support this:

```
$ pip install scales
```

## Sorted sets support (optional)

Cassandra can store entire collections within a column. One of those collection types is a set. Cassandra's sets are actually ordered sets, but by default, the driver uses unordered sets to represent these collections. If you would like to maintain the ordering, install the `blist` library:

```
$ pip install blist
```

## Installation

### Non-Python dependencies (optional)

The driver has several optional features that have non-Python dependencies.

#### C Extensions

By default, two C extensions are compiled: one that adds support for token-aware routing with the `Murmur3Partitioner` and the other that allows you to use `libev` for the event loop, which improves performance.

When installing manually through `setup.py`, you can disable both with the `--no-extensions` option, or selectively disable one or the other with `--no-murmur3` and `--no-libev`.

To compile the extensions, ensure that GCC and the Python headers are available.

On Ubuntu and Debian, this can be accomplished by running:

```
$ sudo apt-get install gcc python-dev
```

On RedHat and RedHat-based systems like CentOS and Fedora: `$ sudo yum install gcc python-devel`

On OS X, homebrew installations of Python should provide the necessary headers.

#### libev support

The driver currently uses Python's `asyncore` module for its default event loop. For better performance, `libev` is also supported through a C extension.

On Linux, you should be able to install `libev` through a package manager. For example, on Debian/Ubuntu:

```
$ sudo apt-get install libev4 libev-dev
```

On RHEL/CentOS/Fedora:

```
$ sudo yum install libev libev-devel
```

On Mac OS X, you should be able to install `libev` through Homebrew. For example, on Mac OS X:

```
$ brew install libev
```

If successful, you should be able to build and install the extension (just using `setup.py build` or `setup.py install`) and then use the `libev` event loop by doing the following:

```
from cassandra.io.libevreactor import LibevConnection
from cassandra.cluster import Cluster
cluster = Cluster()
cluster.connection_class = LibevConnection
session = cluster.connect()
```

### Configuring SSL(optional)

Andrew Mussey has published a thorough guide on [Using SSL with the DataStax Python driver](#).



## Writing your first client

This section walks you through a small sample client application that uses the Python driver to connect to a Cassandra cluster, print out some metadata about the cluster, execute some queries, and print out the results.

### Connecting to a Cassandra cluster

The Python driver provides a `Cluster` class which is your client application's entry point for connecting to a Cassandra cluster and retrieving metadata.

#### Before you begin

This tutorial assumes you have the following software installed, configured, and that you have familiarized yourself with them:

- Apache Cassandra 1.2 or greater
- Python 2.7.x

#### About this task

Before you can start executing any queries against a Cassandra cluster, you must set up an instance of `Cluster`. As the name suggests, you typically have one instance of `Cluster` for each Cassandra cluster your application interacts with.

In this tutorial, you use a `Cluster` object, to connect to a node in your cluster and then retrieves metadata about the cluster and prints it out.

#### Procedure

1. In a text editor create a clients module.

- a) Name the file `clients.py`.
- b) Create a class `SimpleClient`.

```
class SimpleClient:
```

c) Add a `session` attribute.

```
    session = None
```

d) Add a method, `connect()`.

The `connect` method:

- builds a cluster object with the node IP address
- retrieves metadata from the cluster
- connects to the cluster
- prints out:
  - the name of the cluster
  - the datacenter, host name or IP address, and rack for each of the nodes in the cluster

```
def connect(self, nodes):
    cluster = Cluster(nodes)
    metadata = cluster.metadata
    self.session = cluster.connect()
    log.info('Connected to cluster: ' + metadata.cluster_name)
```

## Writing your first client

```
for host in metadata.all_hosts():
    log.info('Datacenter: %s; Host: %s; Rack: %s',
            host.datacenter, host.address, host.rack)
```

e) Add a method, `close`, to shut down the cluster object once you are finished with it.

```
def close(self):
    self.session.cluster.shutdown()
    log.info('Connection closed.')
```

f) In the module main function instantiate a `SimpleClient` object, call `connect()` on it, and then close it.

```
def main():
    client = SimpleClient()
    client.connect(['127.0.0.1'])
    client.close()
```

g) Save the file and open a command prompt.

2. Run the clients module.

```
$ python clients.py
```

### Code listing

The complete code listing illustrates:

- connecting to a cluster
- retrieving metadata and printing it out
- closing the connection to the cluster

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from cassandra.cluster import Cluster
import logging

log = logging.getLogger()
log.setLevel('INFO')

class SimpleClient(object):
    session = None

    def connect(self, nodes):
        cluster = Cluster(nodes)
        metadata = cluster.metadata
        self.session = cluster.connect()
        log.info('Connected to cluster: ' + metadata.cluster_name)
        for host in metadata.all_hosts():
            log.info('Datacenter: %s; Host: %s; Rack: %s',
                    host.datacenter, host.address, host.rack)

    def close(self):
        self.session.cluster.shutdown()
        log.info('Connection closed.')
```

```
def main():
    logging.basicConfig()
    client = SimpleClient()
    client.connect(['127.0.0.1'])
```

```

    client.close()

if __name__ == "__main__":
    main()

```

When run the client program prints out this metadata on the cluster's constituent nodes in the console pane:

```

INFO:cassandra.cluster:New Cassandra host 127.0.0.3 added
INFO:cassandra.cluster:New Cassandra host 127.0.0.2 added
INFO:root:Connected to cluster: firkus
INFO:root:Datacenter: datacenter1; Host: 127.0.0.3; Rack: rack1
INFO:root:Datacenter: datacenter1; Host: 127.0.0.2; Rack: rack1
INFO:root:Datacenter: datacenter1; Host: 127.0.0.1; Rack: rack1

```

## Using a session to execute CQL statements

Once you have connected to a Cassandra cluster using a cluster object, you retrieve a session, which allows you to execute CQL statements to read and write data.

### Before you begin

This tutorial uses a CQL3 schema which is described in a post on the DataStax developer blog. Reading [that post](#), could help with some of the new CQL3 concepts used here.

### About this task

Getting metadata for the cluster is good, but you also want to be able to read and write data to the cluster.

This tutorial uses a CQL3 schema which is described in [a post](#) on the DataStax developer blog. For reference, the schema is reproduced below, but reading that post, could help with some of the new CQL3 concepts used here. crea

```

CREATE KEYSPACE simplex WITH replication
= {'class':'SimpleStrategy', 'replication_factor':3};

CREATE TABLE simplex.songs (
    id uuid PRIMARY KEY,
    title text,
    album text,
    artist text,
    tags set<text>,
    data blob
);

CREATE TABLE simplex.playlists (
    id uuid,
    title text,
    album text,
    artist text,
    song_id uuid,
    PRIMARY KEY (id, title, album, artist)
);

```

The Python driver lets you execute CQL statements using a session object that you retrieve from the Cluster object. You will add code to your client for:

- creating tables

## Writing your first client

- inserting data into those tables
- querying the tables
- printing the results

You can execute queries by calling the `execute()` function on your session object. The session maintains multiple connections to the cluster nodes, provides policies to choose which node to use for each query (round-robin on all nodes of the cluster by default), and handles retries for failed queries when it makes sense.

Session objects are thread-safe and usually a single session object is all you need per application. However, a given session can only be set to one keyspace at a time, so one instance per keyspace is necessary. Your application typically only needs a single cluster object, unless you're dealing with multiple physical clusters.

### Procedure

1. Add a method, `create_schema()`, to the `SimpleClient` class implementation.

```
def create_schema(self):
```

2. Add code to implement the new method which creates a new schema.

- a) Execute a statement that creates a new keyspace.

Add to the `create_schema()` method:

```
self.session.execute("""
    CREATE KEYSPACE simplex WITH replication
        = {'class':'SimpleStrategy', 'replication_factor':3};
    """)
```

In this example, you create a new keyspace, `simplex`.

- b) Execute statements to create two new tables, `songs` and `playlists`.

Add to the `create_schema()` method:

```
self.session.execute("""
    CREATE KEYSPACE simplex WITH replication
        = {'class':'SimpleStrategy', 'replication_factor':3};
    """)
self.session.execute("""
    CREATE TABLE simplex.songs (
        id uuid PRIMARY KEY,
        title text,
        album text,
        artist text,
        tags set<text>,
        data blob
    );
    """)
self.session.execute("""
    CREATE TABLE simplex.playlists (
        id uuid,
        title text,
        album text,
        artist text,
        song_id uuid,
        PRIMARY KEY (id, title, album, artist)
    );
    """)
log.info('Simplex keyspace and schema created.')
```

3. Add an attribute function, `load_data()`, to the `SimpleClient` class implementation.

```
def load_data(self):
```

4. Add code to insert data into the new schema.

```
self.session.execute("""
    INSERT INTO simplex.songs (id, title, album, artist, tags)
    VALUES (
        756716f7-2e54-4715-9f00-91dcbea6cf50,
        'La Petite Tonkinoise',
        'Bye Bye Blackbird',
        'Joséphine Baker',
        {'jazz', '2013'}
    );
""")
self.session.execute("""
    INSERT INTO simplex.playlists (id, song_id, title, album, artist)
    VALUES (
        2cc9ccb7-6221-4ccb-8387-f22b6a1b354d,
        756716f7-2e54-4715-9f00-91dcbea6cf50,
        'La Petite Tonkinoise',
        'Bye Bye Blackbird',
        'Joséphine Baker'
    );
""")
log.info('Data loaded.')
```

5. In the class main function, add a call to the new `create_schema()` method.

```
client.create_schema()
```

6. Add a method, `query_schema()`, that executes a `SELECT` statement on the tables and then prints out the results.

- a) Add the method declaration.

```
def query_schema(self):
```

- b) Add code to execute the query.

Query the playlists table for one of the two records.

```
results = self.session.execute("""
    SELECT * FROM simplex.playlists
    WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6a1b354d;
""")
```

The execute method returns a `ResultSet` that holds rows returned by the `SELECT` statement.

- c) Add code to iterate over the rows and print them out.

```
print "%-30s\t%-20s\t%-20s\n%s" % \
    ("title", "album", "artist",
     "-----+-----")
+-----")
for row in results:
    print "%-30s\t%-20s\t%-20s" % (row.title, row.album, row.artist)
log.info('Schema queried.')
```

## Writing your first client

7. In the module main function, add a call to the new `query_schema()` method.

```
client.query_schema()
```

8. Run the module.

```
$ python clients.py
```

### Complete source code listing

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from cassandra.cluster import Cluster
import logging
import time

log = logging.getLogger()
log.setLevel('INFO')

class SimpleClient(object):
    session = None

    def connect(self, nodes):
        cluster = Cluster(nodes)
        metadata = cluster.metadata
        self.session = cluster.connect()
        log.info('Connected to cluster: ' + metadata.cluster_name)
        for host in metadata.all_hosts():
            log.info('Datacenter: %s; Host: %s; Rack: %s',
                    host.datacenter, host.address, host.rack)

    def close(self):
        self.session.cluster.shutdown()
        log.info('Connection closed.')

    def create_schema(self):
        self.session.execute("""CREATE KEYSPACE simplex WITH replication =
{'class': 'SimpleStrategy', 'replication_factor':3};""")
        self.session.execute("""
CREATE TABLE simplex.songs (
    id uuid PRIMARY KEY,
    title text,
    album text,
    artist text,
    tags set<text>,
    data blob
);
""")
        self.session.execute("""
CREATE TABLE simplex.playlists (
    id uuid,
    title text,
    album text,
    artist text,
    song_id uuid,
    PRIMARY KEY (id, title, album, artist)
);
""")
        log.info('Simplex keyspace and schema created.')
```

```

def load_data(self):
    self.session.execute("""
        INSERT INTO simplex.songs (id, title, album, artist, tags)
        VALUES (
            756716f7-2e54-4715-9f00-91dcbea6cf50,
            'La Petite Tonkinoise',
            'Bye Bye Blackbird',
            'Joséphine Baker',
            {'jazz', '2013'}
        );
    """)
    self.session.execute("""
artist)
        INSERT INTO simplex.playlists (id, song_id, title, album,
        VALUES (
            2cc9ccb7-6221-4ccb-8387-f22b6alb354d,
            756716f7-2e54-4715-9f00-91dcbea6cf50,
            'La Petite Tonkinoise',
            'Bye Bye Blackbird',
            'Joséphine Baker'
        );
    """)
    log.info('Data loaded.')

def query_schema(self):
    results = self.session.execute("""
SELECT * FROM simplex.playlists
WHERE id = 2cc9ccb7-6221-4ccb-8387-f22b6alb354d;
""")
    print "%-30s\t%-20s\t%-20s\n%s" % \
        ("title", "album", "artist",
        "-----+-----")
    +-----")
    for row in results:
        print "%-30s\t%-20s\t%-20s" % (row.title, row.album, row.artist)
    log.info('Schema queried.')

def main():
    logging.basicConfig()
    client = SimpleClient()
    client.connect(['192.168.153.159'])
    client.create_schema()
    time.sleep(10)
    client.load_data()
    client.query_schema()
    client.close()

if __name__ == "__main__":
    main()

```

## Using prepared statements

The previous tutorial used simple CQL statements to read and write data, but you can also use prepared statements, which only need to be parsed once by the cluster, and then bind values to the variables and execute the prepared statement you read or write data to a cluster.

## Writing your first client

### About this task

In the previous tutorial, you added a `load_data` function which creates a new statement for each INSERT, but you may also use prepared statements and bind new values to the columns each time before execution. Doing this increases performance, especially for repeated queries. You add code to your client for:

- creating a prepared statement
- creating a bound statement from the prepared statement and binding values to its variables
- executing the bound statement to insert data

### Procedure

1. In your clients module, add a new class, `BoundStatementsClient`, which extends `SimpleClass`.

```
class BoundStatementsClient(SimpleClient):
```

2. Add an attribute method, `prepare_statements()`, and implement it.

- a) Add the function declaration.

```
def prepare_statements():
```

- b) Add two attributes, `insert_song_prepared_statement` and `insert_playlist_prepared_statement` to hold references to the two prepared statements you will use in the `load_data()` method.

- c) In the `prepare_statements` method body add the following code:

```
self.insert_song_prepared_statement = self.session.prepare(
    """
    INSERT INTO simplex.songs
    (id, title, album, artist, tags)
    VALUES (?, ?, ?, ?, ?);
    """
)
self.insert_playlist_prepared_statement = self.session.prepare(
    """
    INSERT INTO simplex.playlists
    (id, song_id, title, album, artist)
    VALUES (?, ?, ?, ?, ?);
    """
)
```

**Note:** You only need to prepare a statement once per session.

3. Override the `load_data` function and implement it.

- a) Add the function declaration.

```
def load_data(self):
```

- b) Add code to bind values to the prepared statement's variables and execute it. You use the `bind` function to bind values to the bound statement and then execute the bound statement on the your session object..

```
tags = set(['jazz', '2013'])
self.session.execute(insert_song_prepared_statement,
    [ UUID("756716f7-2e54-4715-9f00-91dcbea6cf50"),
      "La Petite Tonkinoise",
      "Bye Bye Blackbird",
      "Joséphine Baker",
      tags ]
)
```



```
)
```

Note that you cannot pass in string representations of UUIDs or sets as you did in the `load_data` function.

4. Add code to insert data into the `simplex.playlists` table.

```
self.session.execute(bound_statement,
    [ UUID("2cc9ccb7-6221-4ccb-8387-f22b6a1b354d"),
      UUID("756716f7-2e54-4715-9f00-91dcbea6cf50"),
      "La Petite Tonkinoise",
      "Bye Bye Blackbird",
      "Joséphine Baker" ]
)
```

5. Add a call in the `main()` function to `prepare_statements` and `load_data()`.

```
def main():
    logging.basicConfig()
    client = BoundStatementsClient()
    client.connect(['127.0.0.1'])
    client.create_schema()
    client.prepare_statements()
    client.load_data()
    client.query_schema()
    client.update_schema()
    client.drop_schema("simplex")
    client.close()
```

# Python driver reference

Reference for the Python driver.

## Asynchronous I/O

You can execute statements on a session object in two different ways. Calling `execute()` blocks the calling thread until the statement finishes executing, but a session also allows for asynchronous and non-blocking I/O by calling the `execute_async()` function.

### About this task

Create a super class of the `SimpleClient` class and execute queries asynchronously on a cluster.

### Procedure

1. Add a new class, `AsynchronousExample`, to your clients module. It should extend the `SimpleClient` class.

```
class AsynchronousExample(SimpleClient):
```

2. Override the attribute function, `query_schema()`, and implement it.

- a) Add the function declaration.

```
def query_schema(self):
```

- b) Execute the query asynchronously and store the returned `ResponseFuture` object.

```
response_future = self.session.execute_async("SELECT * FROM
simplex.songs;")
```

- c) register two callback functions with the `ResponseFuture` object.

```
response_future.add_callbacks(print_results, print_errors)
```

3. Implement the two callback functions, `log_errors()` and `log_results()`.

```
def log_errors(errors):
    log.error(errors)
```

```
def print_results(results):
    print "%-30s\t%-20s\t%-20s%-30s\n%s" % \
          ("title", "album", "artist",
           "tags", "-----")
    +-----+-----
    +-----")
    for row in results:
        print "%-30s\t%-20s\t%-20s%-30s" % \
              (row.title, row.album, row.artist, row.tags)
```

4. In the module `main` function.
  - a) Instantiate an `AsynchronousExample` object.
  - b) Connect to your cluster.
  - c) Create the schema and load the data.

- d) Add a call to your new `query_schema()` function.
- a) Drop the schema and close the connection.

```
def main():
    logging.basicConfig()
    client = AsynchronousExample()
    client.connect(['127.0.0.1'])
    client.create_schema()
    client.load_data()
    client.query_schema()
    client.drop_schema('simplex')
    client.close()
```

Of course, in our implementation, the call to `getUninterruptibly` blocks until the result set future has completed execution of the statement on the session object. Functionally it is no different from executing the `SELECT` query synchronously.

### AsynchronousExample code listing

```
def print_errors(errors):
    log.error(errors)

def print_results(results):
    print "%-30s\t%-20s\t%-20s%-30s\n%s" % \
        ("title", "album", "artist",
         "tags", "-----+-----")
    +-----+-----")
    for row in results:
        print "%-30s\t%-20s\t%-20s%-30s" % \
            (row.title, row.album, row.artist, row.tags)

class AsynchronousExample(SimpleClient):
    def query_schema(self):
        future_results = self.session.execute_async("SELECT * FROM
simplex.songs;")
        future_results.add_callbacks(print_results, print_errors)

#

def main():
    logging.basicConfig()
    client = AsynchronousExample()
    client.connect(['127.0.0.1'])
    client.create_schema()
    client.load_data()
    client.query_schema()
    client.drop_schema('simplex')
    client.close()

if __name__ == "__main__":
    main()
```

### Automatic failover

If a Cassandra node fails or becomes unreachable, the Python driver automatically and transparently tries other nodes in the cluster and schedules reconnections to the dead nodes in the background.

### Description

How the driver handles failover is determined by which retry and reconnection policies are used when building a cluster object.

### Examples

This code illustrates building a cluster instance with a retry policy which sometimes retries with a lower consistency level than the one specified for the query.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from cassandra.cluster import Cluster
from cassandra.policies import DowngradingConsistencyRetryPolicy
from cassandra.policies import ConstantReconnectionPolicy

class RollYourOwnCluster:
    cluster = None
    session = None

    def __init__(self):
        self.cluster = Cluster(
            contact_points=['127.0.0.1', '127.0.0.2'],
            default_retry_policy=DowngradingConsistencyRetryPolicy(),
            reconnection_policy=ConstantReconnectionPolicy(20.0, 10)
        )
        self.session = self.cluster.connect()

#

def main():
    client = RollYourOwnCluster()
    print('Connected to cluster: ' + client.cluster.metadata.cluster_name)
    client.session.shutdown()

if __name__ == "__main__":
    main()
```

## CQL data types to Python types

A summary of the mapping between CQL data types and Python data types is provided.

### Description

When retrieving the value of a column from a `Row` object, you use a getter based on the type of the column.

**Table 1: Python types to CQL data types**

CQL data type	Python type
ascii	str
bigint	int, long
blob	buffer, bytearray
boolean	bool

CQL data type	Python type
counter	int, long
decimal	decimal
double	float
float	float
inet	str
int	int
list	list, tuple, generator
map	dict, OrderedDict
set	set, frozenset
text	str
timestamp	datetime.datetime
timeuuid	uuid.UUID
uuid	uuid.UUID
varchar	str
varint	int, long

### CQL date and time data types

New CQL `date` and `time` data types are being introduced into an upcoming version of Cassandra. This version of the driver includes support for those new types. Results with these types are returned in new types provided in the `cassandra.util` module. While these types are similar to the built-in `datetime.time` and `datetime.date` types, custom types were required to work around limitations in precision (nanosecond for times) and range (outside the `datetime.MINYEAR` and `datetime.MAXYEAR` constants for dates). The new types are described in the [API documentation](#).

CQL data type	Python data type
date	datetime.date
time	datetime.time
timestamp	datetime.datetime

For non-prepared statements, the driver previously had an overloaded mapping, encoding `dates` as `timestamps` in CQL. That overload is removed in support of these new date types. Conveniently, CQL `date` literals are also acceptable `timestamps`, so this updated mapping should not require any application updates.

### Time utility functions

This release includes a number of utility functions for converting between (type 1) `TimeUUID` types and native time representations:

```
cassandra.util.unix_time_from_uuid1
cassandra.util.datetime_from_uuid1
cassandra.util.min_uuid_from_time
cassandra.util.max_uuid_from_time
```

## Python driver reference

```
cassandra.util.uuid_from_time
```

These are described in more detail in the util [API documentation](#). The `datetime_from_timestamp` function has been moved from its previous location in `cassandra.cqltypes` to the `util` module. Although not officially part of the API, the old function remains as a pass-through (deprecated until the next release).

## Debugging

### Enabling debug-level logging

Use the standard [Python logging handlers](#).

#### Logging example

Logs debug-level statements to both the console and a file

```
from cassandra.cluster import Cluster
import logging

log = logging.getLogger()
log.setLevel('DEBUG')

fh = logging.FileHandler('mypyclient.log')
fh.setLevel('DEBUG')

ch = logging.StreamHandler()
ch.setLevel('DEBUG')

log.addHandler(fh)
log.addHandler(ch)

class SimpleClient(object):
    session = None

    def connect(self, nodes):
        cluster = Cluster(nodes)
        metadata = cluster.metadata
        self.session = cluster.connect()
        log.info('Connected to cluster: ' + metadata.cluster_name)
        for host in metadata.all_hosts():
            log.info('Datacenter: %s; Host: %s; Rack: %s', host.datacenter,
                    host.address, host.rack)

    def close(self):
        self.session.cluster.shutdown()
        log.info('Connection closed.')

def main():
    logging.basicConfig()
    client = SimpleClient()
    client.connect(['127.0.0.1'])
    client.close()

if __name__ == "__main__":
    main()
```

#### Output

```

Connecting to cluster, contact points: ['127.0.0.1']; protocol version: 2
Host 127.0.0.1 is now marked up
[control connection] Opening new connection to 127.0.0.1
Not sending options message for new connection(29244048) to 127.0.0.1 because
  compression is disabled and a cql version was not specified
Sending StartupMessage on <LibevConnection(29244048) 127.0.0.1:9042>
Sent StartupMessage on <LibevConnection(29244048) 127.0.0.1:9042>
Starting libev event loop
Got ReadyMessage on new connection (29244048) from 127.0.0.1
[control connection] Established new connection <LibevConnection(29244048)
 127.0.0.1:9042>, registering watchers and refreshing schema and topology
[control connection] Refreshing node list and token map using preloaded
  results
[control connection] Found new host to connect to: 172.31.15.148
New Cassandra host <Host: 172.31.15.148 Solr> discovered
Handling new host <Host: 172.31.15.148 Solr> and notifying listeners
Done preparing queries for new host <Host: 172.31.15.148 Solr>
Host 172.31.15.148 is now marked up
[control connection] Finished fetching ring info
[control connection] Rebuilding token map due to topology changes
[control connection] Attempting to use preloaded results for schema agreement
[control connection] Schemas match
[control connection] user types table not found
[control connection] Fetched schema, rebuilding metadata
Control connection created
Initializing new connection pool for host 172.31.15.148
Initializing new connection pool for host 127.0.0.1

```

## Mapping user-defined types

Cassandra 2.1 introduces support for [User-defined types](#) (UDT). A user-defined type simplifies handling a group of related properties.

A quick example is a user account table that contains address details described through a set of columns: street, city, zip code. With the addition of UDTs, you can define this group of properties as a type and access them as a single entity or separately.

User-defined types are declared at the keyspace level.

In your application, you map your UDTs to application entities. For example, given the following UDT:

```

CREATE TYPE complex.address (
  street text,
  city text,
  zip int,
  phones list<text>
);

```

You create a Python class to map it to:

```

class Address:
    street = None
    city = None
    zip_code = None
    phones = None

    def __init__(self, street = None, city = None, zip_code = None, phones =
None):
        self.street = street

```

```
self.city = city
self.zip_code = zip_code
self.phones = phones
```

### Using the UDT in your client

This example builds on the SimpleClient application [developed earlier](#).

In your application, you then map the UDT by class and use it:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from clients import SimpleClient

from uuid import UUID

class Address:
    street = None
    city = None
    zip_code = None
    phones = None

    def __init__(self, street = None, city = None, zip_code = None, phones =
None):
        self.street = street
        self.city = city
        self.zip_code = zip_code
        self.phones = phones

class UsingUDTs(SimpleClient):
    def load_data(self):
        self.session.execute(
            """
                CREATE KEYSPACE complex WITH replication
                    = {'class':'SimpleStrategy', 'replication_factor':3};
            """)
        self.session.execute(
            """
                CREATE TYPE complex.address (
                    street text,
                    city text,
                    zip_code int,
                    phones list<text>);
            """)
        self.session.execute(
            """
                CREATE TABLE complex.users (
                    id uuid PRIMARY KEY,
                    name text,
                    addresses map<text, frozen<address>>);
            """)

    def insert_address(self):
        # register UDT
        self.cluster.register_user_type("complex", "address", Address)
        # use the UDT in a query
        self.insert_user_prepared_statement = self.session.prepare(
            """
                INSERT INTO complex.users (id, name, addresses)
                VALUES (:id, :name, :addresses);
            """)
```



```

        address = Address('123 Arnold Drive', 'Sonoma', 95476,
['707-555-1234', '800-555-9876'])
        addresses = { 'Home' : address }
        bound_statement = self.insert_user_prepared_statement.bind(
            { 'id' : UUID('756716f7-2e54-4715-9f00-91dcbea6cf50'), 'name' :
'John Doe', 'addresses' : addresses }
        )
        self.session.execute(bound_statement)

def main():
    client = UsingUDTs()
    client.connect(['127.0.0.1'])
    client.load_data()
    client.insert_address()

if __name__ == "__main__":
    main()

```

## Metadata refresh controls

As of version 2.5, the driver includes some new controls for specifying how (or if) cluster metadata is refreshed in response to native protocol events on the control connection. Two new Cluster attributes have been added:

```

schema_event_refresh_window
topology_event_refresh_window

```

This feature is of most interest to users operating the driver in a high fanout from Cassandra nodes to client instances. Previously, the driver would react to these events at a fixed interval, in effect causing a *thundering herd* problem as many instances refreshed parts of the metadata at the same time. Now, this reaction is smoothed out by randomizing the interval within a configured window. Additionally, if redundant events arrive inside of that window, the driver is able to de-duplicate refreshes.

Finally, these controls are used to disable refreshes for one or both types of events. This may be useful for high-fanout deployments where the pertinent schema is static, or the application is not expected to react to schema changes without restart.

More details on these parameters can be found in the [API documentation](#).

## Named parameters

You can bind the parameters in a `BoundStatement` or `SimpleStatement` either by the marker position or by named marker. The example here builds on the `BoundStatementsClient` [developed earlier](#).

```

from clients import BoundStatementsClient

from uuid import UUID

class ByPositionAndByName(BoundStatementsClient):
    def load_data_bound_by_position(self):
        self.insert_user_prepared_statement = self.session.prepare(
            """
            INSERT INTO simplex.songs (id, title, album, artist)
            VALUES (?, ?, ?, ?);
            """
        )
        # bind parameters by position

```

## Python driver reference

```
self.session.execute(self.insert_user_prepared_statement,
    [ UUID("756716f7-2e54-4715-9f00-91dcbea6cf50"),
      "Lazing on a Sunday Afternoon",
      "A Night at the Opera",
      "Queen"
    ]
)

def load_data_bound_by_name(self):
    self.insert_user_prepared_statement = self.session.prepare(
        """
        INSERT INTO simplex.songs (id, title, album, artist)
        VALUES (:id, :title, :album, :artist);
        """
    )
    # bind parameters by name
    bound_statement = self.insert_user_prepared_statement.bind(
        { 'id' : UUID('e0e03270-1e6f-11e4-8c21-0800200c9a66'),
          'title' : 'Coconut',
          'album' : 'Nilsson Schmilsson',
          'artist' : 'Harry Nilsson' }
    )
    self.session.execute(bound_statement)

def main():
    client = ByPositionAndByName()
    client.connect(['127.0.0.1'])
    client.create_schema()
    client.load_data_bound_by_position()
    client.load_data_bound_by_name()

if __name__ == "__main__":
    main()
```

## Object-mapping package

The driver includes `cqlengine`, a CQL object-mapping package for Python.

The `cqlengine` package was a standalone project, but it is now integrated into the driver.

Upgrading to the integrated `cqlengine` should be painless and simple. The existing API and functionality of `cqlengine` has been retained. Most applications will only need to remove the legacy package and update import locations. There is one minor functional change and a number of deprecations. Anyone upgrading is encouraged to consult the [upgrade guide](#) for details.

While efforts were taken to leave most of the existing functionality unchanged, there are a few new features. Most notably, there is now support for modeling [User Defined Types](#) (introduced in Cassandra 2.1). There are also a number of new functions for simplified connection and keyspace management. Please consult the [API documentation](#) for further detail.

## Passing parameters to CQL queries

When executing non-prepared statements, the driver supports two forms of parameter place-holders: positional and named.

### Positional

Positional parameters are used with a `%s` placeholder. For example, when you execute:

```
session.execute (
```

```

"""
INSERT INTO users (name, credits, user_id)
VALUES (%s, %s, %s)
""" ( "John O'Reilly" , 42 , uuid . uuid1 ()
)

```

It is translated to the following CQL query:

```

INSERT INTO users (name, credits, user_id)
VALUES ('John O'Reilly', 42, 2644bada-852c-11e3-89fb-e0b9a54a6d93)

```

You must use `%s` for all types of arguments, not just strings. For example, this would be wrong:

```

session.execute( "INSERT INTO USERS (name, age) VALUES ( %s, %d )",
( "bob", 42 )) # wrong

```

Instead, use `%s` for the age placeholder.

If you need to use a literal `%` character, use `%%`.

You must always use a sequence for the second argument, even if you are only passing in a single variable:

```

session.execute( "INSERT INTO foo (bar) VALUES ( %s )" , "blah" )      #
wrong
session.execute( "INSERT INTO foo (bar) VALUES ( %s )" , ( "blah" ))  #
wrong
session.execute( "INSERT INTO foo (bar) VALUES ( %s )" , ( "blah" , )) #
right
session.execute( "INSERT INTO foo (bar) VALUES ( %s )" , [ "blah" ])  #
right

```

**Note:** The second line is incorrect because in Python, single-element tuples require a comma.

## Named

Named place-holders use the `%(name)s` form:

```

session.execute(
    """
    INSERT INTO users (name, credits, user_id, username)
    VALUES (%(name)s, %(credits)s, %(user_id)s, %(name)s)
    """
    {'name' : "John O'Reilly" , 'credits' : 42 , 'user_id' : uuid . uuid1
    ()}
)

```

You can repeat placeholders with the same name, such as `%(name)s` in the above example, but only data values should be supplied this way. Other items, such as keyspace, table names, and column names should be set ahead of time (typically using normal string formatting).

## Tuple types

Cassandra 2.1 introduced the **tuple type** for CQL. For the following table: A tuple is a fixed-length set of typed positional fields.

```
CREATE TABLE tuple_test (
  the_key int PRIMARY KEY,
  the_tuple frozen<tuple< int, text, float>>)
```

You access the tuple column in Python like this:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from clients import SimpleClient

class TestTuples(SimpleClient):
    def create_schema(self):
        super(TestTuples, self).create_schema()
        self.session.execute(
            """
                CREATE TABLE simplex(tuple_test (
                    the_key int PRIMARY KEY,
                    the_tuple tuple<int, text, float>)
            """)

    def insert_tuple(self):
        the_tuple = (0, 'abc', 1.0)
        prepared = self.session.prepare("INSERT INTO
simplex(tuple_test(the_key, the_tuple) VALUES (?, ?);")
        self.session.execute(prepared, parameters=(1, the_tuple))

def main():
    client = TestTuples()
    client.connect(['127.0.0.1'])
    client.create_schema()
    client.insert_tuple()
    client.pause()
    client.drop_schema("simplex")

if __name__ == "__main__":
    main()
```

**Note:** This example builds off the SimpleClient application [developed earlier](#).

## FAQ

- Is there a distinction between different hosts?
- Are there plans to allow for a retry policy option at the `execute()` or `execute_async()` methods?

### Is there a distinction between different hosts?

For example, if you try an operation and the host goes down, do the retry policies re-use the same host or do they use another one? Is there any control over this decision? Presently retry is only done on the same host (not because it was down, but because it responded with a Coordinator timeout or unavailable exception).

### Are there plans to allow for a retry policy option for the `execute()` or `execute_async()` methods?

Currently I can use the same prepared statements in multiple contexts, but sometimes I may want to retry an insert a few times and other times it may be better to do so higher up in the application. The default policy is specified at the cluster level, but each statement can have its own.










# API reference

DataStax Python Driver for Apache Cassandra.

# Tips for using DataStax documentation

## Navigating the documents

To navigate, use the table of contents or search in the left navigation bar. Additional controls are:

	Hide or display the left navigation.
	Go back or forward through the topics as listed in the table of contents.
	Toggle highlighting of search terms.
	Print page.
	See doc tweets and provide feedback.
	Grab to adjust the size of the navigation pane.
	Appears on headings for bookmarking. Right-click the  to get the link.
	Toggles the legend for CQL statements and nodetool options.

## Other resources

You can find more information and help at:

- [Documentation home page](#)
- [Datasheets](#)
- [Webinars](#)
- [Whitepapers](#)
- [Developer blogs](#)
- [Support](#)