pymssql Documentation

Release 2.2.3.dev0+g58dae49.d20210908

pymssql developers

Sep 08, 2021

Contents

CHAPTER 1

Introduction

1.1 Getting started

Generally, you will want to install pymssql with:

```
pip install -U pip
pip install pymssql
```

Most of the times this should be all what's needed.

> **Note:** On some Linux distributions *pip* version is too old to support all the flavors of manylinux wheels, so upgrading *pip* is necessary. An example of such distributions would be Ubuntu 18.04 or Python3.6 module in RHEL8 and CentOS8.

> **Note:** Starting with pymssql version 2.1.3 we provide such wheel packages that bundle a static copy of FreeTDS so no additional dependency download or compilation steps are necessary.

> **Note:** Starting with pymssql version 2.2.0 official pymssql wheel packages for Linux, Mac OS and Windows have SSL support so they can be used to connect to *Azure*.

- Anaconda / Miniconda

  A conda install of pymssql will mitigate the need to edit config files outside of the user's home directory on some unix-like systems. This is especially useful when root access is restricted and/or Homebrew can't be installed. This method requires no additional compilation or configuration.

  ```
  conda install pymssql
  ```

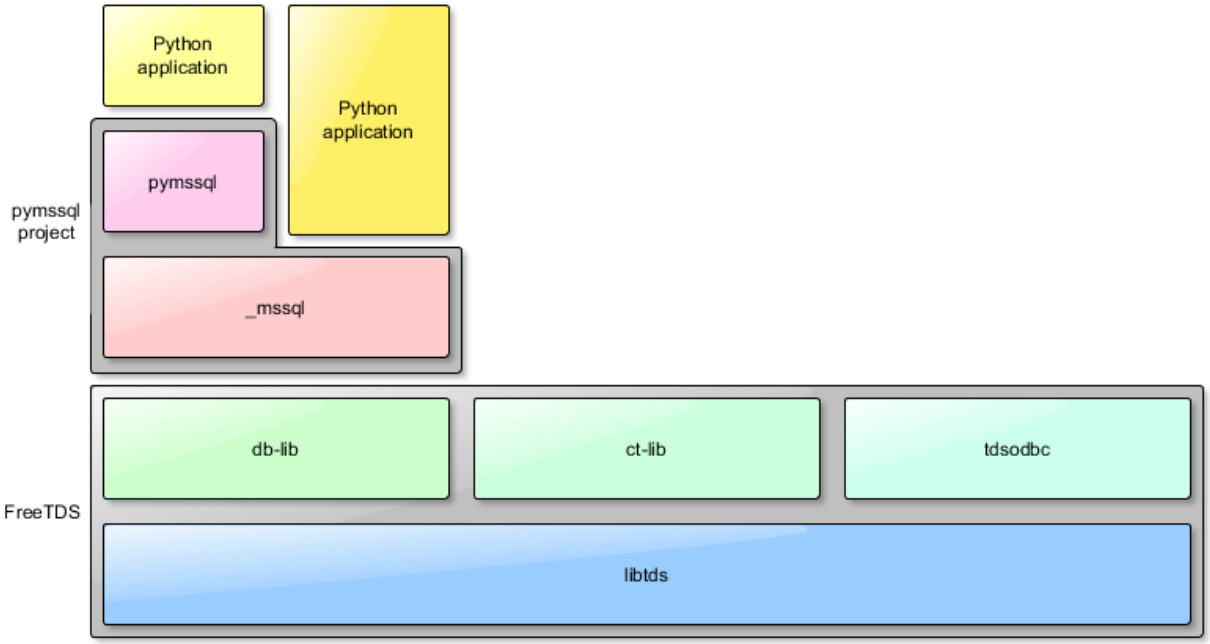
See Installation and *Building and developing pymssql* for more advanced scenarios.

Docker

(Experimental)

Another possible way to get started quickly with pymssql is to use a *Docker* image.

1.2 Architecture



The pymssql package consists of two modules:

- *pymssql* – use it if you care about DB-API compliance, or if you are accustomed to DB-API syntax,
- *_mssql* – use it if you care about performance and ease of use (`_mssql` module is easier to use than `pymssql`).

And, as of version 2.1.x it uses the services of the `db-lib` component of FreeTDS. See the relevant FreeTDS documentation for additional details.

1.3 Supported related software

Python Python 3.x: 3.6 or newer.

FreeTDS 1.2.18 or newer.

Cython 0.29 or newer.

Microsoft SQL Server 2005 or newer.

1.4 Project Discussion

Discussions and support take place on pymssql mailing list here: http://groups.google.com/group/pymssql, you can participate via web, e-mail or read-only subscribing to the mailing list feeds.

This is the best place to get help, please feel free to drop by and ask a question.

1.5 Project Status

Current release: 2.x is the branch under current development. It is a complete rewrite using Cython and the latest FreeTDS libraries (which remove many of the limitations of previous versions of FreeTDS).

Legacy release: 1.0.3 is the legacy version and is no longer under active development.

Note: This documentation is for pymssql 2.x.

The document set you are reading describes exclusively the code base of pymssql 2.x and newer. All description of functionality, workarounds, limitations, dependencies, etc. of older revisions has been removed.

If you need help for building/using pymssql 1.x please refer to the old Google Code documentation Wiki.

1.6 Current Development

Official development repositories and issue trackers have been moved to GitHub at https://github.com/pymssql/pymssql.

We would be happy to have:

- A couple more developers
- Help from the community with maintenance of this documentation.

If interested, please connect with us on the mailing list.

CHAPTER 2

pymssql examples

Example scripts using pymssql module.

2.1 Basic features (strict DB-API compliance)

```
from os import getenv
import pymssql

server = getenv("PYMSSQL_TEST_SERVER")
user = getenv("PYMSSQL_TEST_USERNAME")
password = getenv("PYMSSQL_TEST_PASSWORD")

conn = pymssql.connect(server, user, password, "tempdb")
cursor = conn.cursor()
cursor.execute("""
IF OBJECT_ID('persons', 'U') IS NOT NULL
    DROP TABLE persons
CREATE TABLE persons (
    id INT NOT NULL,
    name VARCHAR(100),
    salesrep VARCHAR(100),
    PRIMARY KEY(id)
)
""")
cursor.executemany(
    "INSERT INTO persons VALUES (%d, %s, %s)",
    [(1, 'John Smith', 'John Doe'),
     (2, 'Jane Doe', 'Joe Dog'),
     (3, 'Mike T.', 'Sarah H.')])
# you must call commit() to persist your data if you don't set autocommit to True
conn.commit()

cursor.execute('SELECT * FROM persons WHERE salesrep=%s', 'John Doe')
```

(continues on next page)

(continued from previous page)

```
row = cursor.fetchone()
while row:
    print("ID=%d, Name=%s" % (row[0], row[1]))
    row = cursor.fetchone()

conn.close()
```

2.2 Connecting using Windows Authentication

When connecting using Windows Authentication, this is how to combine the database's hostname and instance name, and the Active Directory/Windows Domain name and the username. This example uses raw strings (`r'...'`) for the strings that contain a backslash.

```
conn = pymssql.connect(
    host=r'dbhostname\myinstance',
    user=r'companydomain\username',
    password=PASSWORD,
    database='DatabaseOfInterest'
)
```

2.3 Iterating through results

You can also use iterators instead of while loop.

```
conn = pymssql.connect(server, user, password, "tempdb")
cursor = conn.cursor()
cursor.execute('SELECT * FROM persons WHERE salesrep=%s', 'John Doe')

for row in cursor:
    print('row = %r' % (row,))

conn.close()
```

Note: Iterators are a pymssql extension to the DB-API.

2.4 Important note about Cursors

A connection can have only one cursor with an active query at any time. If you have used other Python DBAPI databases, this can lead to surprising results:

```
c1 = conn.cursor()
c1.execute('SELECT * FROM persons')

c2 = conn.cursor()
c2.execute('SELECT * FROM persons WHERE salesrep=%s', 'John Doe')

print( "all persons" )
```

(continues on next page)

(continued from previous page)

```
print( c1.fetchall() )  # shows result from c2 query!

print( "John Doe" )
print( c2.fetchall() )  # shows no results at all!
```

In this example, the result printed after `"all persons"` will be the result of the *second* query (the list where `salesrep='John Doe'`) and the result printed after "John Doe" will be empty. This happens because the underlying TDS protocol does not have client side cursors. The protocol requires that the client flush the results from the first query before it can begin another query.

(Of course, this is a contrived example, intended to demonstrate the failure mode. Actual use cases that follow this pattern are usually much more complicated.)

Here are two reasonable workarounds to this:

- Create a second connection. Each connection can have a query in progress, so multiple connections can execute multiple conccurent queries.
- use the fetchall() method of the cursor to recover all the results before beginning another query:

```
c1.execute('SELECT ...')
c1_list = c1.fetchall()

c2.execute('SELECT ...')
c2_list = c2.fetchall()

# use c1_list and c2_list here instead of fetching individually from
# c1 and c2
```

2.5 Rows as dictionaries

Rows can be fetched as dictionaries instead of tuples. This allows for accessing columns by name instead of index. Note the `as_dict` argument.

```
conn = pymssql.connect(server, user, password, "tempdb")
cursor = conn.cursor(as_dict=True)

cursor.execute('SELECT * FROM persons WHERE salesrep=%s', 'John Doe')
for row in cursor:
    print("ID=%d, Name=%s" % (row['id'], row['name']))

conn.close()
```

Note: The `as_dict` parameter to `cursor()` is a pymssql extension to the DB-API.

2.6 Using the `with` statement (context managers)

You can use Python's `with` statement with connections and cursors. This frees you from having to explicitly close cursors and connections.

```
with pymssql.connect(server, user, password, "tempdb") as conn:
    with conn.cursor(as_dict=True) as cursor:
        cursor.execute('SELECT * FROM persons WHERE salesrep=%s', 'John Doe')
        for row in cursor:
            print("ID=%d, Name=%s" % (row['id'], row['name']))
```

Note: The context manager personality of connections and cursor is a pymssql extension to the DB-API.

2.7 Calling stored procedures

As of pymssql 2.0.0 stored procedures can be called using the rpc interface of db-lib.

```
with pymssql.connect(server, user, password, "tempdb") as conn:
    with conn.cursor(as_dict=True) as cursor:
        cursor.execute("""
        CREATE PROCEDURE FindPerson
            @name VARCHAR(100)
        AS BEGIN
            SELECT * FROM persons WHERE name = @name
        END
        """)
        cursor.callproc('FindPerson', ('Jane Doe',))
        for row in cursor:
            print("ID=%d, Name=%s" % (row['id'], row['name']))
```

2.8 Using pymssql with cooperative multi-tasking systems

New in version 2.1.0.

You can use the *pymssql.set_wait_callback()* function to install a callback function you should write yourself.

This callback can yield to another greenlet, coroutine, etc. For example, for gevent, you could use its `gevent.socket.wait_read()` function:

```
import gevent.socket
import pymssql

def wait_callback(read_fileno):
    gevent.socket.wait_read(read_fileno)

pymssql.set_wait_callback(wait_callback)
```

The above is useful if you're say, running a Gunicorn server with the gevent worker. With this callback in place, when you send a query to SQL server and are waiting for a response, you can yield to other greenlets and process other requests. This is super useful when you have high concurrency and/or slow database queries and lets you use less Gunicorn worker processes and still handle high concurrency.

Note: set_wait_callback() is a pymssql extension to the DB-API 2.0.

2.9 Bulk copy

New in version 2.2.0.

The fastest way to insert data to a SQL Server table is often to use the bulk copy functions, for example:

```
conn = pymssql.connect(server, user, password, "tempdb")
cursor = conn.cursor()
cursor.execute("""
    CREATE TABLE example (
        col1 INT NOT NULL,
        col2 INT NOT NULL
    )
""")
cursor.close()

conn.bulk_copy("example", [(1, 2)] * 1000000)
```

For more detail on fast data loading in SQL Server, including on bulk copy, read The data loading performance guide from Microsoft.

CHAPTER 3

_mssql examples

Example scripts using _mssql module.

3.1 Quickstart usage of various features

```
from pymssql import _mssql
conn = _mssql.connect(server='SQL01', user='user', password='password', \
    database='mydatabase')
conn.execute_non_query('CREATE TABLE persons(id INT, name VARCHAR(100))')
conn.execute_non_query("INSERT INTO persons VALUES(1, 'John Doe')")
conn.execute_non_query("INSERT INTO persons VALUES(2, 'Jane Doe')")
```

```
# how to fetch rows from a table
conn.execute_query('SELECT * FROM persons WHERE salesrep=%s', 'John Doe')
for row in conn:
    print "ID=%d, Name=%s" % (row['id'], row['name'])
```

New in version 2.1.0: Iterating over query results by iterating over the connection object just like it's already possible with pymssql connections is new in 2.1.0.

```
# examples of other query functions
numemployees = conn.execute_scalar("SELECT COUNT(*) FROM employees")
numemployees = conn.execute_scalar("SELECT COUNT(*) FROM employees WHERE name LIKE 'J%
↪'")    # note that '%' is not a special character here
employeedata = conn.execute_row("SELECT * FROM employees WHERE id=%d", 13)
```

```
# how to fetch rows from a stored procedure
conn.execute_query('sp_spaceused')    # sp_spaceused without arguments returns 2␣
↪result sets
res1 = [ row for row in conn ]        # 1st result
res2 = [ row for row in conn ]        # 2nd result
```

```
# how to get an output parameter from a stored procedure
sqlcmd = """
DECLARE @res INT
EXEC usp_mystoredproc @res OUT
SELECT @res
"""
res = conn.execute_scalar(sqlcmd)
```

```
# how to get more output parameters from a stored procedure
sqlcmd = """
DECLARE @res1 INT, @res2 TEXT, @res3 DATETIME
EXEC usp_getEmpData %d, %s, @res1 OUT, @res2 OUT, @res3 OUT
SELECT @res1, @res2, @res3
"""
res = conn.execute_row(sqlcmd, (13, 'John Doe'))
```

```
# examples of queries with parameters
conn.execute_query('SELECT * FROM empl WHERE id=%d', 13)
conn.execute_query('SELECT * FROM empl WHERE name=%s', 'John Doe')
conn.execute_query('SELECT * FROM empl WHERE id IN (%s)', ((5, 6),))
conn.execute_query('SELECT * FROM empl WHERE name LIKE %s', 'J%')
conn.execute_query('SELECT * FROM empl WHERE name=%(name)s AND city=%(city)s', \
    { 'name': 'John Doe', 'city': 'Nowhere' } )
conn.execute_query('SELECT * FROM cust WHERE salesrep=%s AND id IN (%s)', \
    ('John Doe', (1, 2, 3)))
conn.execute_query('SELECT * FROM empl WHERE id IN (%s)', (tuple(xrange(4)),))
conn.execute_query('SELECT * FROM empl WHERE id IN (%s)', \
    (tuple([3, 5, 7, 11]),))
```

```
conn.close()
```

Please note the usage of iterators and ability to access results by column name. Also please note that parameters to connect method have different names than in `pymssql` module.

3.2 An example of exception handling

```
from pymssql import _mssql

conn = _mssql.connect(server='SQL01', user='user', password='password',
                      database='mydatabase')
try:
    conn.execute_non_query('CREATE TABLE t1(id INT, name VARCHAR(50))')
except _mssql.MssqlDatabaseException as e:
    if e.number == 2714 and e.severity == 16:
        # table already existed, so quieten the error
    else:
        raise # re-raise real error
finally:
    conn.close()
```

3.3 Custom message handlers

New in version 2.1.1.

You can provide your own message handler callback function that will be invoked by the stack with informative messages sent by the server. Set it on a per `_mssql` *connection* basis by using the *_mssql.MSSQLConnection.set_msghandler()* method:

```
from pymssql import _mssql

def my_msg_handler(msgstate, severity, srvname, procname, line, msgtext):
    """
    Our custom handler -- It simpy prints a string to stdout assembled from
    the pieces of information sent by the server.
    """
    print("my_msg_handler: msgstate = %d, severity = %d, procname = '%s', "
          "line = %d, msgtext = '%s'" % (msgstate, severity, procname,
                                         line, msgtext))

cnx = _mssql.connect(server='SQL01', user='user', password='password')
try:
    cnx.set_msghandler(my_msg_handler)  # Install our custom handler
    cnx.execute_non_query("USE mydatabase")  # It gets called at this point
finally:
    cnx.close()
```

Something similar to this would be printed to the standard output:

```
my_msg_handler: msgstate = x, severity = y, procname = '', line = 1, msgtext =
↪'Changed database context to 'mydatabase'.'
```

Todo: Add an example of invoking a Stored Procedure using `_mssql`.

CHAPTER 4

Release notes

Release notes – All breaking changes and other noteworthy things.

4.1 pymssql 2.0.0

This is a new major version of pymssql. It is totally rewritten from scratch in Cython. Our goals for this version were to:

- Provide support for Python 3.0 and newer,
- Implement support for stored procedures,
- Rewrite DB-API compilant pymssql module in C (actually in Cython) for increased performance,
- Clean up the module API and the code.

That's why we decided to bump major version number. Unfortunately new version introduces incompatible changes in API. Existing scripts may not work with it, and you'll have to audit them. If you care about compatibility, just continue using pymssql 1.0.x and slowly move to 2.0.

Project hosting has also changed. Now pymssql is developed on GitHub: http://github.com/pymssql/pymssql.

Credits for the release go to:

- Marc Abramowitz <msabramo_at_gmail_com> who joined the project in Jan 2013 and is responsible for the actual release of the 2.0 version by fixing many old tickets, coding the port to Python 3 and driving the migration to Git and GitHub.
- Randy Syring who converted the repository to Mercurial, extended tests and ported them to nose, enhanced the code in several fronts like multi-platform (compilers, OSes) compatibility, error handling, support of new data types, SQLAlchemy compatibility and expanded the documentation.
- Damien Churchill <damoxc_at_gmail_com> who set the foundations of the new Cython-based code base, release engineering, new site features like Sphinx, SimpleJSON and others,
- Andrzej Kukuła <akukula_at_gmail_com> who did all the docs, site migration, and other boring but necessary stuff.

- Jooncheol Park <jooncheol_at_gmail_com> who did develop the initial version of pymssql (until 0.5.2). Now just doing boring translation docs for Korean.

4.1.1 `pymssql` module

- Rewritten from scratch in C, you should observe even better performance than before
- `dsn` parameter to *`pymssql.connect()`* has been removed
- `host` parameter to *`pymssql.connect()`* has been renamed to `server` to be consistent with `_mssql` module
- `max_conn` parameter to *`pymssql.connect()`* has been removed

`Connection` class

- `autocommit()` function has been changed to *`pymssql.Connection.autocommit`* property that you can set or get its current state.

`Cursor` class

- `fetchone_asdict()` method has been removed. Just use *`pymssql.connect()`* with `as_dict=True`, then use regular *`fetchone()`*
- `fetchmany_asdict()` method has been removed. Just use *`pymssql.connect()`* with `as_dict=True`, then use regular *`fetchmany()`*
- `fetchall_asdict()` method has been removed. Just use *`pymssql.connect()`* with `as_dict=True`, then use regular *`fetchall()`*

4.1.2 `_mssql` module

- Added native support for stored procedures (*`MSSQLStoredProcedure`* class)
- `maxconn` parameter to `_mssql.connect()` has been removed
- `timeout` and `login_timeout` parameter to `_mssql.connect()` has been added
- *`get_max_connections()`* and *`set_max_connections()`* module-level methods have been added
- Class names have changed:

Old Name	New name
MssqlException	MSSQLException
MssqlDriverException	MSSQLDriverException
MssqlDatabaseException	MSSQLDatabaseException
MssqlRowIterator	MSSQLRowIterator
MssqlConnection	MSSQLConnection

`MSSQLConnection` class

- Added *`tds_version`* property.

CHAPTER 5

FreeTDS

5.1 Installation

5.1.1 Linux

On Linux you can choose between (for the two former choices, when you start the the pymssql installation process it will look for and pick the header files and libraries for FreeTDS in some usual system-wide locations):

- Use the FreeTDS installation provided by the packages/ports system.
- Build it and install yourself.
- Use the bundled static FreeTDS libraries:

```
export PYMSSQL_BUILD_WITH_BUNDLED_FREETDS=1
pip install pymssql
```

These static libraries are built on a x86_64 Ubuntu 14.04 system by using the following sequence:

```
export CFLAGS="-fPIC"  # for the 64 bits version
```

or

```
export CFLAGS="-m32 -fPIC" LDFLAGS="-m32"  # for the 32 bits version
```

and then:

```
./configure --enable-msdblib \
  --prefix=/usr --sysconfdir=/etc/freetds --with-tdsver=7.1 \
  --disable-apps --disable-server --disable-pool --disable-odbc \
  --with-openssl=no --with-gnutls=no
make
```

Changed in version 2.1.3: Version of FreeTDS Linux static libraries bundled with pymssql is 0.95.95.

Changed in version 2.1.2: Version of FreeTDS Linux static libraries bundled with pymssql is 0.95.81 obtained from branch Branch-0_95 of the official Git repository. Up to 2.1.1 the version of FreeTDS bundled was 0.91.

5.1.2 Mac OS X (with Homebrew)

```
brew install freetds
```

5.1.3 Windows

You can:

1. Simply use our official Wheels which include FreeTDS statically linked and have no SSL support.
2. Build pymssql yourself. In this case you have the following choices regarding FreeTDS:
   - Use binaries we maintain at https://github.com/ramiro/freetds/releases

     Choose the .zip file appropriate for your architecture (`x86` vs. `x86_64`) and your Python version (`vs2008` for Python 2.7, `vs2010` for Python 3.3 and 3.4, `vs2015` for Python 3.5 and 3.6).

     Those builds include iconv support (via win-iconv statically linked).

     They provide both static and dynamic library versions of FreeTDS and versions built both with and without SSL support via OpenSSL (only dinamically linked).

     To install OpenSSL you'll need the distribution that can be downloaded from http://www.npcglib.org/~stathis/blog/precompiled-openssl/. Choose the right .7z file for your Python version (`vs2008` for Python 2.7, `vs2010` for Python 3.3 and 3.4, `vs2015` for Python 3.5 and 3.6).
   - Or you can build it yourself.

Changed in version 2.1.3: FreeTDS is linked statically again on our official Windows binaries.

pymssql version 2.1.2 included a change in the official Windows Wheels by which FreeTDS was dinamically linked. Read the relevant change log entry for the rationale behind that decision.

Given the fact this didn't have a good reception from our users, this change has been undone in 2.1.3, FreeTDS is statically linked like it happened until version 2.1.1.

5.2 Configuration

pymssql uses FreeTDS package to connect to SQL Server instances. You have to tell it how to find your database servers. The most basic info is host name, port number, and protocol version to use.

The system-wide FreeTDS configuration file is `/etc/freetds.conf` or `C:\freetds.conf`, depending upon your system. It is also possible to use a user specific configuration file, which is `$HOME/.freetds.conf` on Linux and `%APPDATA%\.freetds.conf` on Windows. Suggested contents to start with is at least:

```
[global]
    port = 1433
    tds version = 7.0
```

With this config you will be able to enter just the hostname to *pymssql.connect()* and `_mssql.connect()`:

```
import pymssql
connection = pymssql.connect(server='mydbserver', ...)
```

Otherwise you will have to enter the portname as in:

```
connection = pymssql.connect(server='mydbserver:1433', ...)
```

To connect to instance other than the default, you have to know either the instance name or port number on which the instance listens:

```
connection = pymssql.connect(server='mydbserver\\myinstancename', ...)
# or by port number (suppose you confirmed that this instance is on port 1237)
connection = pymssql.connect(server='mydbserver:1237', ...)
```

Please see also the *pymssql module reference*, *_mssql module reference*, and *FAQ* pages.

For more information on configuring FreeTDS please go to https://www.freetds.org/userguide/

5.2.1 Testing the connection

If you're sure that your server is reachable, but pymssql for some reason don't let you connect, you can check the connection with `tsql` utility which is part of FreeTDS package:

```
$ tsql
Usage:  tsql [-S <server> | -H <hostname> -p <port>] -U <username> [-P <password>] [-
↪I <config file>] [-o <options>] [-t delim] [-r delim] [-D database]
(...)
$ tsql -S mydbserver -U user
```

Note: Use the above form if and only if you specified server alias for mydbserver in freetds.conf. Otherwise use the host/port notation:

```
$ tsql -H mydbserver -p 1433 -U user
```

You'll be prompted for a password and if the connection succeeds, you'll see the SQL prompt:

```
1>
```

You can then enter queries and terminate the session with `exit`.

If the connection fails, `tsql` utility will display appropriate message.

CHAPTER 6

pymssql module reference

Complete documentation of pymssql module classes, methods and properties.

6.1 Module-level symbols

pymssql.**__version__**
pymssql version as an Unicode constant. E.g. u"2.1.1", u"2.2.0"

pymssql.**VERSION**
pymssql version in tuple form which is more easily handled (parse, compare) programmatically. E.g. (2, 1, 1), (2, 2, 0)

New in version 2.2.0.

pymssql.**__full_version__**
pymssql version as an Unicode constant but including any (**PEP 440**) suffixes. E.g. u"2.1.0.dev2", u"2.2.0.dev"

Constants, required by the DB-API 2.0 specification:

pymssql.**apilevel**
'2.0' – pymssql strives for compliance with DB-API 2.0.

pymssql.**paramstyle**
'pyformat' – pymssql uses extended python format codes.

pymssql.**threadsafety**
1 – Module may be shared, but not connections.

6.2 Functions

pymssql.**connect**(*server='.'*, *user=None*, *password=None*, *database=''*, *timeout=0*, *login_timeout=60*, *charset='UTF-8'*, *as_dict=False*, *host=''*, *appname=None*, *port='1433'*, *conn_properties=None*, *autocommit=False*, *tds_version=None*)

Constructor for creating a connection to the database. Returns a *Connection* object. Note that in most cases you will want to use keyword arguments, instead of positional arguments.

Parameters

- **server** (*str*) – database host
- **user** (*str*) – database user to connect as
- **password** (*str*) – user's password
- **database** (*str*) – The database to initialize the connection with. By default *SQL Server* selects the database which is set as default for specific user
- **timeout** (*int*) – query timeout in seconds, default `0` (no timeout)
- **login_timeout** (*int*) – timeout for connection and login in seconds, default `60`
- **charset** (*str*) – character set with which to connect to the database
- **as_dict** (*bool*) – Whether rows should be returned as dictionaries instead of tuples. You can access columns by 0-based index or by name. Please see *examples*
- **host** (*str*) – Database host and instance you want to connect to. Valid examples are:
  - `r'.\SQLEXPRESS'` – SQLEXPRESS instance on local machine (Windows only)
  - `r'(local)\SQLEXPRESS'` – same as above (Windows only)
  - `'SQLHOST'` – default instance at default port (Windows only)
  - `'SQLHOST'` – specific instance at specific port set up in freetds.conf (Linux/*nix only)
  - `'SQLHOST,1433'` – specified TCP port at specified host
  - `'SQLHOST:1433'` – the same as above
  - `'SQLHOST,5000'` – if you have set up an instance to listen on port 5000
  - `'SQLHOST:5000'` – the same as above

  `'.'` (the local host) is assumed if host is not provided.
- **appname** (*str*) – Set the application name to use for the connection
- **port** (*str*) – the TCP port to use to connect to the server
- **conn_properties** – SQL queries to send to the server upon connection establishment. Can be a string or another kind of iterable of strings. Default value: See *_mssql.connect()*
- **autocommit** (*bool*) – Whether to use default autocommiting mode or not
- **tds_version** (*str*) – TDS protocol version to use

Warning: Currently, setting *timeout* or *login_timeout* has a process-wide effect because the FreeTDS db-lib API functions used to implement such timeouts have a global effect.

Note: If you need to connect to Azure read the relevant *topic*.

New in version 2.1.1: The ability to connect to Azure.

New in version 2.1.1: The *conn_properties* parameter.

New in version 2.1.1: The *autocommit* parameter.

New in version 2.1.2: The *tds_version* parameter.

Changed in version 2.2.0: The default value of the *tds_version* parameter was changed to `None`. In version 2.1.2 its default value was `'7.1'`.

Warning: The *tds_version* parameter has a default value of `None`. This means two things:

1. You can't rely anymore in the old `'7.1'` default value and
2. Now you'll need to either

- Specify its value explicitly by passing a value for this parameter or
- Configure it using facilities provided by FreeTDS (see here and here)

This might look cumbersome but at the same time means you can now fully configure the characteristics of a connection to SQL Server from Python code when using pymssql without using a stanza for the server in the `freetds.conf` file or even with no `freetds.conf` at all. Up to version 2.1.1 it simply wasn't possible to control the TDS protocol version, and in version 2.1.2 it was possible to set it but version 7.1 was used if not specified.

Warning: FreeTDS added sopport for TDS protocol version 7.3 in version 0.95. You need to be careful of not asking for TDS 7.3 if you know the undelying FreeTDS used by pymssql is version 0.91 as it won't raise any error nor keep you from passing such an invalid value.

Warning: FreeTDS added support for TDS protocol version 7.3 in version 0.95. You need to be careful of not asking for TDS 7.3 if you know the undelying FreeTDS used by pymssql is older as it won't raise any error nor keep you from passing such an invalid value.

`pymssql.`**`get_dbversion`**`()`
Wrapper around DB-Library's `dbversion()` function which returns the version of FreeTDS (actually the version of DB-Lib) in string form. E.g. `"freetds v1.2.5"`.

A pymssql extension to the DB-API 2.0.

`pymssql.`**`set_max_connections`**`(`*number*`)`
Sets maximum number of simultaneous database connections allowed to be open at any given time. Default is 25.

A pymssql extension to the DB-API 2.0.

`pymssql.`**`get_max_connections`**`()`
Gets current maximum number of simultaneous database connections allowed to be open at any given time.

A pymssql extension to the DB-API 2.0.

pymssql.**set_wait_callback**(*wait_callback_callable*)
New in version 2.1.0.

Allows pymssql to be used along cooperative multi-tasking systems and have it call a callback when it's waiting for a response from the server.

The passed callback callable should receive one argument: The file descriptor/handle of the network socket connected to the server, so its signature must be:

```
def wait_callback_callable(read_fileno):
    #...
    pass
```

Its body should invoke the appropiate API of the multi-tasking framework you are using use that results in the current greenlet yielding the CPU to its siblings whilst there isn't incoming data in the socket.

See the *pymssql examples document* for a more concrete example.

A pymssql extension to the DB-API 2.0.

pymssql.**version_info**()
New in version 2.2.0.

Returns string with version information about pymssql, FreeTDS, Python and OS. Please include the output of this function when reporting bus etc.:

```
/<path to your python>/python -c "import pymssql; print(pymssql.version_info())"
```

A pymssql extension to the DB-API 2.0.

6.3 Connection class

class pymssql.**Connection**(*user*, *password*, *host*, *database*, *timeout*, *login_timeout*, *charset*, *as_dict*)
This class represents an MS SQL database connection. You can create an instance of this class by calling constructor *pymssql.connect()*.

6.3.1 Connection object properties

This class has no useful properties and data members.

6.3.2 Connection object methods

Connection.**autocommit**(*status*)
Where *status* is a boolean value. This method turns autocommit mode on or off.

By default, autocommit mode is off, what means every transaction must be explicitly committed if changed data is to be persisted in the database.

You can turn autocommit mode on, what means every single operation commits itself as soon as it succeeds.

A pymssql extension to the DB-API 2.0.

Connection.**close**()
Close the connection.

Connection.**cursor**()
Return a cursor object, that can be used to make queries and fetch results from the database.

Connection.**commit**()
Commit current transaction. You must call this method to persist your data if you leave autocommit at its default value, which is `False`.

See also *pymssql examples*.

Connection.**rollback**()
Roll back current transaction.

Connection.**bulk_copy**(*self*, *table_name*, *elements*, *column_ids=None*, *batch_size=1000*, *tablock=False*, *check_constraints=False*, *fire_triggers=False*)
New in version 2.2.0.

Insert data into the target table using the Bulk Copy protocol.

Parameters

- **table_name** (*str*) – The name of the target table.
- **elements** (*List[Tuple]*) – The data to insert.
- **column_ids** (*List[int]*) – The IDs of the target columns. The first column in a table is index 1. If unset will default to n, where n is the number of elements in each tuple passed as data.
- **batch_size** (*int*) – Commit rows to the target table for every batch_size rows, defaults to 1_000.
- **tablock** (*bool*) – Set TABLOCK hint.
- **check_constraints** (*bool*) – Set CHECK_CONSTRAINTS hint.
- **fire_triggers** (*bool*) – Set FIRE_TRIGGERS hint.

6.4 Cursor class

class pymssql.**Cursor**

This class represents a Cursor (in terms of Python DB-API specs) that is used to make queries against the database and obtaining results. You create `Cursor` instances by calling *cursor()* method on an open *Connection* connection object.

6.4.1 Cursor object properties

Cursor.**rowcount**
Returns number of rows affected by last operation. In case of `SELECT` statements it returns meaningful information only after all rows have been fetched.

Cursor.**connection**
This is the extension of the DB-API specification. Returns a reference to the connection object on which the cursor was created.

Cursor.**lastrowid**
This is the extension of the DB-API specification. Returns identity value of last inserted row. If previous operation did not involve inserting a row into a table with identity column, `None` is returned.

Cursor.**rownumber**
This is the extension of the DB-API specification. Returns current 0-based index of the cursor in the result set.

6.4.2 Cursor object methods

Cursor.**close**()
Close the cursor. The cursor is unusable from this point.

Cursor.**execute**(*operation*)
Cursor.**execute**(*operation*, *params*)
operation is a string and *params*, if specified, is a simple value, a tuple, a dict, or None.

Performs the operation against the database, possibly replacing parameter placeholders with provided values. This should be preferred method of creating SQL commands, instead of concatenating strings manually, what makes a potential of SQL Injection attacks. This method accepts formatting similar to Python's builtin string interpolation operator. However, since formatting and type conversion is handled internally, only the %s and %d placeholders are supported. Both placeholders are functionally equivalent.

Keyed placeholders are supported if you provide a dict for *params*.

If you call execute() with one argument, the % sign loses its special meaning, so you can use it as usual in your query string, for example in LIKE operator. See the *examples*.

You must call *Connection.commit()* after execute() or your data will not be persisted in the database. You can also set connection.autocommit if you want it to be done automatically. This behaviour is required by DB-API, if you don't like it, just use the *_mssql* module instead.

Cursor.**executemany**(*operation*, *params_seq*)
operation is a string and *params_seq* is a sequence of tuples (e.g. a list). Execute a database operation repeatedly for each element in parameter sequence.

Cursor.**fetchone**()
Fetch the next row of a query result, returning a tuple, or a dictionary if as_dict was passed to pymssql.connect(), or None if no more data is available. Raises OperationalError (**PEP 249#operationalerror**) if previous call to execute*() did not produce any result set or no call was issued yet.

Cursor.**fetchmany**(*size=None*)
Fetch the next batch of rows of a query result, returning a list of tuples, or a list of dictionaries if *as_dict* was passed to *pymssql.connect()*, or an empty list if no more data is available. You can adjust the batch size using the *size* parameter, which is preserved across many calls to this method. Raises OperationalError (**PEP 249#operationalerror**) if previous call to execute*() did not produce any result set or no call was issued yet.

Cursor.**fetchall**()
Fetch all remaining rows of a query result, returning a list of tuples, or a list of dictionaries if as_dict was passed to pymssql.connect(), or an empty list if no more data is available. Raises OperationalError (**PEP 249#operationalerror**) if previous call to execute*() did not produce any result set or no call was issued yet.

Cursor.**nextset**()
This method makes the cursor skip to the next available result set, discarding any remaining rows from the current set. Returns True value if next result is available, None if not.

Cursor.**__iter__**()
Cursor.**next**()
These methods facilitate Python iterator protocol. You most likely will not call them directly, but indirectly by using iterators.

A pymssql extension to the DB-API 2.0.

Cursor.**setinputsizes**()
Cursor.**setoutputsize**()
These methods do nothing, as permitted by DB-API specs.

6.5 Exceptions

exception pymssql.**StandardError**
Root of the exception hierarchy.

exception pymssql.**Warning**
Raised for important warnings like data truncations while inserting, etc. A subclass of *StandardError*.

exception pymssql.**Error**
Base class of all other error exceptions. You can use this to catch all errors with one single except statement. A subclass of *StandardError*.

exception pymssql.**InterfaceError**
Raised for errors that are related to the database interface rather than the database itself. A subclass of *Error*.

exception pymssql.**DatabaseError**
Raised for errors that are related to the database. A subclass of *Error*.

exception pymssql.**DataError**
Raised for errors that are due to problems with the processed data like division by zero, numeric value out of range, etc. A subclass of *DatabaseError*.

exception pymssql.**OperationalError**
Raised for errors that are related to the database's operation and not necessarily under the control of the programmer, e.g. an unexpected disconnect occurs, the data source name is not found, a transaction could not be processed, a memory allocation error occurred during processing, etc. A subclass of *DatabaseError*.

exception pymssql.**IntegrityError**
Raised when the relational integrity of the database is affected, e.g. a foreign key check fails. A subclass of *DatabaseError*.

exception pymssql.**InternalError**
Raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the transaction is out of sync, etc. A subclass of *DatabaseError*.

exception pymssql.**ProgrammingError**
Raised for programming errors, e.g. table not found or already exists, syntax error in the SQL statement, wrong number of parameters specified, etc. A subclass of *DatabaseError*.

exception pymssql.**NotSupportedError**
Raised in case a method or database API was used which is not supported by the database, e.g. requesting a *rollback()* on a connection that does not support transaction or has transactions turned off. A subclass of *DatabaseError*.

exception pymssql.**ColumnsWithoutNamesError**
Raised by *Cursor.execute()* when `as_dict=True` has been specified to *open* the *connection* and the query sent to the server doesn't involve columns names in its results. A subclass of *InterfaceError*.

Note: `ColumnsWithoutNamesError` isn't a PEP-249-mandated exception but rather a pymssql extension.

CHAPTER 7

_mssql module reference

Complete documentation of _mssql module classes, methods and properties.

7.1 Module-level symbols

_mssql.**__version__**
See *pymssql.__version__*.

_mssql.**VERSION**
See *pymssql.VERSION*.

New in version 2.2.0.

_mssql.**__full_version__**
See *pymssql.__full_version__*.

Variables whose values you can change to alter behavior on a global basis:

_mssql.**login_timeout**
Timeout for connection and login in seconds, default 60.

_mssql.**min_error_severity**
Minimum severity of errors at which to begin raising exceptions. The default value of 6 should be appropriate in most cases.

7.2 Functions

_mssql.**get_dbversion**()

Wrapper around DB-Library's dbversion() function which returns the version of FreeTDS (actually the version of DB-Lib) in string form. E.g. "freetds v1.2.5".

New in version 2.2.0.

_mssql.**set_max_connections**(*number*)
Sets maximum number of simultaneous connections allowed to be open at any given time. Default is 25.

_mssql.**get_max_connections**()
Gets current maximum number of simultaneous connections allowed to be open at any given time.

7.3 MSSQLConnection class

class _mssql.**MSSQLConnection**
This class represents an MS SQL database connection. You can make queries and obtain results through a database connection.

You can create an instance of this class by calling `_mssql.connect()`. It accepts the following arguments. Note that you can use keyword arguments, instead of positional arguments.

Parameters

- **server** (*str*) – Database server and instance you want to connect to. Valid examples are:
  - `r'.\SQLEXPRESS'` – SQLEXPRESS instance on local machine (Windows only)
  - `r'(local)\SQLEXPRESS'` – Same as above (Windows only)
  - `'SQLHOST'` – Default instance at default port (Windows only)
  - `'SQLHOST'` – Specific instance at specific port set up in freetds.conf (Linux/*nix only)
  - `'SQLHOST,1433'` – Specified TCP port at specified host
  - `'SQLHOST:1433'` – The same as above
  - `'SQLHOST,5000'` – If you have set up an instance to listen on port 5000
  - `'SQLHOST:5000'` – The same as above
- **user** (*str*) – Database user to connect as
- **password** (*str*) – User's password
- **charset** (*str*) – Character set name to set for the connection.
- **database** (*str*) – The database you want to initially to connect to; by default, *SQL Server* selects the database which is set as the default for the specific user
- **appname** (*str*) – Set the application name to use for the connection
- **port** (*str*) – the TCP port to use to connect to the server
- **tds_version** (*str*) – TDS protocol version to ask for. Default value: `None`
- **conn_properties** – SQL queries to send to the server upon connection establishment. Can be a string or another kind of iterable of strings. Default value:

```
SET ARITHABORT ON;
SET CONCAT_NULL_YIELDS_NULL ON;
SET ANSI_NULLS ON;
SET ANSI_NULL_DFLT_ON ON;
SET ANSI_PADDING ON;
SET ANSI_WARNINGS ON;
SET ANSI_NULL_DFLT_ON ON;
SET CURSOR_CLOSE_ON_COMMIT ON;
```

(continues on next page)

(continued from previous page)

```
SET QUOTED_IDENTIFIER ON;
SET TEXTSIZE 2147483647; -- http://msdn.microsoft.com/en-us/library/aa259190
↪%28v=sql.80%29.aspx
```

New in version 2.1.1: The *conn_properties* parameter.

Changed in version 2.1.1: Before 2.1.1, the initialization queries now specified by *conn_properties* wasn't customizable and its value was hard-coded to the literal shown above.

Note: If you need to connect to Azure read the relevant *topic*.

New in version 2.1.1: The ability to connect to Azure.

Changed in version 2.2.0: The default value of the *tds_version* parameter was changed to `None`. Between versions 2.0.0 and 2.1.2 its default value was `'7.1'`.

Warning: The *tds_version* parameter has a default value of `None`. This means two things:

1. You can't rely anymore in the old `'7.1'` default value and
2. Now you'll need to either

- Specify its value explicitly by passing a value for this parameter or
- Configure it using facilities provided by FreeTDS (see here and here)

This might look cumbersome but at the same time means you can now fully configure the characteristics of a connection to SQL Server when using pymssql/_mssql without using a stanza for the server in the `freetds.conf` file or even with no `freetds.conf` at all. Starting with pymssql version 2.0.0 and up to version 2.1.2 it was already possible to set the TDS protocol version to ask for when connecting to the server but version 7.1 was used if not specified.

Warning: FreeTDS added sopport for TDS protocol version 7.3 in version 0.95. You need to be careful of not asking for TDS 7.3 if you know the undelying FreeTDS used by pymssql is version 0.91 as it won't raise any error nor keep you from passing such an invalid value.

Warning: FreeTDS added support for TDS protocol version 7.3 in version 0.95. You need to be careful of not asking for TDS 7.3 if you know the undelying FreeTDS used by pymssql is older as it won't raise any error nor keep you from passing such an invalid value.

7.3.1 `MSSQLConnection` object properties

`MSSQLConnection.`**`connected`**
`True` if the connection object has an open connection to a database, `False` otherwise.

`MSSQLConnection.`**`charset`**
Character set name that was passed to _mssql.connect().

MSSQLConnection.**identity**
Returns identity value of last inserted row. If previous operation did not involve inserting a row into a table with identity column, None is returned. Example usage – assume that persons table contains an identity column in addition to name column:

```
conn.execute_non_query("INSERT INTO persons (name) VALUES('John Doe')")
print "Last inserted row has id = " + conn.identity
```

MSSQLConnection.**query_timeout**
Query timeout in seconds, default is 0, which means to wait indefinitely for results. Due to the way DB-Library for C works, setting this property affects all connections opened from the current Python script (or, very technically, all connections made from this instance of dbinit()).

MSSQLConnection.**rows_affected**
Number of rows affected by last query. For SELECT statements this value is only meaningful after reading all rows.

MSSQLConnection.**debug_queries**
If set to true, all queries are printed to stderr after formatting and quoting, just before being sent to *SQL Server*. It may be helpful if you suspect problems with formatting or quoting.

MSSQLConnection.**tds_version**
The TDS version used by this connection. Can be one of `4.2`, `5.0` `7.0`, `7.1`, `7.2`, `7.3` or `None` if no TDS version could be detected.

Changed in version 2.1.4: For correctness and consistency the value used to indicate TDS 7.1 changed from `8.0` to `7.1` on pymssql 2.1.4.

Changed in version 2.1.3: `7.3` was added as a possible value.

MSSQLConnection.**tds_version_tuple**
New in version 2.2.0.

The TDS version used by this connection in tuple form which is more easily handled (parse, compare) programmatically. Can be one of `(4, 2)`, `(5, 0)`, `(7, 0)`, `(7, 1)`, `(7, 2)`, `(7, 3)` or `None` if no TDS version could be detected.

Changed in version 2.1.3: `7.3` was added as a possible value.

7.3.2 `MSSQLConnection` object methods

MSSQLConnection.**cancel**()
Cancel all pending results from the last SQL operation. It can be called more than one time in a row. No exception is raised in this case.

MSSQLConnection.**close**()
Close the connection and free all memory used. It can be called more than one time in a row. No exception is raised in this case.

MSSQLConnection.**execute_query**(*query_string*)
MSSQLConnection.**execute_query**(*query_string*, *params*)
This method sends a query to the *MS SQL Server* to which this object instance is connected. An exception is raised on failure. If there are pending results or rows prior to executing this command, they are silently discarded.

After calling this method you may iterate over the connection object to get rows returned by the query.

You can use Python formatting and all values get properly quoted. Please see examples for details.

This method is intented to be used on queries that return results, i.e. `SELECT`.

MSSQLConnection.**execute_non_query**(*query_string*)
MSSQLConnection.**execute_non_query**(*query_string*, *params*)
This method sends a query to the *MS SQL Server* to which this object instance is connected. After completion, its results (if any) are discarded. An exception is raised on failure. If there are pending results or rows prior to executing this command, they are silently discarded.

You can use Python formatting and all values get properly quoted. Please see examples for details.

This method is useful for `INSERT`, `UPDATE`, `DELETE`, and for Data Definition Language commands, i.e. when you need to alter your database schema.

MSSQLConnection.**execute_scalar**(*query_string*)
MSSQLConnection.**execute_scalar**(*query_string*, *params*)
This method sends a query to the *MS SQL Server* to which this object instance is connected, then returns first column of first row from result. An exception is raised on failure. If there are pending results or rows prior to executing this command, they are silently discarded.

You can use Python formatting and all values get properly quoted. Please see examples for details.

This method is useful if you want just a single value from a query, as in the example below. This method works in the same way as `iter(conn).next()[0]`. Remaining rows, if any, can still be iterated after calling this method.

Example usage:

```
count = conn.execute_scalar("SELECT COUNT(*) FROM employees")
```

MSSQLConnection.**execute_row**(*query_string*)
MSSQLConnection.**execute_row**(*query_string*, *params*)
This method sends a query to the *MS SQL Server* to which this object instance is connected, then returns first row of data from result. An exception is raised on failure. If there are pending results or rows prior to executing this command, they are silently discarded.

You can use Python formatting and all values get properly quoted. Please see examples for details.

This method is useful if you want just a single row and don't want or don't need to iterate over the connection object. This method works in the same way as `iter(conn).next()` to obtain single row. Remaining rows, if any, can still be iterated after calling this method.

Example usage:

```
empinfo = conn.execute_row("SELECT * FROM employees WHERE empid=10")
```

MSSQLConnection.**get_header**()
This method is infrastructure and doesn't need to be called by your code. It gets the Python DB-API compliant header information. Returns a list of 7-element tuples describing current result header. Only name and DB-API compliant type is filled, rest of the data is `None`, as permitted by the specs.

MSSQLConnection.**init_procedure**(*name*)
Create an MSSQLStoredProcedure object that will be used to invoke thestored procedure with the given name.

MSSQLConnection.**nextresult**()
Move to the next result, skipping all pending rows. This method fetches and discards any rows remaining from current operation, then it advances to next result (if any). Returns `True` value if next set is available, `None` otherwise. An exception is raised on failure.

MSSQLConnection.**select_db**(*dbname*)
This function makes the given database the current one. An exception is raised on failure.

MSSQLConnection.**__iter__**()

MSSQLConnection.**next**()
New in version 2.1.0.

These methods implement the Python iterator protocol. You most likely will not call them directly, but indirectly by using iterators.

MSSQLConnection.**set_msghandler**(*handler*)
New in version 2.1.1.

This method allows setting a message handler function for the connection to allow a client to gain access to the messages returned from the server.

The signature of the message handler function *handler* passed to this method must be:

```
def my_msg_handler(msgstate, severity, srvname, procname, line, msgtext):
    # The body of the message handler.
```

msgstate, *severity* and *line* will be integers, *srvname*, *procname* and *msgtext* will be strings.

7.4 MSSQLStoredProcedure class

class _mssql.**MSSQLStoredProcedure**
This class represents a stored procedure. You create an object of this class by calling the *init_procedure()* method on *MSSQLConnection* object.

7.4.1 MSSQLStoredProcedure object properties

MSSQLStoredProcedure.**connection**
An underlying MSSQLConnection object.

MSSQLStoredProcedure.**name**
The name of the procedure that this object represents.

MSSQLStoredProcedure.**parameters**
The parameters that have been bound to this procedure.

7.4.2 MSSQLStoredProcedure object methods

MSSQLStoredProcedure.**bind**(*value*, *dbtype*, *name=None*, *output=False*, *null=False*, *max_length=-1*)
This method binds a parameter to the stored procedure. *value* and *dbtype* are mandatory arguments, the rest is optional.

Parameters

- **value** – Is the value to store in the parameter.
- **dbtype** – Is one of: SQLBINARY, SQLBIT, SQLBITN, SQLCHAR, SQLDATETIME, SQLDATETIM4, SQLDATETIMN, SQLDECIMAL, SQLFLT4, SQLFLT8, SQLFLTN, SQLIMAGE, SQLINT1, SQLINT2, SQLINT4, SQLINT8, SQLINTN, SQLMONEY, SQLMONEY4, SQLMONEYN, SQLNUMERIC, SQLREAL, SQLTEXT, SQLVARBINARY, SQLVARCHAR, SQLUUID.
- **name** – Is the name of the parameter. Needs to be in "@name" form.
- **output** – Is the direction of the parameter. True indicates that it is an output parameter i.e. it returns a value after procedure execution (in SQL DDL they are declared by using the "output" suffix, e.g. "@aname varchar(10) output").

- **null** – Boolean. Signals than NULL must be the value to be bound to the argument of this input parameter.
- **max_length** – Is the maximum data length for this parameter to be returned from the stored procedure.

MSSQLStoredProcedure.**execute**()
Execute the stored procedure.

7.5 Module-level exceptions

Exception hierarchy:

```
MSSQLException
|
+-- MSSQLDriverException
|
+-- MSSQLDatabaseException
```

exception _mssql.**MSSQLDriverException**
MSSQLDriverException is raised whenever there is a problem within _mssql – e.g. insufficient memory for data structures, and so on.

exception _mssql.**MSSQLDatabaseException**

MSSQLDatabaseException is raised whenever there is a problem with the database – e.g. query syntax error, invalid object name and so on. In this case you can use the following properties to access details of the error:

number
The error code, as returned by *SQL Server*.

severity
The so-called severity level, as returned by *SQL Server*. If value of this property is less than the value of *_mssql.min_error_severity*, such errors are ignored and exceptions are not raised.

state
The third error code, as returned by *SQL Server*.

message
The error message, as returned by *SQL Server*.

You can find an example of how to use this data at the bottom of *_mssql examples page*.

CHAPTER 8

Migrating from 1.x to 2.x

Because of the DB-API standard and because effort was made to make the interface of pymssql 2.x similar to that of pymssql 1.x, there are only a few differences and usually upgrading is pretty easy.

There are a few differences though...

8.1 `str` vs. `unicode`

Note that we are talking about Python 2, because pymssql 1.x doesn't work on Python 3.

pymssql 1.x will return `str` instances:

```
>>> pymssql.__version__
'1.0.3'
>>> conn.as_dict = True
>>> cursor = conn.cursor()
>>> cursor.execute("SELECT 'hello' AS str FROM foo")
>>> cursor.fetchall()
[{0: 'hello', 'str': 'hello'}]
```

whereas pymssql 2.x will return `unicode` instances:

```
>>> pymssql.__version__
u'2.0.1.2'
>>> conn.as_dict = True
>>> cursor = conn.cursor()
>>> cursor.execute("SELECT 'hello' AS str FROM foo")
>>> cursor.fetchall()
[{u'str': u'hello'}]
```

If your application has code that deals with `str` and `unicode` differently, then you may run into issues.

You can always convert a `unicode` to a `str` by encoding:

```
>>> cursor.execute("SELECT 'hello' AS str FROM foo")
>>> s = cursor.fetchone()['str']
>>> s
u'hello'
>>> s.encode('utf-8')
'hello'
```

8.2 Handling of `uniqueidentifier` columns

SQL Server has a data type called uniqueidentifier.

In pymssql 1.x, `uniqueidentifier` columns are returned in results as byte strings with 16 bytes; if you want a `uuid.UUID` instance, then you have to construct it yourself from the byte string:

```
>>> cursor.execute("SELECT * FROM foo")
>>> id_value = cursor.fetchone()['uniqueidentifier']
>>> id_value
'j!\xcf\x14D\xce\xe6B\xab\xe0\xd9\xbey\x0cMK'
>>> type(id_value)
<type 'str'>
>>> len(id_value)
16
>>> import uuid
>>> id_uuid = uuid.UUID(bytes_le=id_value)
>>> id_uuid
UUID('14cf216a-ce44-42e6-abe0-d9be790c4d4b')
```

In pymssql 2.x, `uniqueidentifier` columns are returned in results as instances of `uuid.UUID` and if you want the bytes, like in pymssql 1.x, you have to use `uuid.UUID.bytes_le` to get them:

```
>>> cursor.execute("SELECT * FROM foo")
>>> id_value = cursor.fetchone()['uniqueidentifier']
>>> id_value
UUID('14cf216a-ce44-42e6-abe0-d9be790c4d4b')
>>> type(id_value)
<class 'uuid.UUID'>
>>> id_value.bytes_le
'j!\xcf\x14D\xce\xe6B\xab\xe0\xd9\xbey\x0cMK'
```

8.3 Arguments to `pymssql.connect`

The arguments are a little bit different. Some notable differences:

In pymssql 1.x, the parameter to specify the host is called `host` and it can contain a host and port – e.g.:

```
conn = pymssql.connect(host='SQLHOST:1433')  # specified TCP port at a host
```

There are some other syntaxes for the `host` parameter that allow using a comma instead of a colon to delimit host and port, to specify Windows hosts, to specify a specific SQL Server instance, etc.

```
conn = pymssql.connect(host=r'SQLHOST,5000')  # specified TCP port at a host
conn = pymssql.connect(host=r'(local)\SQLEXPRESS')  # named instance on local machine␣
↪[Win]
```

In pymssql 2.x, the `host` parameter is supported (I am unsure if it has all of the functionality of pymssql 1.x). There is also a parameter to specify the host that is called `server`. There is a separate parameter called `port`.

```
conn = pymssql.connect(server='SQLHOST', port=1500)
```

8.4 Parameter substitution

For parameter substitution, pymssql 2.x supports the `format` and `pyformat` PEP 249 paramstyles.

Note that for `pyformat`, PEP 249 only shows the example of a string substitution – e.g.:

```
%(name)s
```

It is not clear from PEP 249 whether other types should be supported, like:

```
%(name)d
%(name)f
```

However, in this mailing list thread, the general consensus is that the string format should be the only one required.

Note that pymssql 2.x does not support `%(name)d`, whereas pymssql 1.x did. So you may have to change code that uses this notation:

```
>>> pymssql.__version__
u'2.0.1.2'
>>> pymssql.paramstyle
'pyformat'

>>> cursor.execute("select 'hello' where 1 = %(name)d", dict(name=1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pymssql.pyx", line 430, in pymssql.Cursor.execute (pymssql.c:5900)
    if not self._source._conn.nextresult():
pymssql.ProgrammingError: (102, "Incorrect syntax near '('.
DB-Lib error message 20018, severity 15:\n
General SQL Server error: Check messages from the SQL Server\n")
```

to:

```
>>> cursor.execute("select 'hello' where '1' = %(name)s", dict(name='1'))
>>> cursor.fetchall()
[(u'hello',)]
```

or:

```
>>> cursor.execute("select 'hello' where 1 = %d", 1)
>>> cursor.fetchall()
[(u'hello',)]
```

Examples of this problem:

- Google Group post: paramstyle changed?
- GitHub issue #155: pymssql 2.x does not support "%(foo)d" parameter substitution style; pymssql 1.x did

CHAPTER 9

Frequently asked questions

9.1 Cannot connect to SQL Server

If your Python program/script can't connect to a *SQL Server* instance, try the following:

- By default *SQL Server* 2005 and newer don't accept remote connections, you have to use *SQL Server Surface Area Configuration* and/or *SQL Server Configuration Manager* to enable specific protocols and network adapters; don't forget to restart *SQL Server* after making these changes,
- If *SQL Server* is on a remote machine, check whether connections are not blocked by any intermediate firewall device, firewall software, antivirus software, or other security facility,
- Check that you can connect with another tool.

  If you are using FreeTDS, then you can use the included `tsql` command to try to connect – it looks like this:

```
$ tsql -H sqlserverhost -p 1433 -U user -P password -D tempdb
locale is "en_US.UTF-8"
locale charset is "UTF-8"
using default charset "UTF-8"
Setting tempdb as default database in login packet
1> SELECT @@VERSION
2> GO

Microsoft SQL Server 2012 - 11.0.2100.60 (X64)
        Feb 10 2012 19:39:15
        Copyright (c) Microsoft Corporation
        Developer Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601:␣
↪Service Pack 1)

(1 row affected)
```

Note: Note that I use the `-H` option rather than the `-S` option to `tsql`. This is because with `-H`, it will bypass reading settings from the `freetds.conf` file like `port` and `tds version`, and so

this is more similar to what happens with pymssql.

If you **can't** connect with `tsql` or other tools, then the problem is probably not pymssql; you probably have a problem with your server configuration (see below), *FreeTDS Configuration*, network, etc.

If you **can** connect with `tsql`, then you should be able to connect with pymssql with something like this:

```
>>> import pymssql
>>> conn = pymssql.connect(
...     server="sqlserverhost",
...     port=1433,
...     user="user",
...     password="password",
...     database="tempdb")
>>> conn
<pymssql.Connection object at 0x10107a3f8>
>>> cursor = conn.cursor()
>>> cursor.execute("SELECT @@VERSION")
>>> print(cursor.fetchone()[0])
Microsoft SQL Server 2012 - 11.0.2100.60 (X64)
  Feb 10 2012 19:39:15
  Copyright (c) Microsoft Corporation
  Developer Edition (64-bit) on Windows NT 6.1 <X64> (Build 7601: Service␣
↪Pack 1)
```

If something like the above doesn't work, then you can try to diagnose by setting one or both of the following FreeTDS environment variables that control logging:

- `TDSDUMP`
- `TDSDUMPCONFIG`

Either or both of these can be set. They can be set to a filename or to `stdout` or `stderr`.

These will cause FreeTDS to output a ton of information about what it's doing and you may very well spot that it's not using the port that you expected or something similar. For example:

```
>>> import os
>>> os.environ['TDSDUMP'] = 'stdout'
>>>
>>> import pymssql
>>> conn = pymssql.connect(server="sqlserverhost")
log.c:194:Starting log file for FreeTDS 0.92.dev.20140102
  on 2014-01-09 14:05:32 with debug flags 0x4fff.
config.c:731:Setting 'dump_file' to 'stdout' from $TDSDUMP.
...
dblib.c:7934:20013: "Unknown host machine name"
dblib.c:7955:"Unknown host machine name", client returns 2 (INT_CANCEL)
util.c:347:tdserror: client library returned TDS_INT_CANCEL(2)
util.c:370:tdserror: returning TDS_INT_CANCEL(2)
login.c:418:IP address pointer is empty
login.c:420:Server sqlserverhost:1433 not found!
...
```

Note: Note that pymssql will use a default port of 1433, despite any ports you may have specified in your `freetds.conf` file. So if you have SQL Server running on a port other than 1433, you must

explicitly specify the `port` in your call to `pymssql.connect`. You cannot rely on it to pick up the port in your `freetds.conf`, even though `tsql -S` might do this. This is why I recommend using `tsql -H` instead for diagnosing connection problems.

It is also useful to know that `tsql -C` will output a lot of information about FreeTDS, that can be useful for diagnosing problems:

```
$ tsql -C
Compile-time settings (established with the "configure" script)
                             Version: freetds v0.92.dev.20140102
             freetds.conf directory: /usr/local/etc
     MS db-lib source compatibility: no
        Sybase binary compatibility: no
                      Thread safety: yes
                      iconv library: yes
                        TDS version: 5.0
                              iODBC: yes
                           unixodbc: no
              SSPI "trusted" logins: no
                           Kerberos: no
                            OpenSSL: no
                             GnuTLS: no
```

- If you use pymssql on Linux/Unix with FreeTDS, check that FreeTDS's configuration is ok and that it can be found by pymssql. The easiest way is to test connection using `tsql` utility which can be found in FreeTDS package. See *FreeTDS Configuration* for more info,

9.2 Returned dates are not correct

If you use pymssql on Linux/*nix and you suspect that returned dates are not correct, please read the *FreeTDS and dates* page.

9.3 Queries return no rows

There is a known issue where some versions of pymssql 1.x (pymssql 1.0.2 is where I've seen this) work well with FreeTDS 0.82, but return no rows when used with newer versions of FreeTDS, such as FreeTDS 0.91. At SurveyMonkey, we ran into this problem when we were using pymssql 1.0.2 and then upgraded servers from Ubuntu 10 (which includes FreeTDS 0.82) to Ubuntu 12 (which includes FreeTDS 0.91).

E.g.:

```
>>> import pymssql
>>> pymssql.__version__
'1.0.2'
>>> conn = pymssql.connect(host='127.0.0.1:1433', user=user,
...                        password=password, database='tempdb')
>>> cursor = conn.cursor()
>>> cursor.execute('SELECT 1')
>>> cursor.fetchall()
[]
```

See GitHub issue 137: pymssql 1.0.2: No result rows are returned from queries with newer versions of FreeTDS.

There are two way to fix this problem:

1. (Preferred) Upgrade to pymssql 2.x. pymssql 1.x is not actively being worked on. pymssql 2.x is rewritten in Cython, is actively maintained, and offers better performance, Python 3 support, etc. E.g.:

```
>>> import pymssql
>>> pymssql.__version__
u'2.0.1.2'
>>> conn = pymssql.connect(host='127.0.0.1:1433', user=user,
...                        password=password, database='tempdb')
>>> cursor = conn.cursor()
>>> cursor.execute('SELECT 1')
>>> cursor.fetchall()
[(1,)]
```

2. Upgrade to pymssql 1.0.3. This is identical to pymssql 1.0.2 except that it has a very small change that makes it so that it works with newer versions of FreeTDS as well as older versions.

   E.g.:

```
>>> import pymssql
>>> pymssql.__version__
'1.0.3'
>>> conn = pymssql.connect(host='127.0.0.1:1433', user=user,
...                        password=password, database='tempdb')
>>> cursor = conn.cursor()
>>> cursor.execute('SELECT 1')
>>> cursor.fetchall()
[(1,)]
```

9.4 Results are missing columns

One possible cause of your result rows missing columns is if you are using a connection or cursor with `as_dict=True` and your query has columns without names – for example:

```
>>> cursor = conn.cursor(as_dict=True)
>>> cursor.execute("SELECT MAX(x) FROM (VALUES (1), (2), (3)) AS foo(x)")
>>> cursor.fetchall()
[{}]
```

Whoa, what happened to `MAX(x)` ?!?!

In this case, pymssql does not know what name to use for the dict key, so it omits the column.

The solution is to supply a name for all columns – e.g.:

```
>>> cursor.execute("SELECT MAX(x) AS [MAX(x)] FROM (VALUES (1), (2), (3)) AS foo(x)")
>>> cursor.fetchall()
[{u'MAX(x)': 3}]
```

This behavior was changed in https://github.com/pymssql/pymssql/pull/160 – with this change, if you specify `as_dict=True` and omit column names, an exception will be raised:

```
>>> cursor.execute("SELECT MAX(x) FROM (VALUES (1), (2), (3)) AS foo(x)")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pymssql.pyx", line 426, in pymssql.Cursor.execute (pymssql.c:5828)
```

(continues on next page)

(continued from previous page)

```
    raise ColumnsWithoutNamesError(columns_without_names)
pymssql.ColumnsWithoutNamesError: Specified as_dict=True and there are columns with␣
↪no names: [0]
```

Examples of this problem:

- Google Group post: pymssql with MAX(values) function does not appear to work

9.5 pymssql does not unserialize `DATE` and `TIME` columns to `datetime.date` and `datetime.time` instances

You may notice that pymssql will unserialize a `DATETIME` column to a `datetime.datetime` instance, but it will unserialize `DATE` and `TIME` columns as simple strings. For example:

```
>>> cursor.execute("""
... CREATE TABLE dates_and_times (
...     datetime DATETIME,
...     date DATE,
...     time TIME,
... )
... """)
>>> cursor.execute("INSERT INTO dates_and_times VALUES (GETDATE(), '20140109', '6:17')
↪")
>>> cursor.execute("SELECT * FROM dates_and_times")
>>> cursor.fetchall()
[{u'date': u'2014-01-09', u'time': u'06:17:00.0000000',
  u'datetime': datetime.datetime(2014, 1, 9, 12, 41, 59, 403000)}]
>>> cursor.execute("DROP TABLE dates_and_times")
```

Yep, so the problem here is that `DATETIME` has been supported by FreeTDS for a long time, but `DATE` and `TIME` are newer types in SQL Server, Microsoft never added support for them to db-lib and FreeTDS added support for them in version 0.95.

If you need support for these data types (i.e. they get returned from the database as their native corresponding Python data types instead of as strings) as well as for the `DATETIME2` one, then make sure the following conditions are met:

- You are connecting to SQL Server 2008 or newer.
- You are using FreeTDS 0.95 or newer.
- You are using TDS protocol version 7.3 or newer.

9.6 Shared object "libsybdb.so.3" not found

On Linux/*nix you may encounter the following behaviour:

```
>>> from pymssql import _mssql
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ImportError: Shared object "libsybdb.so.3" not found
```

It may mean that the FreeTDS library is unavailable, or that the dynamic linker is unable to find it. Check that it is installed and that the path to `libsybdb.so` is in `/etc/ld.so.conf` file. Then do `ldconfig` as root to refresh

linker database. On Solaris, I just set the `LD_LIBRARY_PATH` environment variable to the directory with the library just before launching Python.

pymssql 2.x bundles the FreeTDS `sybdb` library for supported platforms. This error may show up in 2.x versions if you are trying to build with your own FreeTDS.

9.7 "DB-Lib error message 20004, severity 9: Read from SQL server failed" error appears

On Linux/*nix you may encounter the following behaviour:

```
>>> from pymssql import _mssql
>>> c=_mssql.connect('hostname:portnumber','user','pass')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
_mssql.DatabaseException: DB-Lib error message 20004, severity 9:
Read from SQL server failed.
DB-Lib error message 20014, severity 9:
Login incorrect.
```

It may happen when one of the following is true:

- `freetds.conf` file cannot be found,
- `tds version` in `freetds.conf` file is not `7.0` or `4.2`,
- any character set is specified in `freetds.conf`,
- an unrecognized character set is passed to `_mssql.connect()` or *`pymssql.connect()`* method.

`"Login incorrect"` following this error is spurious, real `"Login incorrect"` messages has code=18456 and severity=14.

9.8 Unable to use long username and password

This is a solved FreeTDS problem but you need to be using FreeTDS 0.95 or newer, if you are stuck with 0.91 then keep in mind this limitation, even when you can get usernames, passwords longer than 30 to work on tsql.

9.9 More troubleshooting

If the above hasn't covered the problem you can send a message describing it to the pymssql mailing list. You can also consult FreeTDS troubleshooting page for issues related to the TDS protocol.

CHAPTER 10

Building and developing pymssql

10.1 Required software

To build `pymssql` you should have:

- Python >= 3.6 including development files. Please research your OS usual software distribution channels, e.g, `python-dev` or `python-devel` packages on Linux.
- Cython - to compile `pymssql` source files to `C`.
- setuptools - for `setup.py` support.
- setuptools_scm - for extracting version information from `git`.
- wheel - for building python wheels.
- FreeTDS >= 1.2 including development files. Please research your OS usual software distribution channels, e.g, `freetds-dev` or `freetds-devel` packages on Linux.
- GNU gperf - a perfect hash function generator, needed for FreeTDS. On Windows prebuild version is available from Chocolatey.
- win-iconv (Windows only) - developing `pymssql` on Windows also requires this library to build FreeTDS.
- OpenSSL - If you need to connect to Azure make sure FreeTDS is built with SSL support. Please research your OS usual software distribution channels. On Windows one easy way is to get prebuild libraries from Chocolatey.

For testing the follwing is required:

- Microsoft SQL Server. One possibility is to use official docker images for Microsoft SQL Server on Linux available here.
- pytest - to run the tests.
- pytest-timeout - for limiting long running tests.
- psutil - for memory monitoring.
- gevent (optional) - for async tests.

- Sqlalchemy - (optional) - for basic Sqlalchemy testing.

To build documentation Sphinx and sphinx-rtd-theme are also needed.

10.2 Windows

In addition to the requirements above when developing `pymssql` on the Windows platform you will need these additional tools installed:

- Visual Studio C++ Compiler Tools, see Python documentation for instructions on what components to install.
- Cmake for building FreeTDS and win-iconv.
- curl - for downloading FreeTDS and win-iconv.

Note: If Windows computer is not readily available then virtual machine from Microsoft could be used.

10.3 Building `pymssql` wheel

It is recommended to use python virtual environment for building `pymssql`:

```
python3 -m venv <path_to_pve>
```

if using `bash`:

```
source <path_to_pve>/bin/activate
```

or if on Windows:

```
<path_to_pve>/scripts/activate.bat
```

then install required python packages:

```
pip intall -U pip
pip install dev/requirements-dev.txt
```

If and now build wheel:

```
python3 setup.py bdist_wheel
```

or:

```
pip wheel .
```

10.4 Environment Variables

By default `setup.py` links against OpenSSL if it is available, links FreeTDS statically and looks for FreeTDS headers and libraries in places standard for the OS, but there are several environment variables for build customization:

LINK_FREETDS_STATICALLY = [YES|NO|1|0|TRUE|FALSE] default - YES, defines if FreeTDS is linked statically or not.

LINK_OPENSSL = [YES|NO|1|0|TRUE|FALSE] default - YES, defines if `pymssql` is linked against OpenSSL.

PYMSSQL_FREETDS if defined, determines prefix of the FreeTDS installation.

PYMSSQL_FREETDS_INCLUDEDIR if defined, alows to fine tune where to search for FreeTDS headers.

PYMSSQL_FREETDS_LIBDIR if defined, alows to fine tune where to search for FreeTDS libraries.

Example:

```
PYMSSQL_FREETDS=/tmp/freetds python3 setup.py bdist_wheel
```

10.5 Building FreeTDS and `pymssql` from scratch

If one wants to use some specific FreeTDS version then there is a script `dev/build.py` that downloads and builds required FreeTDS version sources (and win-conv on Windows) and builds `pymssql` wheel. Run:

```
python dev/build.py --help
```

for supported options.

10.6 Testing

Danger: ALL DATA IN TESTING DBS WILL BE DELETED !!!!

You will need to install two additional packages for testing:

```
easy_install pytest SQLAlchemy
```

You should build the package with:

```
python setup.py develop
```

You need to setup a `tests.cfg` file in `tests/` with the correct DB connection information for your environment:

```
cp tests/tests.cfg.tpl tests/tests.cfg
vim|emacs|notepad tests/tests.cfg
```

To run the tests:

```
cd tests # optional
py.test
```

Which will go through and run all the tests with the settings from the `DEFAULT` section of `tests.cfg`.

To run with a different `tests.cfg` section:

```
py.test --pymssql-section=<secname>
```

example:

```
py.test --pymssql-section=AllTestsWillRun
```

to avoid slow tests:

```
py.test -m "not slow"
```

to select specific tests to run:

```
py.test tests/test_types.py
py.test tests/test_types.py tests/test_sprocs.py
py.test tests/test_types.py::TestTypes
py.test tests/test_types.py::TestTypes::test_image
```

CHAPTER 11

FreeTDS and dates

Explanation of how pymssql and FreeTDS can break dates.

11.1 Summary

Make sure that FreeTDS is compiled with `--enable-msdblib` `configure` option, or your queries will return wrong dates – `"2010-00-01"` instead of `"2010-01-01"`.

11.2 Details

There's an obscure problem on Linux/*nix that results in dates shifted back by 1 month. This behaviour is caused by different `dbdatecrack()` prototypes in *Sybase Open Client DB-Library/C* and the *Microsoft SQL DB Library for C*. The first one returns month as 0..11 whereas the second gives month as 1..12. See this FreeTDS mailing list post, Microsoft manual for dbdatecrack(), and Sybase manual for dbdatecrack() for details.

FreeTDS, which is used on Linux/*nix to connect to *Sybase* and *MS SQL* servers, tries to imitate both modes:

- Default behaviour, when compiled without `--enable-msdblib`, gives `dbdatecrack()` which is Sybase-compatible,
- When configured with `--enable-msdblib`, the `dbdatecrack()` function is compatible with *MS SQL* specs.

pymssql requires *MS SQL* mode, evidently. Unfortunately at runtime we can't reliably detect which mode FreeTDS was compiled in (as of FreeTDS 0.63). Thus at runtime it may turn out that dates are not correct. If there was a way to detect the setting, pymssql would be able to correct dates on the fly.

If you can do nothing about FreeTDS, there's a workaround. You can redesign your queries to return string instead of bare date:

```
SELECT datecolumn FROM tablename
```

can be rewritten into:

```
SELECT CONVERT(CHAR(10),datecolumn,120) AS datecolumn FROM tablename
```

This way SQL will send you string representing the date instead of binary date in datetime or smalldatetime format, which has to be processed by FreeTDS and pymssql.

CHAPTER 12

Connecting to Azure SQL Database

Starting with version 2.1.1 pymssql can be used to connect to *Microsoft Azure SQL Database*.

Make sure the following requirements are met:

- Use FreeTDS 0.91 or newer
- Use TDS protocol 7.1 or newer
- Make sure FreeTDS is built with SSL support
- Specify the database name you are connecting to in the *database* parameter of the relevant `connect()` call
- **IMPORTANT**: Do not use `username@server.database.windows.net` for the *user* parameter of the relevant `connect()` call! You must use the shorter `username@server` form instead!

Example:

```
pymssql.connect("xxx.database.windows.net", "username@xxx", "password", "db_name")
```

or, if you've defined `myalias` in the `freetds.conf` FreeTDS config file:

```
[myalias]
host = xxx.database.windows.net
tds version = 7.1
...
```

then you could use:

```
pymssql.connect("myalias", "username@xxx", "password", "db_name")
```

CHAPTER 13

Docker

(Experimental)

There is a pymssql Docker image on the Docker Registry at:

https://registry.hub.docker.com/u/pymssql/pymssql/

The image bundles:

- Ubuntu 14.04 LTS (trusty)
- Python 2.7.6
- pymssql 2.1.2.dev
- FreeTDS 0.91
- SQLAlchemy 0.9.8
- Alembic 0.7.4
- Pandas 0.15.2
- Numpy 1.9.1
- IPython 2.3.1

To try it, first download the image (this requires Internet access and could take a while):

```
docker pull pymssql/pymssql
```

Then run a Docker container using the image with:

```
docker run -it --rm pymssql/pymssql
```

By default, if no command is specified, an IPython shell is invoked. You can override the command if you wish – e.g.:

```
docker run -it --rm pymssql/pymssql bin/bash
```

Here's how using the Docker container looks in practice:

```
$ docker pull pymssql/pymssql
...
$ docker run -it --rm pymssql/pymssql
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
Type "copyright", "credits" or "license" for more information.

IPython 2.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import pymssql; pymssql.__version__
Out[1]: u'2.1.1'

In [2]: import sqlalchemy; sqlalchemy.__version__
Out[2]: '0.9.7'

In [3]: import pandas; pandas.__version__
Out[3]: '0.14.1'
```

See the Docker docs for installation instructions for a number of platforms; you can try this link: https://docs.docker.com/installation/#installation

CHAPTER 14

Change log

14.1 Recent Changes

14.2 Version 2.2.2 - 2021-07-24 - Mikhail Terekhov

14.2.1 General

- Use FreeTDS-1.3 for official wheels on PyPi.
- On macOS use delocate to bundle dependencies when building wheels.
- Some documentation changes.

14.3 Version 2.2.1 - 2021-04-15 - Mikhail Terekhov

14.3.1 General

- Publish Linux wheels for the all supported platforms. manylinux1 wheels are not compatible with modern glibc and OpenSSL.
- Add readthedocs configuration file.

14.4 Version 2.2.0 - 2021-04-08 - Mikhail Terekhov

14.4.1 General

- Add Python-3.9 to the build and test matrix.
- Drop support for Python2 and Python3 < 3.6.

- Use FreeTDS-1.2.18 for official wheels on PyPi.

14.4.2 Features

- Support bulk copy (#279). Thanks to Simon.StJG (PR-689).
- Wheels on PyPI link FreeTDS statically.
- Wheels on PyPI linked against OpenSSL.
- Convert pymssql to a package. **Potential compatibility issue:** projects using low level *_mssql* module need to import it from *pymssql* first.

14.4.3 Bug fixes

- Fixed a deadlock caused by a missing release of GIL (#540), thanks to filip.stefanak (PR-541) and Juraj Bubniak (PR-683).
- Prevents memory leak on login failure. Thanks to caogtaa and Simon.StJG (PR-690).
- Fix check for TDS version (#652 and #669).
- Documentation fixes. Thanks to Simon Biggs, Shane Kimble, Simon.StJG and Dale Evans.

14.4.4 Internals

- Introduce script dev/build.py to build FreeTDS and pymssql wheels.
- Simplify setup.py, introduce environment variables to select FreeTDS includes and libraries.

14.5 Version 2.1.5 - 2020-09-17 - Mikhail Terekhov

14.5.1 General

- Revert deprecation
- Support Python-3.8. Update tests for Python-3.8 compatibility.
- Use correct language level for building Cython extension.
- Fix FreeTDS version checks. Add check for version 7.4.
- Use Github Actions for building wheels for Linux, macOS and Windows.
- Drop bundled FreeTDS-0.95 binaries.
- Unless some critical bug is discovered, this will be the last release with Python2 support.

14.6 Version 2.1.4 - 2018-08-28 - Alex Hagerman

14.6.1 General

- Drop support for versions of FreeTDS older than 0.91.

- Add Python 3.7 support
- Drop Python 3.3 support

14.6.2 Features

- Support for new in SQL Server 2008 `DATE`, `TIME` and `DATETIME2` data types (GH-156). The following conditions need to be additionally met so values of these column types can be returned from the database as their native corresponding Python data types instead of as strings:
  - Underlying FreeTDS must be 0.95 or newer.
  - TDS protocol version in use must be 7.3 or newer.

  Thanks Ed Avis for the implementation. (GH-331)

14.6.3 Bug fixes

- Fix `tds_version _mssql` connection property value for TDS version. 7.1 is actually 7.1 and not 8.0.

14.7 Version 2.1.3 - 2016-06-22 - Ramiro Morales

- We now publish Linux PEP 513 manylinux wheels on PyPI.
- Windows official binaries: Rollback changes to Windows binaries we had implemented in pymssql 2.1.2; go back to using:
  - A statically linked version of FreeTDS (v0.95.95)
  - No SSL support

14.8 Version 2.1.2 - 2016-02-10 - Ramiro Morales

Attention: Windows users: You need to download and install additional DLLs

pymssql version 2.1.2 includes a change in the official Windows binaries: FreeTDS isn't statically linked as it happened up to release 2.1.1, as that FreeTDS copy lacked SSL support.

Please see http://pymssql.org/en/latest/freetds.html#windows for futher details.

We are trying to find a balance between security and convenience and will be evaluating the situation for future releases. Your feedback is greatly welcome.

14.8.1 Features

- Add ability to set TDS protocol version from pymssql when connecting to SQL Server. For the remaining pymssql 2.1.x releases its default value will be 7.1 (GH-323)
- Add Dockerfile and a Docker image and instructions on how to use it (GH-258). This could be a convenient way to use pymssql without having to build stuff. See http://pymssql.readthedocs.org/en/latest/intro.html#docker Thanks Marc Abramowitz.

- Floating point values are now accepted as Stored Procedure arguments (GH-287). Thanks Runzhou Li (Leo) for the report and Bill Adams for the implementation.
- Send pymssql version in the appname TDS protocol login record field when the application doesn't provide one (GH-354)

14.8.2 Bug fixes

- Fix a couple of very common causes of segmentation faults in presence of network a partition between a pymssql-based app and SQL Server (GH-147, GH-271) Thanks Marc Abramowitz. See also GH-373.
- Fix failures and inconsistencies in query parameter interpolation when UTF-8-encoded literals are present (GH-185). Thanks Bill Adams. Also, GH-291.
- Fix `login_timeout` parameter of `pymssql.connect()` (GH-318)
- Fixed some cases of `cursor.rowcont` having a -1 value after iterating over the value returned by pymssql cursor `fetchmany()` and `fetchone()` methods (GH-141)
- Remove automatic treatment of string literals passed in queries that start with `'0x'` as hexadecimal values (GH-286)
- Fix build fatal error when using Cython >= 0.22 (GH-311)

14.8.3 Internals

- Add Appveyor hosted CI setup for running tests on Windows (GH-347)
- Travis CI: Use newer, faster, container-based infrastructure. Also, test against more than one FreeTDS version.
- Make it possible to build official release files (sdist, wheels) on Travis & AppVeyor.

14.9 Version 2.1.1 - 2014-11-25 - Ramiro Morales

14.9.1 Features

- Custom message handlers (GH-139)

  The DB-Library API includes a callback mechanism so applications can provide functions known as *message handlers* that get passed informative messages sent by the server which then can be logged, shown to the user, etc.

  `_mssql` now allows you to install your own *message handlers* written in Python. See the `_msssql` examples and reference sections of the documentation for more details.

  Thanks Marc Abramowitz.

- Compatibility with Azure

  It is now possible to transparently connect to SQL Server instances accessible as part of the Azure cloud services.

  Note: If you need to connect to Azure make sure you use FreeTDS 0.91 or newer.

- Customizable per-connection initialization SQL clauses (both in `pymssql` and `_mssql`) (GH-97)

  It is now possible to customize the SQL statements sent right after the connection is established (e.g. `'SET ANSI_NULLS ON;'`). Previously it was a hard-coded list of queries. See the `_mssql.MSSQLConnection` documentation for more details.

  Thanks Marc Abramowitz.

- Added ability to handle instances of `uuid.UUID` passed as parameters for SQL queries both in `pymssql` and `_mssql`. (GH-209)

  Thanks Marat Mavlyutov.

- Allow using SQL Server autocommit mode from `pymssql` at connection opening time. This allows e.g. DDL statements like `DROP DATABASE` to be executed. (GH-210)

  Thanks Marat Mavlyutov.

- Documentation: Explicitly mention minimum versions supported of Python (2.6) and SQL Server (2005).
- Incremental enhancements to the documentation.

14.9.2 Bug fixes

- Handle errors when calling Stored Procedures via the `.callproc()` pymssql cursor method. Now it will raise a DB-API `DatabaseException`; previously it allowed a `_mssql.MSSQLDatabaseException` exception to surface.
- Fixes in `tds_version` `_mssql` connections property value

  Made it work with TDS protocol version 7.2. (GH-211)

  The value returned for TDS version 7.1 is still 8.0 for backward compatibility (this is because such feature got added in times when Microsoft documentation labeled the two protocol versions that followed 7.0 as 8.0 and 9.0; later it changed them to 7.1 and 7.2 respectively) and will be corrected in a future release (2.2).

- PEP 249 compliance (GH-251)

  Added type constructors to increase compatibility with other libraries.

  Thanks Aymeric Augustin.

- pymssql: Made handling of integer SP params more robust (GH-237)
- Check lower bound value when convering integer values from to Python to SQL (GH-238)

14.9.3 Internals

- Completed migration of the test suite from nose to py.test.
- Added a few more test cases to our suite.
- Tests: Modified a couple of test cases so the full suite can be run against SQL Server 2005.
- Added testing of successful build of documentation to Travis CI script.
- Build process: Cleanup intermediate and ad-hoc anciliary files (GH-231, GH-273)
- setup.py: Fixed handling of release tarballs contents so no extraneous files are shipped and the documentation tree is actually included. Also, removed unused code.

14.10 Version 2.1.0 - 2014-02-25 - Marc Abramowitz

14.10.1 Features

- Sphinx-based documentation (GH-149)

  Read it online at http://pymssql.org/

  Thanks, Ramiro Morales!

  See:

  - https://github.com/pymssql/pymssql/pull/149
  - https://github.com/pymssql/pymssql/pull/162
  - https://github.com/pymssql/pymssql/pull/164
  - https://github.com/pymssql/pymssql/pull/165
  - https://github.com/pymssql/pymssql/pull/166
  - https://github.com/pymssql/pymssql/pull/167
  - https://github.com/pymssql/pymssql/pull/169
  - https://github.com/pymssql/pymssql/pull/174
  - https://github.com/pymssql/pymssql/pull/175

- "Green" support (GH-135)

  Lets you use pymssql with cooperative multi-tasking systems like gevent and have pymssql call a callback when it is waiting for a response from the server. You can set this callback to yield to another greenlet, coroutine, etc. For example, for gevent, you could do:

```
def wait_callback(read_fileno):
    gevent.socket.wait_read(read_fileno)

pymssql.set_wait_callback(wait_callback)
```

  The above is useful if you're say, running a gunicorn server with the gevent worker. With this callback in place, when you send a query to SQL server and are waiting for a response, you can yield to other greenlets and process other requests. This is super useful when you have high concurrency and/or slow database queries and lets you use less gunicorn worker processes and still handle high concurrency.

  See https://github.com/pymssql/pymssql/pull/135

- Better error messages.

  E.g.: For a connection failure, instead of:

  > pymssql.OperationalError: (20009, 'Net-Lib error during Connection refused')

  the dberrstr is also included, resulting in:

  > pymssql.OperationalError: (20009, 'DB-Lib error message 20009, severity 9:nUnable to connect: Adaptive Server is unavailable or does not existnNet-Lib error during Connection refusedn')

  See: * https://github.com/pymssql/pymssql/pull/151

  In the area of error messages, we also made this change:

  execute: Raise ColumnsWithoutNamesError when as_dict=True and missing column names (GH-160)

because the previous behavior was very confusing; instead of raising an exception, we would just return row dicts with those columns missing. This prompted at least one question on the mailing list (https://groups.google.com/forum/?fromgroups#!topic/pymssql/JoZpmNZFtxM), so we thought it was better to handle this explicitly by raising an exception, so the user would understand what went wrong.

See: * https://github.com/pymssql/pymssql/pull/160 * https://github.com/pymssql/pymssql/pull/168

- Performance improvements

  You are most likely to notice a difference from these when you are fetching a large number of rows.

  - Reworked row fetching (GH-159)

    There was a rather large amount of type conversion occuring when fetching a row from pymssql. The number of conversions required have been cut down significantly with these changes. Thanks Damien, Churchill!

    See: * https://github.com/pymssql/pymssql/pull/158 * https://github.com/pymssql/pymssql/pull/159

  - Modify get_row() to use the CPython tuple API (GH-178)

    This drops the previous method of building up a row tuple and switches to using the CPython API, which allows you to create a correctly sized tuple at the beginning and simply fill it in. This appears to offer around a 10% boost when fetching rows from a table where the data is already in memory. Thanks Damien, Churchill!

    See: * https://github.com/pymssql/pymssql/pull/178

- MSSQLConnection: Add *with* (context manager) support (GH-171)

  This adds *with* statement support for MSSQLConnection in the *_mssql* module – e.g.:

```
with mssqlconn() as conn:
    conn.execute_query("SELECT @@version AS version")
```

  We already have *with* statement support for the *pymssql* module. See:

  - https://github.com/pymssql/pymssql/pull/171

- Allow passing in binary data (GH-179)

  Use the bytesarray type added in Python 2.6 to signify that this is binary data and to quote it accordingly. Also modify the handling of str/bytes types checking the first 2 characters for b'0x' and insert that as binary data. See:

  - https://github.com/pymssql/pymssql/pull/179

- Add support for binding uuid.UUID instances to stored procedures input params (GH-143) Thanks, Ramiro Morales!

  See: * https://github.com/pymssql/pymssql/pull/143 * https://github.com/pymssql/pymssql/commit/1689c83878304f735eb38b1c63c31e210b028ea7

- The version number is now stored in one place, in pymssql_version.h This makes it easier to update the version number and not forget any places, like I did with pymssql 2.0.1

  - See https://github.com/pymssql/pymssql/commit/fd317df65fa62691c2af377e4661defb721b2699

- Improved support for using py.test as test runner (GH-183)

  - See: https://github.com/pymssql/pymssql/pull/183

- Improved PEP-8 and pylint compliance

14.10.2 Bug Fixes

- GH-142 ("Change how `*.pyx` files are included in package") - this should prevent pymssql.pyx and _mssql.pyx from getting copied into the root of your virtualenv. Thanks, @Arfrever!
  - See: https://github.com/pymssql/pymssql/issues/142
- GH-145 ("Prevent error string growing with repeated failed connection attempts.")

  See:
  - https://github.com/pymssql/pymssql/issues/145
  - https://github.com/pymssql/pymssql/pull/146
- GH-151 ("err_handler: Don't clobber dberrstr with oserrstr")
  - https://github.com/pymssql/pymssql/pull/151
- GH-152 ("_mssql.pyx: Zero init global last_msg_* vars") See: https://github.com/pymssql/pymssql/pull/152
- GH-177 ("binary columns sometimes are processed as varchar") Better mechanism for pymssql to detect that user is passing binary data.

  See: https://github.com/pymssql/pymssql/issues/177
- buffer overflow fix (GH-182)
  - See: https://github.com/pymssql/pymssql/pull/181
  - See: https://github.com/pymssql/pymssql/pull/182
- Return uniqueidentifer columns as uuid.UUID objects on Python 3

See ChangeLog.old for older history...

CHAPTER 15

TODO

15.1 Documentation

Todo: Add an example of invoking a Stored Procedure using `_mssql`.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pymssql/checkouts/v2.2.2/docs/_mssql_examples.rst line 141.)

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

V

W