Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The end-user documentation included with the redistribution, if any, must include the following acknowledgement: "This product includes software developed by Object Craft." Alternately, this acknowledgement may appear in the software itself, if and wherever such third-party acknowledgements normally appear.

THIS SOFTWARE IS PROVIDED BY OBJECT CRAFT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL OBJECT CRAFT OR THEIR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Abstract

The Sybase module is a DB-API 2.0 compliant interface to the Sybase Relational Database.

See Also:
Sybase Module Web Site
for information on Sybase module
Sybase Web Site
(http://sybase.com/)
for information on Sybase

Contents

1 Installation .................................................. 2
   1.1 Prerequisites .............................................. 2
   1.2 Installing ................................................ 2
   1.3 Testing ................................................... 3
   1.4 Installing Sybase on Linux ............................ 4
1 Installation

1.1 Prerequisites

- Python 1.5.2 or later.

- C compiler

  The Sybase package contains an extension module written in C. The extension module is used by the Python wrapper module `Sybase.py`.

- Sybase client libraries

  The Sybase package uses the Sybase client libraries which should have come as part of your Sybase installation.
  
  If you are using Linux, Sybase provide a free version of their database which can be downloaded from http://linux.sybase.com/.

- mxDateTime

  If the mxDateTime package is installed the Sybase module can return `datetime` values as `DateTime` objects. If mxDateTime is not present the module will use the `DateTime` object defined in the `sybasect` extension module.

1.2 Installing

The Sybase package uses the `distutils` package so all you need to do is type the following command as root:

```
python setup.py install
```

To disable bulkcopy support you should use the following commands:
python setup.py build_ext -U WANT_BULKCOPY
python setup.py install

The default build does not enable threading in the extension module so if you want threading enabled you will have to do this:

    python setup.py build_ext -D WANT_THREADS
    python setup.py install

This is the first release which supports FreeTDS. To compile for FreeTDS you should use the following commands:

    python setup.py build_ext -D HAVE_FREETDS -U WANT_BULKCOPY
    python setup.py install

To build with FreeTDS and threads use the following commands:

    python setup.py build_ext -D WANT_THREADS,HAVE_FREETDS -U WANT_BULKCOPY
    python setup.py install

You will probably experience some segfaults with FreeTDS using the Cursor callproc() method and when using named arguments.

If you have problems with the installation step, edit the setup.py file to specify where Sybase is installed and the name of the client libraries. Make sure that you contact the package author so that the installation process can be made more robust for other people.

1.3 Testing

The most simple way to test the Sybase package is to run a test application. The arguments to the Sybase.connect() function are server (from the interfaces file), username, password, and database. The database argument is optional. Make sure you substitute values that will work in your environment.
>>> import Sybase
>>> db = Sybase.connect('SYBASE', 'sa', '')
>>> c = db.cursor()
>>> c.callproc('sp_help')
>>> for t in c.description:
...    print t
... ('Name', 0, 0, 30, 0, 0, 0)
('Owner', 0, 0, 30, 0, 0, 32)
('Object_type', 0, 0, 22, 0, 0, 32)
>>> for r in c.fetchall():
...    print r
... ('spt_datatype_info', 'dbo', 'user table')
('spt_datatype_info_ext', 'dbo', 'user table')
('spt_limit_types', 'dbo', 'user table')
: :
('sp_prtsybsysmsgs', 'dbo', 'stored procedure')
('sp_validlang', 'dbo', 'stored procedure')
>>> c.nextset()
1
>>> for t in c.description:
...    print t
...
('User_type', 0, 0, 15, 0, 0, 0)
('Storage_type', 0, 0, 15, 0, 0, 0)
('Length', 6, 0, 1, 0, 0, 0)
('Nulls', 11, 0, 1, 0, 0, 0)
('Default_name', 0, 0, 15, 0, 0, 32)
('Rule_name', 0, 0, 15, 0, 0, 32)

1.4 Installing Sybase on Linux

There is a very nice guide to installing Sybase on Linux at Linux Planet. http://www.linuxplanet.com/linuxplanet/tutorials/4323/1/
(Thanks to Tim Churches for pointing this out).

2 Sybase — Provides interface to Sybase relational database

The Sybase module contains the following:

__version__
A string which specifies the version of the Sybase module.

use_datetime
When you import the Sybase module it will try to import the mxDateTime module. If the mxDateTime module is successfully imported then this variable will be set to 1. All datetime columns will then be returned as mxDateTime.DateTime objects.

If you do not wish to use mxDateTime.DateTime objects, set this to 0 immediately after importing the Sybase module. All datetime columns will then be returned as sybasect.DateTime objects.

apilevel
Specifies the level of DB-API compliance. Currently set to ‘2.0’.

threadsafety
Specifies the DB-API threadsafety. The Sybase module allows threads to share the module, connections
and cursors.

paramstyle
Specifies the DB-API parameter style. This variable is set to the value ‘named’ which indicates that the
Sybase module uses named parameters. For example:

    c.execute("select * from titles where title like @arg",
               {'@arg': 'The %'})

exception Warning
Exception raised for important warnings like data truncations while inserting, etc. It is a subclass of the
Python StandardError (defined in the module exceptions).

exception Error
Exception that is the base class of all other error exceptions. You can use this to catch all errors with
one single except statement. It is a subclass of the Python StandardError (defined in the module exceptions).

exception InterfaceError
Exception raised for errors that are related to the database interface rather than the database itself. It is a
subclass of Error.

exception DatabaseError
Exception raised for errors that are related to the database. It is a subclass of Error.

exception DataError
Exception raised for errors that are due to problems with the processed data like division by zero, numeric
value out of range, etc. It is a subclass of DatabaseError.

exception OperationalError
Exception 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. It is a subclass of
DatabaseError.

exception IntegrityError
Exception raised when the relational integrity of the database is affected, e.g. a foreign key check fails. It is a
subclass of DatabaseError.

exception InternalError
Exception raised when the database encounters an internal error, e.g. the cursor is not valid anymore, the
transaction is out of sync, etc. It is a subclass of DatabaseError.

exception ProgrammingError
Exception raised for programming errors, e.g. table not found or already exists, syntax error in the SQL
statement, wrong number of parameters specified, etc. It is a subclass of DatabaseError.

exception NotSupportedError
Exception 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.
It is a subclass of DatabaseError.

This is the exception inheritance layout:
StandardError
|__Warning
|__Error
 |__InterfaceError
 |__DatabaseError
 |__DataError
 |__OperationalError
 |__IntegrityError
 |__InternalError
 |__ProgrammingError
 |__NotSupportedError

STRING
An instance of the DBAPITypeObject class which compares equal to all Sybase type codes for string-based column types (char, varchar, text).

BINARY
An instance of the DBAPITypeObject class which compares equal to all Sybase type codes which describe binary columns (image, binary, varbinary).

NUMBER
An instance of the DBAPITypeObject class which compares equal to all Sybase type codes which describe numeric columns (bit, tinyint, smallint, int, decimal, numeric, float, real, money, smallmoney).

DATETIME
An instance of the DBAPITypeObject class which compares equal to all Sybase type codes which describe date/time columns (datetime, smalldatetime).

ROWID
An instance of the DBAPITypeObject class which compares equal to all Sybase type codes which describe date/time columns (decimal, numeric).

Date (year, month, day)
DB-API 2.0 function which returns a Sybase datetime value for the supplied arguments.

Time (hour, minute, second)
Sybase does not have a native type for representing times – this DB-API 2.0 function is not implemented.

Timestamp (year, month, day, hour, minute, second)
DB-API 2.0 function which returns a Sybase datetime value for the supplied arguments.

DateFromTicks (ticks)
DB-API 2.0 function which returns a Sybase datetime value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

TimeFromTicks (ticks)
Sybase does not have a native type for representing times – this DB-API 2.0 function is not implemented.

TimestampFromTicks (ticks)
DB-API 2.0 function which returns a Sybase datetime value from the given ticks value (number of seconds since the epoch; see the documentation of the standard Python time module for details).

Binary (str)
DB-API 2.0 function which constructs an object capable of holding a binary (long) string value.

class Cursor (owner)
Return a new instance of the Cursor class which implements the DB-API 2.0 cursor functionality.
Cursor objects are usually created via the cursor() method of the Connection object.
The owner argument must be an instance of the Connection class.

class Bulkcopy (owner, table, direction = CS_BLK_IN, arraysize = 20)
Return a new instance of the Bulkcopy class.
The owner argument must be an instance of the Connection class. A bulk copy context will be established for the table named in the table argument, the bulkcopy direction must be either CS_BLK_IN or CS_BLK_OUT as defined in the Sybase module. arraysize specifies the number of in-memory rows that will be batched for each DB request.

This functionality can only be called when the Connection is in auto_commit mode. Otherwise a ProgrammingError exception is raised.

Bulkcopy objects are usually created via the bulkcopy() method of the Connection object.

This is an extension of the DB-API 2.0 specification.

class Connection (dsn, user, passwd [ . . . ])
Return a new instance of the Connection class which implements the DB-API 2.0 connection functionality.

The dsn argument identifies the Sybase server, user and passwd are the Sybase username and password respectively.

The optional arguments are the same as those supported by the connect() function.

connect (dsn, user, passwd [ . . . ])
Implements the DB-API 2.0 connect() function.

Creates a new Connection object passing the function arguments to the Connection constructor. The optional arguments and their effect are:

database = None
Specifies the database to use - has the same effect as the following SQL.

    use database

strip = 0
If non-zero then all char columns will be right stripped of whitespace.

auto_commit = 0
Controls Sybase chained transaction mode. When non-zero, chained transaction mode is turned off.

From the Sybase SQL manual:

If you set chained transaction mode, Adaptive Server implicitly invokes a begin transaction before the following statements: delete, insert, open, fetch, select, and update. You must still explicitly close the transaction with a commit.

bulkcopy = 0
Must be non-zero if you are going to perform bulkcopy operations on the connection. You will also need to turn off chained transactions in order to use bulkcopy (auto_commit=1).

delay_connect = 0
If non-zero the returned Connection object will be initialised but not connected. This allows you to set additional options on the connection before completing the connection to the server. Call the connect() method to complete the connection.

    db = Sybase.connect(‘SYBASE’, ‘sa’, ‘’, delay_connect = 1)
    db.set_property(Sybase.CS_HOSTNAME, ‘secret’)
    db.connect()

locale = None
Controls the locale of the connection to match that of the server.

    db = Sybase.connect(‘SYBASE’, ‘sa’, ‘’, locale = ‘utf8’)

locking = 1

Controls whether or not thread locks will be used on the connection object. When non-zero, the connection allows connections and cursors to be shared between threads. If your program is not multi-threaded you can gain a slight performance improvement by passing zero in this argument.

datetime = ‘auto’

Controls the type used when returning a date or time.

The Sybase module imports the low level sybasect extension module via

    from sybasect import *

which means that the Sybase module inherits all of the objects defined in that module.

Some of the functions will be useful in your programs.

datetime(str [, type = CS_DATETIME_TYPE])

Creates a new instance of the DateTime class. This is used to construct native Sybase datetime and smalldatetime values.

The string passed in the str argument is converted to a datetime value of the type specified in the optional type argument.

CS_DATETIME_TYPE represents the datetime Sybase type and CS_DATETIME4_TYPE represents smalldatetime.

The DateTime class is described in the sybasect module.

money(num)

Creates a new instance of the Money class. This is used to construct native Sybase money values.

The value passed in the num argument is converted to a native Sybase money value.

The Money class is described in the sybasect module.

numeric(num, [precision = -1 [, scale = -1 ]])

Creates a new instance of the Numeric class. This is used to construct native Sybase numeric and decimal values.

Converts the value passed in the num argument to a native Sybase numeric value. The precision and scale arguments control the precision and scale of the returned value.

The Numeric class is described in the sybasect module.

2.1 Connection Objects

Implements the DB-API 2.0 Connection class.

Connection objects have the following interface:

close()

Implements the DB-API 2.0 connection close() method.

Forces the database connection to be closed immediately. Any operation on the connection (including cursors) after calling this method will raise an exception.

This method is called by the __del__() method.

commit([name = None])

Implements the DB-API 2.0 connection commit() method.

Calling this method commits any pending transaction to the database. By default Sybase transaction chaining is enabled. If you pass auto_commit = 1 to the connect() function when creating this Connection object then chained transaction mode will be turned off.

From the Sybase manual:
If you set chained transaction mode, Adaptive Server implicitly invokes a begin transaction before the following statements: delete, insert, open, fetch, select, and update. You must still explicitly close the transaction with a commit.

The optional name argument is an extension of the DB-API 2.0 specification. It allows you to make use of the Sybase ability to nest and name transactions.

### rollback(name=None)
Implements the DB-API 2.0 connection `rollback()` method.
Tells Sybase to roll back to the start of any pending transaction. Closing a connection without committing the changes first will cause an implicit rollback to be performed.

The optional name argument is an extension of the DB-API 2.0 specification. It allows you to make use of the Sybase ability to nest and name transactions.

### cursor()
Implements the DB-API 2.0 connection `cursor()` method.
Returns a new `Cursor` object using the connection.

### begin(name=None)
This is an extension of the DB-API 2.0 specification.
If you have turned off chained transaction mode via the `auto_commit` argument to the connection constructor then you can use this method to begin a transaction. It also allows you to use the nested transaction support in Sybase.

### connect()
This is an extension of the DB-API 2.0 specification.
If you pass a `TRUE` value to the `delay_connect` argument to the `connect()` function then you must call this method to complete the server connection process. This is useful if you wish to set connection properties which cannot be set after you have connected to the server.

### get_property(prop)
This is an extension of the DB-API 2.0 specification.
Use this function to retrieve properties of the connection to the server.

```python
import Sybase
db = Sybase.connect('SYBASE', 'sa', '', 'pubs2')
print db.get_property(Sybase.CS_TDS_VERSION)
```

### set_property(prop, value)
This is an extension of the DB-API 2.0 specification.
Use this function to set properties of the connection to the server.

### execute(sql)
This is an extension of the DB-API 2.0 specification.
This method executes the SQL passed in the `sql` argument via the `ct_command(CS_LANG_CMD, ...) Sybase function. This is what programs such as `sqsh` use to send SQL to the server. The return value is a list of logical results. Each logical result is a list of row tuples.

The disadvantage to using this method is that you are not able to bind binary parameters to the command, you must format the parameters as part of the SQL string in the `sql` argument.

### bulkcopy(table, direction=CS_BLK_IN, arraysize=20)
This is an extension of the DB-API 2.0 specification.
Returns a new `Bulkcopy` object using the connection. The `table` argument identifies the table which you wish to perform bulkcopy operations upon. The `arraysize` argument specifies the number of rows stored in-memory for each DB request.

The `direction` argument controls the direction of the bulkcopy operation. Specify `Sybase.CS_BLK_IN` to copy data in, or `Sybase.CS_BLK_OUT` to copy data out.

### bulkcopy(table, out=1, arraysize=20)
A more convenient way of specifying BCP out.
2.2 Cursor Objects

Implements the DB-API 2.0 `Cursor` class.

`Cursor` objects have the following interface:

**description**

The DB-API 2.0 cursor `description` member.

A list of 7-item tuples. Each of the tuples describes one result column: `(name, type_code, display_size, internal_size, precision, scale, null_ok)`. This attribute will be None for operations which do not return rows or if the cursor has not had an operation invoked via `execute()` or `executemany()`.

The `type_code` can be interpreted by comparing it to the `DBAPITypeObject` objects STRING, BINARY, NUMBER, DATETIME, or ROWID.

**rowcount**

The DB-API 2.0 cursor `rowcount` member.

This attribute reports the number of rows that the last `execute()` or `executemany()` method produced or affected.

The attribute is -1 if no `execute()` or `executemany()` has been performed on the cursor.

**callproc**(procname [parameters])

Implements the DB-API 2.0 cursor `callproc()` method.

Calls the stored database procedure named in the `procname` argument. The optional `parameters` argument can be a sequence or dictionary which contains one entry for each argument that the procedure expects. For example:

```python
    c.callproc('sp_help', ['titles'])
    c.callproc('sp_help', {'@objname': 'titles'})
```

The DB-API 2.0 specification says:

The result of the call is returned as modified copy of the input sequence. Input parameters are left untouched, output and input/output parameters replaced with possibly new values.

This method is is not DB-API compliant in because there does not seem to be a way to query Sybase to determine which parameters are output parameters. This method returns None.

The procedure may also provide a result set as output. This can be retrieved via the `fetchone()`, `fetchmany()`, and `fetchall()` methods.

**close**()

Implements the DB-API 2.0 cursor `close()` method.

Cancels any pending results immediately. Any operation on the cursor after calling this method will raise an exception.

This method is called by the `__del__()` method.

**execute**(sql [params])

Implements the DB-API 2.0 cursor `execute()` method.

Prepares a dynamic SQL command on the Sybase server and executes it. The optional `params` argument is a dictionary of parameters which are bound as parameters to the dynamic SQL command. Sybase uses name place holders to specify which the parameters will be used by the SQL command.

```python
    import Sybase
    db = Sybase.connect('SYBASE', 'sa', '', 'pubs2')
    c = db.cursor()
    c.execute("select * from titles where price > @price", { '@price': 15.00 })
    c.fetchall()
```
The prepared dynamic SQL will be reused by the cursor if the same SQL is passed in the sql argument. This is most effective for algorithms where the same operation is used, but different parameters are bound to it (many times).

The method returns None.

**executemany (sql, params_seq)**

Implements the DB-API 2.0 cursor executemany() method.

Calls the execute() method for every parameter sequence in the sequence passed as the params_seq argument.

The method returns None.

**fetchone ()**

Implements the DB-API 2.0 cursor fetchone() method.

Fetches the next row of a logical result and returns it as a tuple. None is returned when no more rows are available in the logical result.

**fetchmany ([size = cursor.arraysize])**

Implements the DB-API 2.0 cursor fetchmany() method.

Fetches the next set of rows of a logical result, returning a list of tuples. An empty list is returned when no more rows are available.

The number of rows to fetch per call is specified by the optional size argument. If size is not supplied, the arraysize member determines the number of rows to be fetched. The method will try to fetch the number of rows indicated by the size parameter. If this is not possible due to the specified number of rows not being available, fewer rows will be returned.

**fetchall ()**

Implements the DB-API 2.0 cursor fetchall() method.

Fetches all remaining rows of a logical result returning them as a list of tuples.

**nextset ()**

Implements the DB-API 2.0 cursor nextset() method.

Makes the cursor skip to the next logical result, discarding any remaining rows from the current logical result.

If there are no more logical results, the method returns None. Otherwise, it returns 1 and subsequent calls to the fetchone(), fetchmany(), and fetchall() methods will return rows from the next logical result.

**arraysize**

The DB-API 2.0 cursor arraysize member.

This read/write attribute specifies the number of rows to fetch at a time with fetchmany(). It defaults to 1 meaning to fetch a single row at a time.

**setinputsizes (size)**

Implements the DB-API 2.0 cursor setinputsizes() method.

This method does nothing – it is provided for DB-API compliance.

**setoutputsize (size [, column])**

Implements the DB-API 2.0 cursor setoutputsize() method.

This method does nothing – it is provided for DB-API compliance.

### 2.3 Bulkcopy Objects

This is an extension of the DB-API 2.0 specification.

This object provides an interface to the Sybase bulkcopy functionality.

**rowxfer ([data])**

If the Bulkcopy object direction is CS_BLK_IN then the sequence passed as the data argument is sent as
one row to the server. If the direction is CS_BLK_OUT then one row will be returned from the server. If there are no more rows, None is returned.

**batch()**
Marks a complete bulkcopy batch. The number of rows transferred in the batch is returned.

**done()**
Marks a complete bulkcopy operation. The number of rows transferred in the batch is returned. done() must be called (or the Bulkcopy object destroyed) to flush any outstanding cached rows to the DB. In addition, any other operation on the associated Connection object will fail until done is called.

**__iter__()**
Returns an iterator that will iterate over all the rows in the table returned by the bulkcopy out operation. The Bulkcopy object should be created with the CS_BLK_OUT direction argument. Intermixed use of the iterator returned from **__iter__** and the rowxfer to retrieve rows is supported; both use the same underlying function and will return all rows without missing or duplicating any row.

**rows()**
An alias for **__iter__**

**arraysize** *(T)*
The arraysize passed to the constructor. Up to this many rows will be cached in memory and sent to the DB server in one request. Read only.

**totalcount** *(A)*
The count of the total number of lines passed to/from the DB server. Read only.

An example of using the Bulkcopy class follows:

```python
from Sybase import *

c = connect('server', 'user', 'password', bulkcopy=1, auto_commit=1);
c.execute("Create table #b(a int, b varchar(10), c float)")

b = c.bulkcopy('#b') # CS_BLK_IN is default
for r in range(32):
    b.rowxfer([r, ('xxx%d' % r), r * 0.1])
    if r % 5 == 4:
        ret = b.batch()
        print "post batch, batch was", ret, "count = ", b.totalcount
    ret = b.done()
    print "post done, last batch was", ret, "count = ", b.totalcount

for r in c.bulkcopy('#b', out=1):
    print r
```

### 3 sybasect — Interface to Sybase-CT library

This is not a complete reference to the Sybase CT library. Sybase produce excellent documentation which fully describes the use of the CT library.

This section describes how to access the Sybase CT library while using this wrapper module.

The sybasect extension contains the following:

#### 3.1 Types

**ContextType**
The type of CS_CONTEXT objects which wrap the Sybase CS_CONTEXT structure pointer.
ConnectionType
The type of CS_CONNECTION objects which wrap the Sybase CS_CONNECTION structure pointer.

CommandType
The type of CS_COMMAND objects which wrap the Sybase CS_COMMAND structure pointer.

BlkDescType
The type of CS_BLKDESC objects which wrap the Sybase CS_BLKDESC structure pointer.

DataFmtType
The type of CS_DATAFMT objects which wrap the Sybase CS_DATAFMT structure.

IODescType
The type of CS_IODESC objects which wrap the Sybase CS_IODESC structure.

ClientMsgType
The type of CS_CLIENTMSG objects which wrap the Sybase CS_CLIENTMSG structure.

ServerMsgType
The type of CS_SERVERMSG objects which wrap the Sybase CS_SERVERMSG structure.

DataBufType
The type of data buffers for sending and receiving data to and from Sybase. The type of object returned by DataBuf('hello').

NumericType
The type used to store Sybase CS_NUMERIC and CS_DECIMAL data values.

DateTimeType
The type used to store Sybase CS_DATETIME and CS_DATETIME4 data values.

MoneyType
The type used to store Sybase CS_MONEY and CS_MONEY4 data values.

3.2 Functions

set_global_ctx(ctx)
The sybasect module uses a CS_CONTEXT structure internally for conversions and calculations. You must allocate a context via cs_ctx_alloc() and establish the internal context using this function.

set_debug(file)
Directs all debug text to the object passed in the file argument. The file argument must be either None or an object which has write(data) and flush() methods.

The function returns the previous debug file. The default file is None.

Setting a debug file does not enable debug messages. To produce debug messages you need to set the debug member of a context, connection, command, etc.

cs_ctx_alloc([version = CS_VERSION_100])
Calls the Sybase-CT cs_ctx_alloc() function:

    result = cs_ctx_alloc(version, &ctx);

Returns a tuple containing the Sybase result code and a new instance of the CS_CONTEXT class constructed from the ctx value returned by cs_ctx_alloc(). None is returned as the CS_CONTEXT object if the result code is not CS_SUCCEED.

cs_ctx_global([version = CS_VERSION_100])
Calls the Sybase-CT cs_ctx_global() function:

    result = cs_ctx_global(version, &ctx);

Returns a tuple containing the Sybase result code and a new instance of the CS_CONTEXT class constructed from the ctx value returned by cs_ctx_global(). None is returned as the CS_CONTEXT object if the result code is not CS_SUCCEED.
**DataBuf**(obj)

Return a new instance of the `DataBuf` class. The `obj` argument is used to initialise the `DataBuf` object. For all types of `obj` other than `CS_DATAFMT` a buffer will be initialised which contains a single value.

When `obj` is a `CS_DATAFMT` object an empty buffer will be created according to the attributes of the `CS_DATAFMT` object. It is most common to create and bind a buffer in a single operation via the `ct_bind()` method of the `CS_COMMAND` class.

For example, the following code creates a set of buffers for retrieving 16 rows at a time. Note that it is your responsibility to ensure that the buffers are not released until they are no longer required.

```python
status, num_cols = cmd.ct_res_info(CS_NUMDATA)
if status != CS_SUCCEED:
    raise 'ct_res_info'
bufs = []
for i in range(num_cols):
    status, fmt = cmd.ct_describe(i + 1)
    if status != CS_SUCCEED:
        raise 'ct_describe'
    fmt.count = 16
    status, buf = cmd.ct_bind(i + 1, fmt)
    if status != CS_SUCCEED:
        raise 'ct_bind'
    bufs.append(buf)
```

**numeric**(obj[, precision = -1] [, scale = -1])

Return a new instance of the `Numeric` class.

The `obj` argument can accept any of the following types: `IntType`, `LongType`, `FloatType`, `StringType`, or `NumericType`.

If greater than or equal to zero the `precision` and `scale` arguments are used as the precision and scale attributes of the `CS_DATAFMT` which is used in the Sybase `cs_convert()` function to create the new `NumericType` object. The default values for these arguments depends upon the type being converted to `NumericType`.

<table>
<thead>
<tr>
<th>Type</th>
<th>precision</th>
<th>scale</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IntType</code></td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td><code>LongType</code></td>
<td># of digits in <code>str()</code> conversion</td>
<td>0</td>
</tr>
<tr>
<td><code>FloatType</code></td>
<td><code>CS_MAX_PREC</code></td>
<td>12</td>
</tr>
<tr>
<td><code>StringType</code></td>
<td># digits before decimal point</td>
<td># digits after decimal point</td>
</tr>
<tr>
<td><code>NumericType</code></td>
<td>input precision</td>
<td>input scale</td>
</tr>
</tbody>
</table>

**money**(obj[, type = `CS_MONEY_TYPE`])

Return a new instance of the `Money` class.

The `obj` argument can accept any of the following types: `IntType`, `LongType`, `FloatType`, `StringType`, or `MoneyType`.

Passing `CS_MONEY4_TYPE` in the `type` argument will create a smallmoney value instead of the default `money`.

**datetime**(str[, type = `CS_DATETIME_TYPE`])

Return a new instance of the `DateTime` class.

The `str` argument must be a string.

Passing `CS_DATETIME4_TYPE` in the `type` argument will create a smalldatetime value instead of the default `datetime`.

**sizeof_type**(type_code)

Returns the size of the type identified by the Sybase type code specified in the `type_code` argument.

The values expected for the `type_code` argument things like `CS_CHAR_TYPE`, `CS_TINYINT_TYPE`, etc.

**CS_DATAFMT**()

Return a new instance of the `CS_DATAFMT` class. This is used to wrap the Sybase `CS_DATAFMT` structure.
CS_IODESC()
  Return a new instance of the CS_IODESC class. This is used to wrap the Sybase CS_IODESC structure.

CS_LAYER(msgnumber)
  Return the result of applying the Sybase CS_LAYER macro to the msgnumber argument.

CS_ORIGIN(msgnumber)
  Return the result of applying the Sybase CS_ORIGIN macro to the msgnumber argument.

CS_SEVERITY(msgnumber)
  Return the result of applying the Sybase CS_SEVERITY macro to the msgnumber argument.

CS_NUMBER(msgnumber)
  Return the result of applying the Sybase CS_NUMBER macro to the msgnumber argument.

3.3 CS_CONTEXT Objects

Calling the cs_ctx_alloc() or cs_ctx_global() function will create a CS_CONTEXT object. When the CS_CONTEXT object is deallocated the Sybase cs_ctx_drop() function will be called for the context.

CS_CONTEXT objects have the following interface:

debug
  An integer which controls printing of debug messages to the debug file established by the set_debug() function. The default value is zero.

debug_msg(msg)
  If the debug member is non-zero the msg argument will be written to the debug file established by the set_debug() function.

cs_config(action, property [], value)
  Configures, retrieves and clears properties of the comm library for the context.

When action is CS_SET a compatible value argument must be supplied and the method returns the Sybase result code. The Sybase-CT cs_config() function is called like this:

```c
/* bool property value */
status = cs_config(ctx, CS_SET, property, &bool_value, CS_UNUSED, NULL);

/* int property value */
status = cs_config(ctx, CS_SET, property, &int_value, CS_UNUSED, NULL);

/* string property value */
status = cs_config(ctx, CS_SET, property, str_value, CS_NULLTERM, NULL);

/* locale property value */
status = cs_config(ctx, CS_SET, property, locale, CS_UNUSED, NULL);

/* callback property value */
status = cs_config(ctx, CS_SET, property, cslib_cb, CS_UNUSED, NULL);
```

When action is CS_GET the method returns a tuple containing the Sybase result code and the property value. The Sybase-CT cs_callback() function is called like this:

```c
/* bool property value */
status = cs_config(ctx, CS_GET, property, &bool_value, CS_UNUSED, NULL);

/* int property value */
status = cs_config(ctx, CS_GET, property, &int_value, CS_UNUSED, NULL);

/* string property value */
status = cs_config(ctx, CS_GET, property, str_buff, sizeof(str_buff), &buff_len);
```
When `action` is `CS_CLEAR` the method clears the property and returns the Sybase result code. The Sybase-CT `cs_callback()` function is called like this:

```c
status = cs_config(ctx, CS_CLEAR, property, NULL, CS_UNUSED, NULL);
```

The recognised properties are:

<table>
<thead>
<tr>
<th>property</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_EXTERNAL_CONFIG</td>
<td>bool</td>
</tr>
<tr>
<td>CS_EXTRA_INF</td>
<td>bool</td>
</tr>
<tr>
<td>CS_NOAPI_CHK</td>
<td>bool</td>
</tr>
<tr>
<td>CS_VERSION</td>
<td>int</td>
</tr>
<tr>
<td>CS_APPNAME</td>
<td>string</td>
</tr>
<tr>
<td>CS_CONFIG_FILE</td>
<td>string</td>
</tr>
<tr>
<td>CS_LOC_PROP</td>
<td>locale</td>
</tr>
<tr>
<td>CS_MESSAGE_CB</td>
<td>function</td>
</tr>
</tbody>
</table>

For an explanation of the property values and get/set/clear semantics please refer to the Sybase documentation.

`ct_callback (action, type [, cb_func = None])`

Installs, removes, or queries current Sybase callback function. This is only available when the `sybasect` module has been compiled without the `WANT_THREADS` macro defined in `sybasect.h`.

When `CS_SET` is passed as the `action` argument the Sybase-CT `ct_callback()` function is called like this:

```c
status = ct_callback(ctx, NULL, CS_SET, type, cb_func);
```

The `cb_func` argument is stored inside the `CS_CONTEXT` object. Whenever a callback of the specified type is called by the Sybase CT library, the `sybasect` wrapper locates the corresponding `CS_CONTEXT` object and calls the stored function.

If `None` is passed in the `cb_func` argument the callback identified by `type` will be removed. The Sybase result code is returned.

When `action` is `CS_GET` the Sybase-CT `ct_callback()` function is called like this:

```c
status = ct_callback(ctx, NULL, CS_GET, type, &cb_func);
```

The return value is a two element tuple containing the Sybase result code and the current callback function. When the Sybase result code is not `CS_SUCCEED` or there is no current callback, the returned function will be `None`.

The `type` argument identifies the callback function type. Currently only the following callback functions are supported.

<table>
<thead>
<tr>
<th>type</th>
<th>callback function arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_CLIENTMSG_CB</td>
<td>ctx, conn, msg</td>
</tr>
<tr>
<td>CS_SERVERMSG_CB</td>
<td>ctx, conn, msg</td>
</tr>
</tbody>
</table>

The following will allocate and initialise a CT library context then will install a callback.
from sybasect import *

def ctlib_server_msg_handler(conn, cmd, msg):
    return CS_SUCCEED

status, ctx = cs_ctx_alloc()
if status != CS_SUCCEED:
    raise CSError(ctx, 'cs_ctx_alloc')
if ctx.ct_init(CS_VERSION_100):
    raise CSError(ctx, 'ct_init')
if ctx.ct_callback(CS_SET, CS_SERVERMSG_CB,
                  ctlib_server_msg_handler) != CS_SUCCEED:
    raise CSError(ctx, 'ct_callback')

### cs_loc_alloc()
Allocates a new CS_LOCALE object which is used to control locale settings. Calls the Sybase-CT cs_loc_alloc() function like this:

```
status = cs_loc_alloc(ctx, &locale);
```

Returns a tuple containing the Sybase result code and a new instance of the CS_LOCALE class constructed from the locale returned by cs_loc_alloc(). None is returned as the CS_LOCALE object when the result code is not CS_SUCCEED.

### ct_con_alloc()
Allocates a new CS_CONNECTION object which is used to connect to a Sybase server. Calls the Sybase-CT ct_callback() function like this:

```
status = ct_con_alloc(ctx, &conn);
```

Returns a tuple containing the Sybase result code and a new instance of the CS_CONNECTION class constructed from the conn returned by ct_con_alloc(). None is returned as the CS_CONNECTION object when the result code is not CS_SUCCEED.

### ct_config(action, property [, value ])
Sets, retrieves and clears properties of the context object

When `action` is CS_SET a compatible `value` argument must be supplied and the method returns the Sybase result code. The Sybase-CT ct_config() function is called like this:

```
/* int property value */
status = ct_config(ctx, CS_SET, property, &int_value, CS_UNUSED, NULL);

/* string property value */
status = ct_config(ctx, CS_SET, property, str_value, CS_NULLTERM, NULL);
```

When `action` is CS_GET the method returns a tuple containing the Sybase result code and the property value. The Sybase-CT ct_callback() function is called like this:

```
/* int property value */
status = ct_config(ctx, CS_GET, property, &int_value, CS_UNUSED, NULL);

/* string property value */
status = ct_config(ctx, CS_GET, property, str_buff, sizeof(str_buff), &buff_len);
```

When `action` is CS_CLEAR the method clears the property and returns the Sybase result code. The Sybase-CT ct_callback() function is called like this:

```
status = ct_config(ctx, CS_CLEAR, property, NULL, CS_UNUSED, NULL);
```
The recognised properties are:
<table>
<thead>
<tr>
<th>property</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_LOGIN_TIMEOUT</td>
<td>int</td>
</tr>
<tr>
<td>CS_MAX_CONNECT</td>
<td>int</td>
</tr>
<tr>
<td>CS_NETIO</td>
<td>int</td>
</tr>
<tr>
<td>CS_NO_TRUNCATE</td>
<td>int</td>
</tr>
<tr>
<td>CS_TEXTLIMIT</td>
<td>int</td>
</tr>
<tr>
<td>CS_TIMEOUT</td>
<td>int</td>
</tr>
<tr>
<td>CS_VERSION</td>
<td>int</td>
</tr>
<tr>
<td>CS_IFILE</td>
<td>string</td>
</tr>
<tr>
<td>CS_VER_STRING</td>
<td>string</td>
</tr>
</tbody>
</table>

For an explanation of the property values and get/set/clear semantics please refer to the Sybase documentation.

**ct_exit** \{\texttt{option = CS_UNUSED}\}

Calls the Sybase \texttt{ct_exit()} function like this:

```c
status = ct_exit(ctx, option);
```

Returns the Sybase result code.

**ct_init** \{\texttt{version = CS_VERSION_100}\}

Initialises the context object and tells the CT library which version of behaviour is expected. This method must be called immediately after creating the context. The Sybase \texttt{ct_init()} function is called like this:

```c
status = ct_init(ctx, version);
```

Returns the Sybase result code.

**cs_ctx_drop**()

Calls the Sybase \texttt{cs_ctx_drop()} function like this:

```c
status = cs_ctx_drop(ctx);
```

Returns the Sybase result code.

This method will be automatically called when the \texttt{CS_CONTEXT} object is deleted. Applications do not need to call the method.

**cs_diag** \{\texttt{operation \ldots} \}

Manage Open Client/Server error messages for the context.

When \texttt{operation} is \texttt{CS_INIT} a single argument is accepted and the Sybase result code is returned. The Sybase \texttt{cs_diag()} function is called like this:

```c
status = cs_diag(ctx, CS_INIT, CS_UNUSED, CS_UNUSED, NULL);
```

When \texttt{operation} is \texttt{CS_MSGLIMIT} two additional arguments are expected; \texttt{type} and \texttt{num}. The Sybase result code is returned. The Sybase \texttt{cs_diag()} function is called like this:

```c
status = cs_diag(ctx, CS_MSGLIMIT, type, CS_UNUSED, &num);
```

When \texttt{operation} is \texttt{CS_CLEAR} an additional \texttt{type} argument is accepted and the Sybase result code is returned. The Sybase \texttt{cs_diag()} function is called like this:

```c
status = cs_diag(ctx, CS_CLEAR, type, CS_UNUSED, NULL);
```

When \texttt{operation} is \texttt{CS_GET} two additional arguments are expected; \texttt{type} which currently must be \texttt{CS_CLIENTMSG_TYPE}, and \texttt{index}. A tuple is returned which contains the Sybase result code and the requested \texttt{CS_CLIENTMSG} message. None is returned as the message object when the result code is not \texttt{CS_SUCCEED}. The Sybase \texttt{cs_diag()} function is called like this:
status = cs_diag(ctx, CS_GET, type, index, &msg);

When operation is CS_STATUS an additional type argument is accepted. A tuple is returned which contains the Sybase result code and the number of messages available for retrieval. The Sybase `cs_diag()` function is called like this:

status = cs_diag(ctx, CS_STATUS, type, CS_UNUSED, &num);

The following will retrieve and print all messages from the context.

def print_msgs(ctx):
    status, num_msgs = ctx.cs_diag(CS_STATUS, CS_CLIENTMSG_TYPE)
    if status == CS_SUCCEED:
        for i in range(num_msgs):
            status, msg = ctx.cs_diag(CS_GET, CS_CLIENTMSG_TYPE, i + 1)
            if status != CS_SUCCEED:
                continue
            for attr in dir(msg):
                print '%s: %s' % (attr, getattr(msg, attr))
        ctx.cs_diag(CS_CLEAR, CS_CLIENTMSG_TYPE)

3.4 CS_LOCALE Objects

CS_LOCALE objects are a wrapper around the Sybase CS_LOCALE structure. The objects are created by calling the `cs_loc_alloc()` method of a CS_CONTEXT object.

They have the following interface:

`cs_dt_info(action, type [ . . . ]
Sets or retrieves datetime information of the locale object

When action is CS_SET a compatible value argument must be supplied and the method returns the Sybase result code. The Sybase-CT `cs_dt_info()` function is called like this:

status = cs_dt_info(ctx, CS_SET, locale, type, CS_UNUSED,
                   &int_value, sizeof(int_value), &out_len);

When action is CS_GET the method returns a tuple containing the Sybase result code and a value. When a string value is requested an optional item argument can be passed which defaults to CS_UNUSED.

The return result depends upon the value of the type argument.

<table>
<thead>
<tr>
<th>type</th>
<th>need item?</th>
<th>return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_12HOUR</td>
<td>no</td>
<td>status, bool</td>
</tr>
<tr>
<td>CS_DT_CONV_FMT</td>
<td>no</td>
<td>status, int</td>
</tr>
<tr>
<td>CS_MONTH</td>
<td>yes</td>
<td>status, string</td>
</tr>
<tr>
<td>CS_SHORT_MONTH</td>
<td>yes</td>
<td>status, string</td>
</tr>
<tr>
<td>CS_DAYNAME</td>
<td>yes</td>
<td>status, string</td>
</tr>
<tr>
<td>CS_DATEORDER</td>
<td>no</td>
<td>status, string</td>
</tr>
</tbody>
</table>

The Sybase-CT `cs_dt_info()` function is called like this:
/* bool value */
status = cs_dt_info(ctx, CS_GET, locale, type, CS_UNUSED,
    &bool_value, sizeof(bool_value), &out_len);

/* int value */
status = cs_dt_info(ctx, CS_GET, locale, type, CS_UNUSED,
    &int_value, sizeof(int_value), &out_len);

/* string value */
status = cs_dt_info(ctx, CS_GET, locale, type, item,
    str_buff, sizeof(str_buff), &buff_len);

\textbf{cs\_loc\_drop()}\newline
Calls the Sybase \texttt{cs\_loc\_drop()} function like this:

\begin{verbatim}
status = cs_loc_drop(ctx, locale);
\end{verbatim}

Returns the Sybase result code.

This method will be automatically called when the CS\_LOCALE object is deleted. Applications do not need
to call the method.

\textbf{cs\_locale}(action, type \{, value \})
Load the object with localisation values or retrieves the locale name previously used to load the object.

When \textit{action} is \texttt{CS\_SET} a string \textit{value} argument must be supplied and the method returns the Sybase result code. The Sybase-CT \texttt{cs\_locale()} function is called like this:

\begin{verbatim}
status = cs_locale(ctx, CS_SET, locale, type, value, CS_NULLTERM, NULL);
\end{verbatim}

The recognised values for \textit{type} are:

\begin{center}
\begin{tabular}{l}
\texttt{CS\_LC\_COLLATE} \\
\texttt{CS\_LC\_CTYPE} \\
\texttt{CS\_LC\_MESSAGE} \\
\texttt{CS\_LC\_MONETARY} \\
\texttt{CS\_LC\_NUMERIC} \\
\texttt{CS\_LC\_TIME} \\
\texttt{CS\_LC\_ALL} \\
\texttt{CS\_SYB\_LANG} \\
\texttt{CS\_SYB\_CHARSET} \\
\texttt{CS\_SYB\_SORTORDER} \\
\texttt{CS\_SYB\_COLLATE} \\
\texttt{CS\_SYB\_LANG\_CHARSET} \\
\texttt{CS\_SYB\_TIME} \\
\texttt{CS\_SYB\_MONETARY} \\
\texttt{CS\_SYB\_NUMERIC}
\end{tabular}
\end{center}

When \textit{action} is \texttt{CS\_GET} the method returns a tuple containing the Sybase result code and a locale name. The Sybase-CT \texttt{cs\_locale()} function is called like this:

\begin{verbatim}
status = cs_locale(ctx, CS_GET, locale, type, str_buff, sizeof(str_buff), &str_len);
\end{verbatim}

\section*{3.5 CS\_CONNECTION Objects}

Calling the \texttt{ct\_con\_alloc()} method of a \texttt{CS\_CONTEXT} object will create a \texttt{CS\_CONNECTION} object. When the \texttt{CS\_CONNECTION} object is deallocated the Sybase \texttt{ct\_con\_drop()} function will be called for the con-
CS\_CONNECTION objects have the following interface:

**ctx**

This is a read only reference to the parent CS\_CONTEXT object. This prevents the context from being dropped while the connection still exists.

**strip**

An integer which controls right whitespace stripping of char columns. The default value is zero.

**debug**

An integer which controls printing of debug messages to the debug file established by the set\_debug() function. The default value is inherited from the CS\_CONTEXT object.

**ct\_diag**(operation [ ... ])

Manage Sybase error messages for the connection.

When operation is CS\_INIT a single argument is accepted and the Sybase result code is returned. The Sybase ct\_diag() function is called like this:

```
status = ct_diag(conn, CS_INIT, CS_UNUSED, CS_UNUSED, NULL);
```

When operation is CS\_MSGLIMIT two additional arguments are expected; type and num. The Sybase result code is returned. The Sybase ct\_diag() function is called like this:

```
status = ct_diag(conn, CS_MSGLIMIT, type, CS_UNUSED, &num);
```

When operation is CS\_CLEAR an additional type argument is accepted and the Sybase result code is returned. The Sybase ct\_diag() function is called like this:

```
status = ct_diag(conn, CS_CLEAR, type, CS_UNUSED, NULL);
```

When operation is CS\_GET two additional arguments are expected; type and index. A tuple is returned which contains the Sybase result code and the requested CS\_SERVERMSG or CS\_CLIENTMSG message. None is returned as the message object when the result code is not CS\_SUCCEED. The Sybase ct\_diag() function is called like this:

```
status = ct_diag(conn, CS_GET, type, index, &msg);
```

When operation is CS\_STATUS an additional type argument is accepted. A tuple is returned which contains the Sybase result code and the number of messages available for retrieval. The Sybase ct\_diag() function is called like this:

```
status = ct_diag(conn, CS_STATUS, type, CS_UNUSED, &num);
```

When operation is CS\_EED\_CMD two additional arguments are expected; type and index. A tuple is returned which contains the Sybase result code and a CS\_COMMAND object which is used to retrieve extended error data. The Sybase ct\_diag() function is called like this:

```
status = ct_diag(conn, CS_EED\_CMD, type, index, &eed);
```

The following will retrieve and print all messages from a connection.
```python
def print_msgs(conn, type):
    status, num_msgs = conn.ct_diag(CS_STATUS, type)
    if status != CS_SUCCEED:
        return
    for i in range(num_msgs):
        status, msg = conn.ct_diag(CS_GET, type, i + 1)
        if status != CS_SUCCEED:
            continue
        for attr in dir(msg):
            print '%s: %s' % (attr, getattr(msg, attr))

def print_all_msgs(conn):
    print_msgs(conn, CS_SERVERMSG_TYPE)
    print_msgs(conn, CS_CLIENTMSG_TYPE)
    conn.ct_diag(CS_CLEAR, CS_ALLMSG_TYPE)

def ct_cancel(type):
    Calls the Sybase ct_cancel() function and returns the Sybase result code. The Sybase ct_cancel() function is called like this:

    status = ct_cancel(conn, NULL, type);

def ct_connect(server = None):
    Calls the Sybase ct_connect() function and returns the Sybase result code. The Sybase ct_connect() function is called like this:

    status = ct_connect(conn, server, CS_NULLTERM);

    When no server argument is supplied the Sybase ct_connect() function is called like this:

    status = ct_connect(conn, NULL, 0);

def ct_cmd_alloc():
    Allocates and returns a new CS_COMMAND object which is used to send commands over the connection. Calls the Sybase-CT ct_callback() function like this:

    status = ct_cmd_alloc(conn, &cmd);

    The result is a tuple containing the Sybase result code and a new instance of the CS_COMMAND class. None is returned as the CS_COMMAND object when the result code is not CS_SUCCEED.

def blk_alloc(version = BLK_VERSION_100):
    Allocates and returns a new CS_BLKDESC object which is used to perform bulkcopy over the connection. Calls the Sybase blk_alloc() function like this:

    status = blk_alloc(conn, version, &blk);

    The result is a tuple containing the Sybase result code and a new instance of the CS_BLKDESC class. None is returned as the CS_BLKDESC object when the result code is not CS_SUCCEED.

def ct_close(option = CS_UNUSED):
    Calls the Sybase ct_close() function like this:

    status = ct_close(conn, option);

    Returns the Sybase result code.

def ct_con_drop():
    Calls the Sybase ct_con_drop() function like this:
```

3.5 CS_CONNECTION Objects
status = ct_con_drop(conn);

Returns the Sybase result code.
This method will be automatically called when the CS_CONNECTION object is deleted. Applications do not need to call the method.

\texttt{ct\_con\_props} (\texttt{action, property [value]})

Sets, retrieves and clears properties of the connection object.

When \texttt{action} is \texttt{CS\_SET} a compatible value argument must be supplied and the method returns the Sybase result code. The Sybase-CT \texttt{ct\_con\_props()} function is called like this:

```c
/* boolean property value */
status = ct_con_props(conn, CS_SET, property, &bool_value, CS_UNUSED, NULL);

/* int property value */
status = ct_con_props(conn, CS_SET, property, &int_value, CS UNUSED, NULL);

/* string property value */
status = ct_con_props(conn, CS_SET, property, str_value, CS_NULLTERM, NULL);
```

When \texttt{action} is \texttt{CS\_GET} the method returns a tuple containing the Sybase result code and the property value. The Sybase-CT \texttt{ct\_con\_props()} function is called like this:

```c
/* boolean property value */
status = ct_con_props(conn, CS_GET, property, &bool_value, CS_UNUSED, NULL);

/* int property value */
status = ct_con_props(conn, CS_GET, property, &int_value, CS_UNUSED, NULL);

/* string property value */
status = ct_con_props(conn, CS_GET, property, str_buff, sizeof(str_buff), &buff_len);
```

When \texttt{action} is \texttt{CS\_CLEAR} the method returns the Sybase result code. The Sybase-CT \texttt{ct\_con\_props()} function is called like this:

```c
status = ct_con_props(conn, CS_CLEAR, property, NULL, CS_UNUSED, NULL);
```

The recognised properties are:

<table>
<thead>
<tr>
<th>property</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_ANSI_BINDS</td>
<td>bool</td>
</tr>
<tr>
<td>CS_ASYNC_NOTIFS</td>
<td>bool</td>
</tr>
<tr>
<td>CS_BULK_LOGIN</td>
<td>bool</td>
</tr>
<tr>
<td>CS_CHARSETCNV</td>
<td>bool</td>
</tr>
<tr>
<td>CS_CONFIG_BY_SERVERNAME</td>
<td>bool</td>
</tr>
<tr>
<td>CS_DIAG_TIMEOUT</td>
<td>bool</td>
</tr>
<tr>
<td>CS_DISABLE_POLL</td>
<td>bool</td>
</tr>
<tr>
<td>CS_DS_COPY</td>
<td>bool</td>
</tr>
<tr>
<td>CS_DS_EXPANDALIAS</td>
<td>bool</td>
</tr>
<tr>
<td>CS_DS_FAILOVER</td>
<td>bool</td>
</tr>
<tr>
<td>CS_EXPOSE_FMTS</td>
<td>bool</td>
</tr>
<tr>
<td>CS_EXTERNAL_CONFIG</td>
<td>bool</td>
</tr>
<tr>
<td>CS_EXTRA_INF</td>
<td>bool</td>
</tr>
<tr>
<td>CS_HIDDEN_KEYS</td>
<td>bool</td>
</tr>
<tr>
<td>CS_LOGIN_STATUS</td>
<td>bool</td>
</tr>
<tr>
<td>CS_NOCHARSETCNV_REQD</td>
<td>bool</td>
</tr>
<tr>
<td>CS_SEC_APPDEFINED</td>
<td>bool</td>
</tr>
<tr>
<td>CS_SEC_CHALLENGE</td>
<td>bool</td>
</tr>
</tbody>
</table>
For an explanation of the property values and get/set/clear semantics please refer to the Sybase documentation.

The following will allocate a connection from a library context, initialise the connection for in-line message handling, and connect to the named server using the specified username and password.

```python
def connect_db(ctx, server, user, passwd):
    status, conn = ctx.ct_con_alloc()
    if status != CS_SUCCEED:
        raise CSError(ctx, 'ct_con_alloc')
    if conn.ct_diag(CS_INIT) != CS_SUCCEED:
        raise CTError(conn, 'ct_diag')
    if conn.ct_con_props(CS_SET, CS_USERNAME, user) != CS_SUCCEED:
        raise CTError(conn, 'ct_con_props CS_USERNAME')
    if conn.ct_con_props(CS_SET, CS_PASSWORD, passwd) != CS_SUCCEED:
        raise CTError(conn, 'ct_con_props CS_PASSWORD')
    if conn.ct_connect(server) != CS_SUCCEED:
        raise CTError(conn, 'ct_connect')
    return conn
```
**ct_options**(action, option [., value])

Sets, retrieves and clears server query processing options for connection.

When `action` is `CS_SET` a compatible `value` argument must be supplied and the method returns the Sybase result code. The Sybase-CT `ct_options()` function is called like this:

```c
/* bool option value */
status = ct_options(conn, CS_SET, option, &bool_value, CS_UNUSED, NULL);
/* int option value */
status = ct_options(conn, CS_SET, option, &int_value, CS_UNUSED, NULL);
/* string option value */
status = ct_options(conn, CS_SET, option, str_value, CS_NULLTERM, NULL);
/* locale option value */
status = ct_options(conn, CS_SET, option, locale, CSUNUSED, NULL);
```

When `action` is `CS_GET` the method returns a tuple containing the Sybase result code and the option value. The Sybase-CT `ct_options()` function is called like this:

```c
/* bool option value */
status = ct_options(conn, CS_GET, option, &bool_value, CS_UNUSED, NULL);
/* int option value */
status = ct_options(conn, CS_GET, option, &int_value, CS_UNUSED, NULL);
/* string option value */
status = ct_options(conn, CS_GET, option, str_buff, sizeof(str_buff), &buff_len);
```

When `action` is `CS_CLEAR` the method returns the Sybase result code. The Sybase-CT `ct_options()` function is called like this:

```c
status = ct_options(conn, CS_CLEAR, option, NULL, CSUNUSED, NULL);
```

The recognised options are:

<table>
<thead>
<tr>
<th>option</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_OPT_ANSINULL</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_ANSIPERM</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_ARITHABORT</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_ARITHIGNORE</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_CHAINXACTS</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_CURCLOSEONXACT</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_FIPSFLAG</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_FORCEPLAN</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_FORMATONLY</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_GETDATA</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_NOCOUNT</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_NOEXEC</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_PARSEONLY</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_QUOTED_IDENT</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_RESTREES</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_SHOWPLAN</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_STATS_IO</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_STATS_TIME</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_STR_RTRUNC</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_TRUNCIGNORE</td>
<td>bool</td>
</tr>
<tr>
<td>CS_OPT_DATEFIRST</td>
<td>int</td>
</tr>
<tr>
<td>CS_OPT_DATEFORMAT</td>
<td>int</td>
</tr>
</tbody>
</table>
For an explanation of the option values and get/set/clear semantics please refer to the Sybase documentation.

### 3.6 CS_COMMAND Objects

Calling the `ct_cmd_alloc()` method of a `CS_CONNECTION` object will create a `CS_COMMAND` object. When the `CS_COMMAND` object is deallocated the Sybase `ct_cmd_drop()` function will be called for the command.

**CS_COMMAND** objects have the following interface:

**is_eed**
A read only integer which indicates when the `CS_COMMAND` object is an extended error data command structure.

**conn**
This is a read only reference to the parent `CS_CONNECTION` object. This prevents the connection from being dropped while the command still exists.

**strip**
An integer which controls right whitespace stripping of `char` columns. The default value is inherited from the parent connection when the command is created.

**debug**
An integer which controls printing of debug messages to the debug file established by the `set_debug()` function. The default value is inherited from the parent connection when the command is created.

**ct_bind(num, datafmt)**
Calls the Sybase-CT `ct_bind()` function and returns a tuple containing the Sybase result code and a `DataBuf` object which is used to retrieve data from the column identified by `num`. None is returned as the `DataBuf` object when the result code is not `CS_SUCCEED`. The Sybase-CT `ct_bind()` function is called like this:

```c
status = ct_bind(cmd, num, &datafmt, databuf->buff, databuf->copied, databuf->indicator);
```

See the description of the `ct_describe()` method for an example of how to use this method in Python.

**ct_cancel(type)**
Calls the Sybase `ct_cancel()` function like this:

```c
status = ct_cancel(NULL, cmd, type);
```

Returns the Sybase result code.

**ct_cmd_drop()**
Calls the Sybase-CT `ct_cmd_drop()` function like this:

```c
status = ct_cmd_drop(cmd);
```

Returns the Sybase result code.

This method will be automatically called when the `CS_COMMAND` object is deleted. Applications do not need to call the method.
ct_command(type[,...])

Calls the Sybase-CT ct_command() function and returns the result code. The type argument determines the type and number of additional arguments.

When type is CS_LANG_CMD the method must be called like this:
ct_command(CS_LANG_CMD, sql_text[, option = CS_UNUSED])

Then the Sybase-CT ct_command() function is called like this:

status = ct_command(cmd, CS_LANG_CMD, sql_text, CS_NULLTERM, option);

When type is CS_RPC_CMD the method must be called like this:
ct_command(CS_RPC_CMD, proc_name[, option = CS_UNUSED])

Then the Sybase-CT ct_command() function is called like this:

status = ct_command(cmd, CS_RPC_CMD, proc_name, CS_NULLTERM, option);

When type is CS_MSG_CMD the method must be called like this:
ct_command(CS_MSG_CMD, msg_num)

Then the Sybase-CT ct_command() function is called like this:

status = ct_command(cmd, CS_MSG_CMD, &msg_num, CS_UNUSED, CS_UNUSED);

When type is CS_PACKAGE_CMD the method must be called like this:
ct_command(CS_PACKAGE_CMD, pkg_name)

Then the Sybase-CT ct_command() function is called like this:

status = ct_command(cmd, CS_PACKAGE_CMD, pkg_name, CS_NULLTERM, CS_UNUSED);

When type is CS_SEND_DATA_CMD the method must be called like this:
ct_command(CS_SEND_DATA_CMD)

Then the Sybase-CT ct_command() function is called like this:

status = ct_command(cmd, CS_SEND_DATA_CMD, NULL, CS_UNUSED, CS_COLUMN_DATA);

For an explanation of the argument semantics please refer to the Sybase documentation.

cursor(type[,...])

Calls the Sybase ct_cursor() function and returns the result code. The type argument determines the type and number of additional arguments.

When type is CS_CURSOR_DECLARE the method must be called like this:
cursor(CS_CURSOR_DECLARE, cursor_id, sql_text[, option = CS_UNUSED])

Then the Sybase-CT ct_cursor() function is called like this:

status = ct_cursor(cmd, CS_CURSOR_DECLARE, cursor_id, CS_NULLTERM, sql_text, CS_NULLTERM, option);

When type is CS_CURSOR_UPDATE the method must be called like this:
cursor(CS_CURSOR_UPDATE, table_name, sql_text[, option = CS_UNUSED])

Then the Sybase-CT ct_cursor() function is called like this:

status = ct_cursor(cmd, CS_CURSOR_UPDATE, table_name, CS_NULLTERM, sql_text, CS_NULLTERM, option);

When type is CS_CURSOR_OPTION the method must be called like this:
cursor(CS_CURSOR_OPTION[, option = CS_UNUSED])
Then the Sybase-CT ct_cursor() function is called like this:

```c
cstatus = ct_cursor(cmd, CS_CURSOR_OPTION, NULL, CS_UNUSED, NULL, CS_UNUSED, option);
```

When `type` is `CS_CURSOR_OPEN` the method must be called like this:

```c
ct_cursor(CS_CURSOR_OPEN [, option = CS_UNUSED])
```

Then the Sybase-CT ct_cursor() function is called like this:

```c
cstatus = ct_cursor(cmd, CS_CURSOR_OPEN, NULL, CS_UNUSED, NULL, CS_UNUSED, option);
```

When `type` is `CS_CURSOR_CLOSE` the method must be called like this:

```c
ct_cursor(CS_CURSOR_CLOSE [, option = CS_UNUSED])
```

Then the Sybase-CT ct_cursor() function is called like this:

```c
cstatus = ct_cursor(cmd, CS_CURSOR_CLOSE, NULL, CS_UNUSED, NULL, CS_UNUSED, option);
```

When `type` is `CS_CURSOR_ROWS` the method must be called like this:

```c
ct_cursor(CS_CURSOR_ROWS, num_rows)
```

Then the Sybase-CT ct_cursor() function is called like this:

```c
cstatus = ct_cursor(cmd, CS_CURSOR_ROWS, NULL, CS_UNUSED, NULL, CS_UNUSED, num_rows);
```

When `type` is `CS_CURSOR_DELETE` the method must be called like this:

```c
ct_cursor(CS_CURSOR_DELETE, table_name)
```

Then the Sybase-CT ct_cursor() function is called like this:

```c
cstatus = ct_cursor(cmd, CS_CURSOR_DELETE, table_name, CS_NULLTERM, NULL, CS_UNUSED, CS_UNUSED);
```

For an explanation of the argument semantics please refer to the Sybase documentation.

The `cursor_sel.py`, `cursor_upd.py`, and `dynamic_cur.py` example programs contain examples of this function.

`ct_data_info(action, ...)`

Sets and retrieves column IO descriptors.

When `action` is `CS_SET` the method must be called like this:

```c
ct_data_info(CS_SET, iodesc)
```

Returns the Sybase result code. The Sybase-CT `ct_data_info()` function is called like this:

```c
status = ct_data_info(cmd, CS_SET, CS_UNUSED, &iodesc);
```

When `action` is `CS_GET` the method must be called like this:

```c
ct_data_info(CS_SET, num)
```

Returns a tuple containing the Sybase result code and an `CS_IODESC` object. If the result code is not `CS_SUCCEED` then `None` is returned as the `CS_IODESC` object. The Sybase-CT `ct_data_info()` function is called like this:

```c
status = ct_data_info(cmd, CS_SET, CS_UNUSED, &iodesc);
```
status = ct_data_info(cmd, CS_GET, num, &iodesc);

For an explanation of the argument semantics please refer to the Sybase documentation. The mult_text.py example program contains examples of this function.

**ct_describe (num)**

Calls the Sybase ct_describe() function and returns a tuple containing the Sybase result code and a CS_DATAFMT object which describes the column identified by num. None is returned as the CS_DATAFMT object when the result code is not CS_SUCCEED.

The Sybase-CT ct_describe() function is called like this:

```python
status = ct_describe(cmd, num, &datafmt);
```

The following constructs a list of buffers for retrieving a number of rows from a command object.

```python
def row_bind(cmd, count = 1):
    status, num_cols = cmd.ct_res_info(CS_NUMDATA)
    if status != CS_SUCCEED:
        raise 'ct_res_info'
    bufs = []
    for i in range(num_cols):
        status, fmt = cmd.ct_describe(i + 1)
        if status != CS_SUCCEED:
            raise 'ct_describe'
        fmt.count = count
        status, buf = cmd.ct_bind(i + 1, fmt)
        if status != CS_SUCCEED:
            raise 'ct_bind'
        bufs.append(buf)
    return bufs
```

**ct_dynamic (type, . . .)**

Calls the Sybase ct_dynamic() function and returns the result code. The type argument determines the type and number of additional arguments.

- **When type is CS_CURSOR_DECLARE** the method must be called like this:

```python
ct_dynamic(CS_CURSOR_DECLARE, dyn_id, cursor_id)
```

Then the Sybase-CT ct_dynamic() function is called like this:

```python
status = ct_dynamic(cmd, CS_CURSOR_DECLARE, dyn_id, CS_NULLTERM, cursor_id, CS_NULLTERM);
```

- **When type is CS_DEALLOC** the method must be called like this:

```python
call dynamic(CS_DEALLOC, dyn_id)
```

Then the Sybase-CT ct_dynamic() function is called like this:

```python
status = ct_dynamic(cmd, CS_DEALLOC, dyn_id, CS_NULLTERM, NULL, CS_UNUSED);
```

- **When type is CS_DESCRIBE_INPUT** the method must be called like this:

```python
ct_dynamic(CS_DESCRIBE_INPUT, dyn_id)
```

Then the Sybase-CT ct_dynamic() function is called like this:

```python
status = ct_dynamic(cmd, CS_DESCRIBE_INPUT, dyn_id, CS_NULLTERM, NULL, CS_UNUSED);
```

- **When type is CS_DESCRIBE_OUTPUT** the method must be called like this:

```python
ct_dynamic(CS_DESCRIBE_OUTPUT, dyn_id)
```

Then the Sybase-CT ct_dynamic() function is called like this:
status = ct_dynamic(cmd, CS_DESCRIBE_OUTPUT, dyn_id, CS_NULLTERM, NULL, CS_UNUSED);

When type is CS_EXECUTE the method must be called like this:
ct_dynamic(CS_EXECUTE, dyn_id)
Then the Sybase-CT ct_dynamic() function is called like this:

status = ct_dynamic(cmd, CS_EXECUTE, dyn_id, CS_NULLTERM, NULL, CS_UNUSED);

When type is CS_EXEC_IMMEDIATE the method must be called like this:
ct_dynamic(CS_EXEC_IMMEDIATE, sql_text)
Then the Sybase-CT ct_dynamic() function is called like this:

status = ct_dynamic(cmd, CS_EXEC_IMMEDIATE, NULL, CS_UNUSED, sql_text, CS_NULLTERM);

When type is CS_EXECUTE the method must be called like this:
ct_dynamic(CS_PREPARE, dyn_id, sql_text)
Then the Sybase-CT ct_dynamic() function is called like this:

status = ct_dynamic(cmd, CS_PREPARE, dyn_id, CS_NULLTERM, sql_text, CS_NULLTERM);

For an explanation of the argument semantics please refer to the Sybase documentation.
The dynamic_cur.py, and dynamic_ins.py example programs contain examples of this function.

**ct_fetch()**
Calls the Sybase ct_fetch() function and returns a tuple containing the Sybase result code and the number of rows read (for array binding).
The Sybase-CT ct_fetch() function is called like this:

status = ct_fetch(cmd, CS_UNUSED, CS_UNUSED, CS_UNUSED, &rows_read);

**ct_get_data(num, databuf)**
Calls the Sybase ct_get_data() function and returns a tuple containing the Sybase result code and the length of the data for item number num which was read into the DataBuf object in the databuf argument.
The Sybase-CT ct_get_data() function is called like this:

status = ct_get_data(cmd, num, databuf->buff, databuf->fmt.maxlength, databuf->copied);

The following will retrieve the contents of a BLOB column:

def get_blob_column(cmd, col):
    fmt = CS_DATAFMT()
    fmt.maxlength = 32768
    buf = DataBuf(fmt)
    parts = []
    while 1:
        status, count = cmd.ct_get_data(col, buf)
        if count:
            parts.append(buf[0])
        if status != CS_SUCCEED:
            break
    return string.join(parts, '')

**ct_param(param)**
Calls the Sybase ct_param() function and returns the Sybase result code.
The `param` argument is usually an instance of the `DataBuf` class. A `CS_DATAFMT` object can be used in a cursor declare context to define the format of the host variable.

When `param` is a `DataBuf` the Sybase-CT `ct_param()` function is called like this:

```c
status = ct_param(cmd, &databuf->fmt, databuf->buff, databuf->copied[0], databuf->indicator[0]);
```

When `param` is a `CS_DATAFMT` the Sybase-CT `ct_param()` function is called like this:

```c
status = ct_param(cmd, &datafmt, NULL, CS_UNUSED, CS_UNUSED);
```

The semantics of the `CS_DATAFMT` attributes are quite complex. Please refer to the Sybase documentation.

### `ct_res_info()`

Calls the Sybase `ct_res_info()` function. The return result depends upon the value of the `type` argument.

<table>
<thead>
<tr>
<th>type</th>
<th>return values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_BROWSE_INFO</td>
<td>status, bool</td>
</tr>
<tr>
<td>CS_CMD_NUMBER</td>
<td>status, int</td>
</tr>
<tr>
<td>CS_MSGTYPE</td>
<td>status, int</td>
</tr>
<tr>
<td>CS_NUM_COMPUTES</td>
<td>status, int</td>
</tr>
<tr>
<td>CS_NUMDATA</td>
<td>status, int</td>
</tr>
<tr>
<td>CS_NUMORDER_COLS</td>
<td>status, list of int</td>
</tr>
<tr>
<td>CS_ORDERBY_COLS</td>
<td>status, int</td>
</tr>
<tr>
<td>CS_ROW_COUNT</td>
<td>status, int</td>
</tr>
<tr>
<td>CS_TRANS_STATE</td>
<td>status, int</td>
</tr>
</tbody>
</table>

Depending on `type` the Sybase-CT `ct_res_info()` function is called like this:

```c
status = ct_res_info(cmd, CS_BROWSE_INFO, &bool_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_CMD_NUMBER, &int_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_MSGTYPE, &ushort_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_NUM_COMPUTES, &int_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_NUMDATA, &int_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_NUMORDER_COLS, &int_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_ROW_COUNT, &int_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_TRANS_STATE, &int_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_NUMORDER_COLS, &int_val, CS_UNUSED, NULL);
status = ct_res_info(cmd, CS_ORDERBY_COLS, col_nums, sizeof(*col_nums) * int_val, NULL);
```

### `ct_results()`

Calls the Sybase `ct_results()` function and returns a tuple containing the Sybase result code and the result type returned by the Sybase function.

The Sybase-CT `ct_results()` function is called like this:

```c
status = ct_results(cmd, &result);
```

### `ct_send()`

Calls the Sybase `ct_send()` function and returns the Sybase result code.

The Sybase-CT `ct_send()` function is called like this:
status = ct_send(cmd);

**ct_send_data** (*databuf*)

Calls the Sybase *ct_send_data()* function and returns the Sybase result code. The *databuf* argument must be a DataBuf object.

The Sybase-CT *ct_send_data()* function is called like this:

```c
status = ct_send_data(cmd, databuf->buff, databuf->copied[0]);
```

**ct_setparam** (*databuf*)

Calls the Sybase *ct_setparam()* function and returns the Sybase result code. The *databuf* argument must be a DataBuf object.

The Sybase-CT *ct_setparam()* function is called like this:

```c
status = ct_setparam(cmd, &databuf->fmt, databuf->buff, databuf->copied, databuf->indicator);
```

### 3.7 CS_CLIENTMSG Objects

CS_CLIENTMSG objects are a very thin wrapper around the Sybase CS_CLIENTMSG structure. They have the following read-only attributes:

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>severity</td>
<td>int</td>
</tr>
<tr>
<td>msgnumber</td>
<td>int</td>
</tr>
<tr>
<td>msgstring</td>
<td>string</td>
</tr>
<tr>
<td>osnumber</td>
<td>int</td>
</tr>
<tr>
<td>osstring</td>
<td>string</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
</tr>
<tr>
<td>sqlstate</td>
<td>string</td>
</tr>
</tbody>
</table>

### 3.8 CS_SERVERMSG Objects

CS_SERVERMSG objects are a very thin wrapper around the Sybase CS_SERVERMSG structure. They have the following read-only attributes:

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>msgnumber</td>
<td>int</td>
</tr>
<tr>
<td>state</td>
<td>int</td>
</tr>
<tr>
<td>severity</td>
<td>int</td>
</tr>
<tr>
<td>text</td>
<td>string</td>
</tr>
<tr>
<td>svrname</td>
<td>string</td>
</tr>
<tr>
<td>proc</td>
<td>string</td>
</tr>
<tr>
<td>line</td>
<td>int</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
</tr>
<tr>
<td>sqlstate</td>
<td>string</td>
</tr>
</tbody>
</table>

### 3.9 CS_DATAFMT Objects

CS_DATAFMT objects are a very thin wrapper around the Sybase CS_DATAFMT structure. They have the following attributes:
The `strip` attribute is an extension of the Sybase CS_DATAFMT structure. Please refer to the DataBuf documentation.

CS_DATAFMT structures are mostly used to create DataBuf objects for sending data to and receiving data from the server.

A CS_DATAFMT object created via the CS_DATAFMT() constructor will have the following values:

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>'\0'</td>
</tr>
<tr>
<td>datatype</td>
<td>CS_CHAR_TYPE</td>
</tr>
<tr>
<td>format</td>
<td>CS_FMT_NULLTERM</td>
</tr>
<tr>
<td>maxlength</td>
<td>1</td>
</tr>
<tr>
<td>scale</td>
<td>0</td>
</tr>
<tr>
<td>precision</td>
<td>0</td>
</tr>
<tr>
<td>status</td>
<td>0</td>
</tr>
<tr>
<td>count</td>
<td>0</td>
</tr>
<tr>
<td>usertype</td>
<td>0</td>
</tr>
<tr>
<td>strip</td>
<td>0</td>
</tr>
</tbody>
</table>

You will almost certainly need to provide new values for some of the attributes before you use the object.

A CS_DATAFMT object created as a return value from the ct_bind() function will be ready to use for creating a DataBuf object.

### 3.10 DataBuf Objects

DataBuf objects manage buffers which are used to hold data to be sent to and received from the server.

DataBuf objects contain an embedded Sybase CS_DATAFMT structure and allocated buffers suitable for binding the contained data to Sybase-CT API functions.

When constructed from native Python or Sybase data types a buffer is created for a single value. When created using a CS_DATAFMT object the `count` attribute is used to allocate buffers suitable for array binding. A `count` of zero is treated the same as 1.

The DataBuf objects have the same attributes as a CS_DATAFMT object but the attributes which describe the memory are read only and cannot be modified.

<table>
<thead>
<tr>
<th>attribute</th>
<th>type</th>
<th>read only?</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>string</td>
<td>no</td>
</tr>
<tr>
<td>datatype</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>format</td>
<td>int</td>
<td>no</td>
</tr>
<tr>
<td>maxlength</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>scale</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>precision</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>no</td>
</tr>
<tr>
<td>count</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>usertype</td>
<td>int</td>
<td>yes</td>
</tr>
<tr>
<td>strip</td>
<td>int</td>
<td>no</td>
</tr>
</tbody>
</table>
In addition the `DataBuf` object behaves like a fixed length mutable sequence.

Adapted from `Sybase.py`, this is how you create a set of buffers suitable for retrieving a number of rows from the server:

```python
def row_bind(cmd, count = 1):
    status, num_cols = cmd.ct_res_info(CS_NUMDATA)
    if status != CS_SUCCEED:
        raise 'ct_res_info'
    bufs = []
    for i in range(num_cols):
        status, fmt = cmd.ct_describe(i + 1)
        if status != CS_SUCCEED:
            raise 'ct_describe'
        fmt.count = count
        status, buf = cmd.ct_bind(i + 1, fmt)
        if status != CS_SUCCEED:
            raise 'ct_bind'
        bufs.append(buf)
    return bufs
```

Then once the rows have been fetched, this is how you extract the data from the buffers:

```python
def fetch_rows(cmd, bufs):
    rows = []
    status, rows_read = cmd.ct_fetch()
    if status == CS_SUCCEED:
        for i in range(rows_read):
            row = []
            for buf in bufs:
                row.append(buf[i])
            rows.append(tuple(row))
    return rows
```

To send a parameter to a dynamic SQL command or a stored procedure you are likely to create a `DataBuf` object directly from the value you wish to send. For example:

```python
if cmd.ct_command(CS_RPC_CMD, 'sp_help', CS_NO_RECOMPILE) != CS_SUCCEED:
    raise 'ct_command'
buf = DataBuf('sysobjects')
buf.status = CS_INPUTVALUE
if cmd.ct_param(buf) != CS_SUCCEED:
    raise 'ct_param'
if cmd.ct_send() != CS_SUCCEED:
    raise 'ct_send'
```

Note that it is your responsibility to make sure that the buffers are not deallocated before you have finished using them. If you are not careful you will get a segmentation fault.

### 3.11 CS_IODESC Objects

`CS_IODESC` objects are a very thing wrapper around the Sybase `CS_IODESC` structure. They have the following attributes:
These objects are created either by calling the `ct_data_info()` method of a `CS_COMMAND` object, or by calling the `CS_IODESC` constructor.

### 3.12 CS_BlkDESC Objects

Calling the `blk_alloc()` method of a `CS_CONNECTION` object will create a `CS_BlkDESC` object. When the `CS_BlkDESC` object is deallocated the Sybase `blk_drop()` function will be called for the command.

**CS_BlkDESC** objects have the following interface:

**blk_bind**(num, databuf)

Calls the Sybase `blk_bind()` function and returns the Sybase result code. The Sybase-CT `blk_bind()` function is called like this:

```c
status = blk_bind(blk, num, &datafmt, buffer->buff, buffer->copied, buffer->indicator);
```

**blk_describe**(num)

Calls the Sybase `blk_describe()` function and returns a tuple containing the Sybase result code and a `CS_DATAFMT` object which describes the column identified by num. None is returned as the `CS_DATAFMT` object when the result code is not `CS_SUCCEED`.

The Sybase `blk_describe()` function is called like this:

```c
status = blk_describe(blk, num, &datafmt);
```

**blk_done**(type)

Calls the Sybase `blk_done()` function and returns a tuple containing the Sybase result code and the number of rows copied in the current batch.

The Sybase `blk_done()` function is called like this:

```c
status = blk_done(blk, type, &num_rows);
```

**blk_drop**()

Calls the Sybase `blk_drop()` function and returns the Sybase result code.

The Sybase `blk_drop()` function is called like this:

```c
status = blk_drop(blk);
```

This method will be automatically called when the `CS_BlkDESC` object is deleted. Applications do not need to call the method.

**blk_init**(direction, table)

Calls the Sybase `blk_init()` function and returns the Sybase result code.

The Sybase `blk_init()` function is called like this:

```c
status = blk_init(blk, direction, table, CS_NULLTERM);
```
**blk_props** *(action, property [, value]*)

Sets, retrieves and clears properties of the bulk descriptor object.

When *action* is **CS_SET** a compatible *value* argument must be supplied and the method returns the Sybase result code. The Sybase `blk_props()` function is called like this:

```c
/* boolean property value */
status = blk_props(blk, CS_SET, property, &bool_value, CS_UNUSED, NULL);

/* int property value */
status = blk_props(blk, CS_SET, property, &int_value, CS_UNUSED, NULL);

/* numeric property value */
status = blk_props(blk, CS_SET, property, &numeric_value, CS_UNUSED, NULL);
```

When *action* is **CS_GET** the method returns a tuple containing the Sybase result code and the property value. The Sybase `blk_props()` function is called like this:

```c
/* boolean property value */
status = blk_props(blk, CS_GET, property, &bool_value, CS_UNUSED, NULL);

/* int property value */
status = blk_props(blk, CS_GET, property, &int_value, CS_UNUSED, NULL);

/* numeric property value */
status = blk_props(blk, CS_GET, property, &numeric_value, CS_UNUSED, NULL);
```

When *action* is **CS_CLEAR** the method returns the Sybase result code. The Sybase `blk_props()` function is called like this:

```c
status = blk_props(blk, CS_CLEAR, property, NULL, CS_UNUSED, NULL);
```

The recognised properties are:

<table>
<thead>
<tr>
<th>property</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLK.IDENTITY</td>
<td>bool</td>
</tr>
<tr>
<td>BLK_NOAPI_CHK</td>
<td>bool</td>
</tr>
<tr>
<td>BLK_SENSITIVITY_LBL</td>
<td>bool</td>
</tr>
<tr>
<td>ARRAY_INSERT</td>
<td>bool</td>
</tr>
<tr>
<td>BLK_SLICENUM</td>
<td>int</td>
</tr>
<tr>
<td>BLK_IDSTARTNUM</td>
<td>numeric</td>
</tr>
</tbody>
</table>

For an explanation of the property values and get/set/clear semantics please refer to the Sybase documentation.

**blk_rowxfer()**

Calls the Sybase `blk_rowxfer()` function and returns the Sybase result code.

The Sybase `blk_rowxfer()` function is called like this:

```c
status = blk_rowxfer(blk);
```

**blk_rowxfer_mult ([row_count])**

Calls the Sybase `blk_rowxfer_mult()` function and returns a tuple containing the Sybase result code and the number of rows transferred.

The Sybase `blk_rowxfer_mult()` function is called like this:

```c
status = blk_rowxfer_mult(blk, &row_count);
```
blk_textxfer([str])

Calls the Sybase blk_textxfer() function. Depending on the direction of the bulkcopy (established via the blk_init() method) the method expects different arguments.

When direction CS_BLK_IN the str argument must be supplied and method returns the Sybase result code.

The Sybase blk_textxfer() function is called like this:

```
status = blk_textxfer(blk, str, str_len, NULL);
```

When direction CS_BLK_OUT the str argument must not be present and method returns a tuple containing the Sybase result code and a string.

The Sybase blk_textxfer() function is called like this:

```
status = blk_textxfer(blk, buff, sizeof(buff), &out_len);
```

A simplistic program to bulkcopy a table from one server to another server follows:

The first section contains the code to display client and server messages in case of failure.

```
import sys
from sybasect import *

def print_msgs(conn, type):
    status, num_msgs = conn.ct_diag(CS_STATUS, type)
    if status != CS_SUCCEED:
        return
    for i in range(num_msgs):
        status, msg = conn.ct_diag(CS_GET, type, i + 1)
        if status != CS_SUCCEED:
            continue
        for attr in dir(msg):
            sys.stderr.write('%s: %s
' % (attr, getattr(msg, attr))

def die(conn, func):
    sys.stderr.write('%s failed!
' % func)
    print_msgs(conn, CS_SERVERMSG_TYPE)
    print_msgs(conn, CS_CLIENTMSG_TYPE)
    sys.exit(1)
```

The next section is fairly constant for all CT library programs. A library context is allocated and connections established. The only thing which is unique to bulk copy operations is setting the CS_BULK_LOGIN option on the connection.
def init_db():
    status, ctx = cs_ctx_alloc()
    if status != CS_SUCCEED:
        raise 'cs_ctx_alloc'
    if ctx.ct_init(CS_VERSION_100) != CS_SUCCEED:
        raise 'ct_init'
    return ctx

def connect_db(ctx, server, user, passwd):
    status, conn = ctx.ct_con_alloc()
    if status != CS_SUCCEED:
        raise 'ct_con_alloc'
    if conn.ct_diag(CS_INIT) != CS_SUCCEED:
        die(conn, 'ct_diag')
    if conn.ct_con_props(CS_SET, CS_USERNAME, user) != CS_SUCCEED:
        die(conn, 'ct_con_props CS_USERNAME')
    if conn.ct_con_props(CS_SET, CS_PASSWORD, passwd) != CS_SUCCEED:
        die(conn, 'ct_con_props CS_PASSWORD')
    if conn.ct_con_props(CS_SET, CS_BULK_LOGIN, 1) != CS_SUCCEED:
        die(conn, 'ct_con_props CS_BULK_LOGIN')
    if conn.ct_connect(server) != CS_SUCCEED:
        die(conn, 'ct_connect')
    return conn

The next segment allocates bulkcopy descriptors, data buffers, and binds the data buffers to the bulk copy descriptors. The same buffers are used for copying out and copying in - not bad. Note that for array binding we need to use loose packing for copy in; hence the line setting the format member of Databuf CS_DATAFMT to CS_BLK_ARRAY_MAXLEN. Without this the bulkcopy operation assumes tight packing and the data is corrupted on input.

def alloc_bcp(conn, dirn, table):
    status, blk = conn.blk_alloc()
    if status != CS_SUCCEED:
        die(conn, 'blk_alloc')
    if blk.blk_init(dirn, table) != CS_SUCCEED:
        die(conn, 'blk_init')
    return blk

def alloc_bufs(bcp, num):
    bufs = []
    while 1:
        status, fmt = bcp.blk_describe(len(bufs) + 1)
        if status != CS_SUCCEED:
            break
        fmt.count = num
        bufs.append(DataBuf(fmt))
    return bufs

def bcp_bind(bcp, bufs):
    for i in range(len(bufs)):
        buf = bufs[i]
        if bcp.direction == CS_BLK_OUT:
            buf.format = 0
        else:
            buf.format = CS_BLK_ARRAY_MAXLEN
        if bcp.blk_bind(i + 1, buf) != CS_SUCCEED:
            die(bcp.conn, 'blk_bind')

This next section actually performs the bulkcopy. Note that there is no attempt to deal with BLOB columns.
def bcp_copy(src, dst, batch_size):
    total = batch = 0
    while 1:
        status, num_rows = src.blk_rowxfer_mult()
        if status == CS_END_DATA:
            break
        if status != CS_SUCCEED:
            die(src, 'blk_rowxfer_mult out')
        status, dummy = dst.blk_rowxfer_mult(num_rows)
        if status != CS_SUCCEED:
            die(src, 'blk_rowxfer_mult in')
        batch = batch + num_rows
        if batch >= batch_size:
            total = total + batch
            batch = 0
        src.blk_done(CS_BLK_BATCH)
        dst.blk_done(CS_BLK_BATCH)
    print 'batch - %d rows transferred' % total
    status, num_rows = src.blk_done(CS_BLK_ALL)
    status, num_rows = dst.blk_done(CS_BLK_ALL)
    return total + batch

Finally the code which drives the whole process.

ctx = init_db()
src_conn = connect_db(ctx, 'drama', 'sa', '')
dst_conn = connect_db(ctx, 'SYBASE', 'sa', '')
src = alloc_bcp(src_conn, CS_BLK_OUT, 'pubs2.dbo.authors')
dst = alloc_bcp(dst_conn, CS_BLK_IN, 'test.dbo.authors')

bufs = alloc_bufs(src, 5)
bcp_bind(src, bufs)
bcp_bind(dst, bufs)

total = bcp_copy(src, dst, 10)
print 'all done - %d rows transferred' % total
Index

Symbols
__iter__() (Bulkcopy method), 12
__version__ (data in Sybase), 4

A
apilevel (data in Sybase), 4
arraysize() (Bulkcopy method), 12
arraysize (Cursor attribute), 11

B
batch() (Bulkcopy method), 12
begin() (Connection method), 9
BINARY (data in Sybase), 6
Binary() (in module Sybase), 6

C
callproc() (Cursor method), 10
ClientMsgType (data in sybasect), 13
close()
  Connection method, 8
  Cursor method, 10
CommandType (data in sybasect), 13
commit() (Connection method), 8
conn (CS_COMMAND attribute), 27
connect()
  Connection method, 9
  in module Sybase, 7
Connection (class in Sybase), 7
ConnectionType (data in sybasect), 13
ContextType (data in sybasect), 12
cs_config() (CS_CONTEXT method), 15
cs_ctx_alloc() (in module sybasect), 13
cs_ctx_drop() (CS_CONTEXT method), 19
cs_ctx_global() (in module sybasect), 13
CS_DATAFMT() (in module sybasect), 14
cs_diag() (CS_CONTEXT method), 19
cs_dt_info() (CS_LOCALE method), 20
CS_IODESC() (in module sybasect), 15
CS_LAYER() (in module sybasect), 15
cs_loc_alloc() (CS_CONTEXT method), 17
cs_loc_drop() (CS_LOCALE method), 21
cs_locale() (CS_LOCALE method), 21
CS_NUMBER() (in module sybasect), 15
CS_ORIGIN() (in module sybasect), 15
CS_SEVERITY() (in module sybasect), 15
c_t_bind() (CS_COMMAND method), 27
c_t_callback() (CS_CONTEXT method), 16
c_t_cancel()
  CS_COMMAND method, 27
  CS_CONNECTION method, 23
c_t_close() (CS_CONNECTION method), 23
c_t_cmd_alloc() (CS_CONNECTION method), 23
c_t_cmd_drop() (CS_COMMAND method), 27
c_t_command() (CS_COMMAND method), 28
c_t_con_alloc() (CS_CONTEXT method), 17
c_t_con_drop() (CS_CONNECTION method), 23
c_t_con_props() (CS_CONNECTION method), 24
c_t_config() (CS_CONTEXT method), 17
c_t_connect() (CS_CONNECTION method), 23
c_t_cursor() (CS_COMMAND method), 28
c_t_data_info() (CS_COMMAND method), 29
c_t_describe() (CS_COMMAND method), 30
c_t_diag() (CS_COMMAND method), 22
c_t_dynamic() (CS_COMMAND method), 30
c_t_exit() (CS_CONTEXT method), 19
c_t_fetch() (CS_COMMAND method), 31
c_t_get_data() (CS_COMMAND method), 31
c_t_init() (CS_CONTEXT method), 19
c_t_options() (CS_CONNECTION method), 26
c_t_param() (CS_COMMAND method), 31
c_t_res_info() (CS_COMMAND method), 32
c_t_results() (CS_COMMAND method), 32
c_t_send() (CS_COMMAND method), 32
c_t_send_data() (CS_COMMAND method), 33
c_t_setparam() (CS_COMMAND method), 33
tx (CS_CONNECTION attribute), 22
Cursor (class in Sybase), 6
cursor() (Connection method), 9

d
DatabaseError (exception in Sybase), 5
DataBuf() (in module sybasect), 14
DataBufType (data in sybasect), 13
DataError (exception in Sybase), 5
DataFmtType (data in sybasect), 13
date() (in module Sybase), 6
dateFromTicks() (in module Sybase), 6
DATETIME (data in Sybase), 6
datetime()
  in module Sybase, 8
  in module sybasect, 14
datetimeType (data in sybasect), 13
debug
CS_COMMAND attribute, 27
CS_CONNECTION attribute, 22
CS_CONTEXT attribute, 15
debug_msg() (CS_CONTEXT method), 15
description (Cursor attribute), 10
done() (Bulkcopy method), 12

E
Error (exception in Sybase), 5
execute()
    Connection method, 9
    Cursor method, 10
executemany() (Cursor method), 11

F
fetchall() (Cursor method), 11
fetchmany() (Cursor method), 11
fetchone() (Cursor method), 11

G
get_property() (Connection method), 9

I
IntegrityError (exception in Sybase), 5
InterfaceError (exception in Sybase), 5
InternalError (exception in Sybase), 5
IODescType (data in sybasect), 13
is__eed (CS_COMMAND attribute), 27

M
money()
    in module Sybase, 8
    in module sybasect, 14
MoneyType (data in sybasect), 13

N
nextset() (Cursor method), 11
NotSupportedError (exception in Sybase), 5
NUMBER (data in Sybase), 6
numeric()
    in module Sybase, 8
    in module sybasect, 14
NumericType (data in sybasect), 13

O
OperationalError (exception in Sybase), 5

P
paramstyle (data in Sybase), 5
ProgrammingError (exception in Sybase), 5

R
rollback() (Connection method), 9
rowcount (Cursor attribute), 10
ROWID (data in Sybase), 6
rows() (Bulkcopy method), 12
rowxfer() (Bulkcopy method), 11

S
ServerMsgType (data in sybasect), 13
set_debug() (module sybasect), 13
set_global_ctx() (module sybasect), 13
set_property() (Connection method), 9
setinputsizes() (Cursor method), 11
setoutputsize() (Cursor method), 11
sizeof_type() (module sybasect), 14
STRING (data in Sybase), 6
strip
    CS_COMMAND attribute, 27
    CS_CONNECTION attribute, 22
Sybase (standard module), 4
sybasect (extension module), 12

T
threadsafety (data in Sybase), 4
Time() (module Sybase), 6
TimeFromTicks() (module Sybase), 6
Timestamp() (module Sybase), 6
TimestampFromTicks() (module Sybase), 6
totalcount() (Bulkcopy method), 12

U
use_datetime (data in Sybase), 4

W
Warning (exception in Sybase), 5