pymonetdb Documentation

Release 1.7.1

Gijs Molenaar

Sep 22, 2023

Contents

pymonetdb is the native Python client API for monetDB. It is cross-platform and does not depend on any MonetDB libraries. It supports Python 3.6+ and PyPy and is Python DBAPI 2.0 compatible.

Besides the functionality required by DBAPI 2.0, pymonetdb also provides some MonetDB-specific functionality, in particular file transfers. These are detailed in the API section.

CHAPTER 1

Contents

1.1 Getting Started

1.1.1 Installation

pymonetdb is available on PyPI and can be installed with the following command:

```
$ pip install pymonetdb
```

It can also be installed from its source directory by running:

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

1.1.2 Connecting

In its simplest form, the function *pymonetdb.connect()* takes a single parameter, the database name:

```
conn = pymonetdb.connect('demo')
```

Usually, you have to pass more:

```
conn = pymonetdb.connect(
  'demo',
  hostname='dbhost', port=50001,
  username='yours', password='truly')
```

There are also some options you can set, for example `autocommit=True`.

It is also possible to combine everything in a URL:

```
url = 'mapi:monetdb://yours:truly@dbhost:50001/demo?autocommit=true'
conn = pymonetdb.connect(url)
```

For more details see the documentation of *pymonetdb.connect()*.

1.2 Examples

Here are some examples of how to use pymonetdb.

1.2.1 Example session

```
> # import the SQL module
> import pymonetdb
>
> # set up a connection. arguments below are the defaults
> connection = pymonetdb.connect(username="monetdb", password="monetdb",
>                                hostname="localhost", database="demo")
>
> # create a cursor
> cursor = connection.cursor()
>
> # increase the rows fetched to increase performance (optional)
> cursor.arraysize = 100
>
> # execute a query (return the number of rows to fetch)
> cursor.execute('SELECT * FROM tables')
26
>
> # fetch only one row
> cursor.fetchone()
[1062, 'schemas', 1061, None, 0, True, 0, 0]
>
> # fetch the remaining rows
> cursor.fetchall()
[[1067, 'types', 1061, None, 0, True, 0, 0],
 [1076, 'functions', 1061, None, 0, True, 0, 0],
 [1085, 'args', 1061, None, 0, True, 0, 0],
 [1093, 'sequences', 1061, None, 0, True, 0, 0],
 [1103, 'dependencies', 1061, None, 0, True, 0, 0],
 [1107, 'connections', 1061, None, 0, True, 0, 0],
 [1116, '_tables', 1061, None, 0, True, 0, 0],
 ...
 [4141, 'user_role', 1061, None, 0, True, 0, 0],
 [4144, 'auths', 1061, None, 0, True, 0, 0],
 [4148, 'privileges', 1061, None, 0, True, 0, 0]]
>
> # Show the table meta data
> cursor.description
[('id', 'int', 4, 4, None, None, None),
 ('name', 'varchar', 12, 12, None, None, None),
 ('schema_id', 'int', 4, 4, None, None, None),
 ('query', 'varchar', 168, 168, None, None, None),
 ('type', 'smallint', 1, 1, None, None, None),
 ('system', 'boolean', 5, 5, None, None, None),
 ('commit_action', 'smallint', 1, 1, None, None, None),
 ('temporary', 'tinyint', 1, 1, None, None, None)]
```

1.2.2 MAPI Connection

If you would like to communicate with the database at a lower level you can use the MAPI library (but not recommended):

```
> from pymonetdb import mapi
> server = mapi.Connection()
> server.connect(hostname="localhost", port=50000, username="monetdb",
                 password="monetdb", database="demo", language="sql")
> server.cmd("sSELECT * FROM tables;")
...
```

1.2.3 CSV Upload

This is an example script that uploads some CSV data from the local file system:

```
#!/usr/bin/env python3

import os
import pymonetdb

# Create the data directory and the CSV file
try:
    os.mkdir("datadir")
except FileExistsError:
    pass
with open("datadir/data.csv", "w") as f:
    for i in range(10):
        print(f"{i},item{i + 1}", file=f)

# Connect to MonetDB and register the upload handler
conn = pymonetdb.connect('demo')
handler = pymonetdb.SafeDirectoryHandler("datadir")
conn.set_uploader(handler)
cursor = conn.cursor()

# Set up the table
cursor.execute("DROP TABLE foo")
cursor.execute("CREATE TABLE foo(i INT, t TEXT)")

# Upload the data, this will ask the handler to upload data.csv
cursor.execute("COPY INTO foo FROM 'data.csv' ON CLIENT USING DELIMITERS ','")

# Check that it has loaded
cursor.execute("SELECT t FROM foo WHERE i = 9")
row = cursor.fetchone()
assert row[0] == 'item10'

# Goodbye
conn.commit()
cursor.close()
conn.close()
```

1.3 File Transfers

MonetDB supports the non-standard `COPY INTO` statement to load a CSV-like text file into a table or to dump a table into a text file. This statement has an optional modifier `ON CLIENT` to indicate that the server should not try to open the file on the server side but instead ask the client to open it on its behalf.

For example:

```
COPY INTO mytable FROM 'data.csv' ON CLIENT
USING DELIMITERS ',', E'\n', '"';
```

However, by default, if pymonetdb receives a file request from the server, it will refuse it for security considerations. You do not want an unauthorised party pretending to be the server to be able to request arbitrary files on your system and even overwrite them.

To enable file transfers, create a *pymonetdb.Uploader* or *pymonetdb.Downloader* and register them with your connection:

```
transfer_handler = pymonetdb.SafeDirectoryHandler(datadir)
conn.set_uploader(transfer_handler)
conn.set_downloader(transfer_handler)
```

With this in place, the `COPY INTO ... ON CLIENT` statement above will cause pymonetdb to open the file *data.csv* in the given *datadir* and upload its contents. As its name suggests, `SafeDirectoryHandler` will only allow access to the files in that directory.

Note that in this example, we register the same handler object as an uploader and a downloader for demonstration purposes. In the real world, it is good security practice only to register an uploader or a downloader It is also possible to use two separate handlers.

See the API documentation for details.

1.3.1 Make up data as you go

You can also write your own transfer handlers. And instead of opening a file, such handlers can make up the data on the fly, for instance, retrieve it from a remote microservice, prompt the user interactively or do whatever else you come up with:

```
class MyUploader(pymonetdb.Uploader):
    def handle_upload(self, upload, filename, text_mode, skip_amount):
        tw = upload.text_writer()
        for i in range(skip_amount, 1000):
            print(f'{i},number{i}', file=tw)
```

In this example, we call *upload.text_writer()* to yield a text-mode file-like object. There is also an *upload.binary_writer()*, which creates a binary-mode file-like object. The *binary_writer()* works even if the server requests a text mode object, but in that case, you have to make sure the bytes you write are valid UTF-8 and delimited with Unix line endings rather than Windows line endings.

If you want to refuse an upload or download, call *upload.send_error()* to send an error message *before* any call to *text_writer()* or *binary_writer()*.

For custom downloaders, the situation is similar, except that instead of *text_writer* and *binary_writer*, the *download* parameter offers *download.text_reader()* and *download.text_writer()*.

1.3.2 Skip amount

MonetDB's `COPY INTO` statement allows you to skip, for example, the first line in a file using the modifier `OFFSET 2`. In such a case, the *skip_amount* parameter to *handle_upload()* will be greater than zero.

Note that the offset in the SQL statement is 1-based, whereas the *skip_amount* parameter has already been converted to 0-based. The example above thus allows us to write `for i in range(skip_amount, 1000):` rather than `for i in range(1000):`.

1.3.3 Cancellation

In cases depicted by the following query, the server does not need to receive all data of the input file:

COPY 100 RECORDS INTO mytable FROM 'data.csv' ON CLIENT

Therefore, pymonetdb regularly asks the server if it is still interested in receiving more data. In this way, the server can cancel the uploading after it has received sufficient data to process the query. By default, pymonetdb does this after every MiB of data, but you can change this frequency using *upload.set_chunk_size()*.

If the server answers that it is no longer interested, pymonetdb will discard any further data written to the writer. It is recommended to call *upload.is_cancelled()* occasionally to check for this and exit early if the upload has been cancelled.

Upload handlers also have an optional method *cancel()* that you can override. This method is called when pymonetdb receives the cancellation request.

1.3.4 Copying data from or to a file-like object

If you are moving large amounts of data between pymonetdb and a file-like object such as a file, Python's copyfileobj function may come in handy:

```
class MyUploader(pymonetdb.Uploader):
    def __init__(self, dir):
        self.dir = pathlib.Path(dir)

    def handle_upload(self, upload, filename, text_mode, skip_amount):
        # security check
        path = self.dir.joinpath(filename).resolve()
        if not str(path).startswith(str(self.dir.resolve())):
            return upload.send_error('Forbidden')
        # open
        tw = upload.text_writer()
        with open(path) as f:
            # skip
            for i in range(skip_amount):
                f.readline()
            # bulk upload
            shutil.copyfileobj(f, tw)
```

However, note that copyfileobj does not handle cancellations as described above.

1.3.5 Security considerations

If your handler accesses the file system or the network, it is critical to validate the file name you are given carefully. Otherwise, an attacker can take over the server or the connection to the server and cause great damage.

The code sample above also includes an example of validating file systems paths. Similar considerations apply to text inserted into network URLs and other resource identifiers.

1.4 Result set batch size

When a query produces a large result set, pymonetdb will often only retrieve part of the result set, retrieving the rest later, one batch at a time. The default behavior is to start with a reasonably small batch size but increase it rapidly. However, if necessary, the application can configure this behavior. In the table below, you can see the settings controlling the behavior of large transfers.

Setting name	Defined by	Range	Default
replysize	pymonetdb	positive integer or -1 [*]	100
maxprefetch	pymonetdb	positive integer or -1 [*]	2500
arraysize	DBAPI 2.0	positive integer	*Connection.replysize*

[*] The value -1 means unlimited.

The *replysize* and *maxprefetch* settings can be set as attributes of both *Connection* and *Cursor*. They can also be passed as parameters in the connection URL. The *arraysize* setting only exists for *Cursor*. It defaults to the *replysize* of the connection when the cursor was created if that is positive, or 100 otherwise.

1.4.1 Batching behavior

When MonetDB has finished executing a query, the server includes the first rows of the result set in its response to *Cursor.execute()*. The exact number of rows it includes can be configured using the *replysize* setting.

How the rest of the rows are retrieved depends on how they are accessed. *Cursor.fetchone()* and *Cursor.fetchmany()* retrieve the remaining rows in batches of increasing size. Every batch is twice as large as the previous one until the prefetch limit *maxprefetch* has been reached. This setting controls the maximum number of fetched rows that are not immediately used.

For instance, with *replysize = 100*, the first 100 *fetchone()* calls immediately return the next row from the cache. For the 101-st *fetchone()*, pymonetdb will first double the *replysize* and retrieve rows 101-300 before returning row 101. When *Cursor.fetchmany()* is used, pymonetdb also adjusts the *replysize* to the requested stride. For example, for *fetchmany(40)*, the first two calls will return rows from the cache. However, for the third call, pymonetdb will first retrieve rows 101-320, i.e. double the *replysize* and enlarge it to reach a multiple of 40, before returning rows 81 - 120.

With *Cursor.fetchall()*, all rows are retrieved at once.

1.4.2 New result set format

Version Jun2023 of MonetDB introduces a new, binary result set format that is much more efficient to parse. The initial transfer of *replysize* rows still uses the existing text-based format; however, the subsequent batches can be transferred much more efficiently with the binary format. By default, pymonetdb will automatically use it when possible unless configured otherwise using the *binary* setting, e.g. *pymonetdb.connect('demo', binary=0)* or *pymonetdb.connect('mapi:monetdb://localhost/demo?binary=0')*.

Normally, the binary result set transfer is transparent to the user applications. The result set fetching functions automatically do the necessary data conversion. However, if you want to know explicitly if the binary format has been used, you can use *Cursor.used_binary_protocol()*, e.g. after having called a fetch function.

We have implemented a special case to benefit from the binary protocol even when the *replysize* is set to -1. When pymonetdb knows that binary transfers are possible (e.g. learnt when connecting with MoentDB) while *replysize* is -1,

it overrides the *replysize*. Pymonetdb will use a small size for the initial transfer and then retrieve the rest of the result set in one large binary batch.

1.4.3 Tweaking the behavior

Usually, the batching behavior does not need to be tweaked.

When deciding which function to use to fetch the result sets, *Cursor.fetchmany()* seems to be a few percent more efficient than *Cursor.fetchall()*, while *Cursor.fetchone()* tends to be 10-15% slower.

To reduce the amount of prefetched data, set *maxprefetch* to a lower value or even 0. The value 0 disables prefetch entirely, only fetching the requested rows. Setting *maxprefetch* to -1 has the opposite effect: it allows the prefetch size to increase without a bound.

If you expect the size of the individual rows to be huge, consider setting both *replysize* and *maxprefetch* to small values, for example, 10 and 20, respectively, or even 1 and 0. These small batch sizes limit the memory each batch consumes. As a quick rule of thumb for the memory requirements, one can assume that pymonetdb may need up to three times the size of the result set. Also, remember that if MonetDB is running on the same host, the server will also need at least that amount of memory.

Generally, one does not need to make *replysize* larger than the default because it will grow rapidly. Furthermore, with the newer versions of MonetDB and pymonetdb, it is better to keep the size of the initial response small to transfer more data in the binary format.

1.4.4 Arraysize

The batching behavior of pymonetdb is governed mainly by *replysize* and *maxprefetch*, but the Python DBAPI also specifies the setting arraysize. The relationship between these three is as follows:

1. The *replysize* and *maxprefetch* settings are specific to pymonetdb, while *arraysize* comes from the Python DBAPI.
2. The DBAPI only uses *arraysize* as the default value for *fetchmany()* and says that it may influence the efficiency of *fetchall()*. It does not mention *arraysize* anywhere else.
3. In pymonetdb, the batching behavior is only influenced by *arraysize* if *fetchmany()* is used without an explicit size because then *arraysize* is used as the default size, and *fetchmany()* tries to round the batches to this size. It has no effect on *fetchall()* because that always fetches everything at once.
4. The DBAPI says that the default value for the *arraysize* of a newly created cursor is 1. Pymonetdb deviates from that, similar to, for example, python-oracledb. Pymonetdb uses the *replysize* of the connection instead. If *replysize* is not a positive integer, the default is 100.

In general, all this means that *arraysize* needs no tweaking.

1.5 API

1.5.1 Basic SQL usage

pymonetdb.**connect**(**args*, ***kwargs*) → pymonetdb.sql.connections.Connection

Set up a connection to a MonetDB SQL database.

database (str) name of the database, or MAPI URI (see below)

hostname (str) Hostname where MonetDB is running

port (int) port to connect to (default: 50000)

username (str) username for connection (default: "monetdb")

password (str) password for connection (default: "monetdb")

unix_socket (str) socket to connect to. used when hostname not set (default: "/tmp/.s.monetdb.50000")

autocommit (bool) enable/disable auto commit (default: false)

connect_timeout (int) the socket timeout while connecting

binary (int) enable binary result sets when possible if > 0 (default: 1)

replysize(int) number of rows to retrieve immediately after query execution (default: 100, -1 means everything)

maxprefetch(int) max. number of rows to prefetch during Cursor.fetchone() or Cursor.fetchmany()

MAPI URI Syntax:

tcp socket mapi:monetdb://[<username>[:<password>]@]<host>[:<port>]/<database>

unix domain socket mapi:monetdb:///[<username>[:<password>]@]path/to/socket?database=<database>

class pymonetdb.sql.connections.**Connection**(*database*, *hostname=None*, *port=50000*, *username='monetdb'*, *password='monetdb'*, *unix_socket=None*, *autocommit=False*, *host=None*, *user=None*, *connect_timeout=-1*, *binary=1*, *replysize=None*, *maxprefetch=None*)

Bases: object

A MonetDB SQL database connection

exception DataError

Bases: *pymonetdb.exceptions.DatabaseError*

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

exception DatabaseError

Bases: *pymonetdb.exceptions.Error*

Exception raised for errors that are related to the database. It must be a subclass of Error.

exception Error

Bases: Exception

Exception that is the base class of all other error exceptions. You can use this to catch all errors with one single 'except' statement. Warnings are not considered errors and thus should not use this class as base. It must be a subclass of the Python StandardError (defined in the module exceptions).

exception IntegrityError

Bases: *pymonetdb.exceptions.DatabaseError*

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

exception InterfaceError

Bases: *pymonetdb.exceptions.Error*

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

exception InternalError

Bases: *pymonetdb.exceptions.DatabaseError*

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 must be a subclass of DatabaseError.

exception NotSupportedError
Bases: *pymonetdb.exceptions.DatabaseError*

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 must be a subclass of DatabaseError.

exception OperationalError
Bases: *pymonetdb.exceptions.DatabaseError*

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 must be a subclass of DatabaseError.

exception ProgrammingError
Bases: *pymonetdb.exceptions.DatabaseError*

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 must be a subclass of DatabaseError.

exception Warning
Bases: `Exception`

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

binary

binary_command(*command*)
use this function to send low level mapi commands that return raw bytes

close()
Close the connection.

The connection will be unusable from this point forward; an Error exception will be raised if any operation is attempted with the connection. The same applies to all cursor objects trying to use the connection. Note that closing a connection without committing the changes first will cause an implicit rollback to be performed.

command(*command*)
use this function to send low level mapi commands

commit()
Commit any pending transaction to the database. Note that if the database supports an auto-commit feature, this must be initially off. An interface method may be provided to turn it back on.

Database modules that do not support transactions should implement this method with void functionality.

cursor()
Return a new Cursor Object using the connection. If the database does not provide a direct cursor concept, the module will have to emulate cursors using other means to the extent needed by this specification.

default_cursor
alias of *pymonetdb.sql.cursors.Cursor*

execute(*query*)
use this for executing SQL queries

get_binary() → int

get_maxprefetch() → int

get_replysize() → int

gettimeout()
get the amount of time before a connection times out

maxprefetch

replysize

rollback()
This method is optional since not all databases provide transaction support.

In case a database does provide transactions this method causes the database 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.

set_autocommit(*autocommit*)
Set auto commit on or off. 'autocommit' must be a boolean

set_binary(*binary: int*)

set_downloader(*downloader*)
Register a Downloader object which will handle file download requests.

Must be an instance of class pymonetdb.Downloader or None

set_maxprefetch(*maxprefetch: int*)

set_replysize(*replysize: int*)

set_sizeheader(*sizeheader*)
Set sizeheader on or off. When enabled monetdb will return the size a type. 'sizeheader' must be a boolean.

set_timezone(*seconds_east_of_utc*)

set_uploader(*uploader*)
Register an Uploader object which will handle file upload requests.

Must be an instance of class pymonetdb.Uploader or None.

settimeout(*timeout*)
set the amount of time before a connection times out

class pymonetdb.sql.cursors.**Cursor**(*connection: pymonetdb.sql.connections.Connection*)
Bases: object

This object represents a database cursor, which is used to manage the context of a fetch operation. Cursors created from the same connection are not isolated, i.e., any changes done to the database by a cursor are immediately visible by the other cursors

arraysize = None
Default value for the size parameter of *fetchmany()*.

binary

close()
Close the cursor now (rather than whenever __del__ is called). The cursor will be unusable from this point forward; an Error (or subclass) exception will be raised if any operation is attempted with the cursor.

debug(*query*, *fname*, *sample=-1*)
Locally debug a given Python UDF function in a SQL query using the PDB debugger. Optionally can run on only a sample of the input data, for faster data export.

execute(*operation: str*, *parameters: Optional[Dict[KT, VT]] = None*)
Prepare and execute a database operation (query or command). Parameters may be provided as mapping and will be bound to variables in the operation.

executemany(*operation*, *seq_of_parameters*)
Prepare a database operation (query or command) and then execute it against all parameter sequences or mappings found in the sequence seq_of_parameters.

It will return the number or rows affected

export(*query*, *fname*, *sample=-1*, *filespath='./'*)

fetchall()
Fetch all remaining rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples).

A `ProgrammingError` is raised if the previous call to .execute*() did not produce any result set or no call was issued yet.

fetchmany(*size=None*)
Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples). An empty sequence is returned when no more rows are available.

The number of rows to fetch per call is specified by the parameter. If it is not given, the cursor's arraysize determines the number of rows to be fetched.

A `ProgrammingError` is raised if the previous call to .execute*() did not produce any result set or no call was issued yet.

fetchone()
Fetch the next row of a query result set, returning a single sequence, or None when no more data is available.

get_binary() → int

get_maxprefetch() → int

get_replysize() → int

maxprefetch

next()

replysize

scroll(*value*, *mode='relative'*)
Scroll the cursor in the result set to a new position according to mode.

If mode is 'relative' (default), value is taken as offset to the current position in the result set, if set to 'absolute', value states an absolute target position.

An IndexError is raised in case a scroll operation would leave the result set.

set_binary(*level: int*)

set_maxprefetch(*maxprefetch: int*)

set_replysize(*replysize: int*)

setinputsizes(*sizes*)
This method would be used before the .execute*() method is invoked to reserve memory. This implementation doesn't use this.

setoutputsize(*size*, *column=None*)
Set a column buffer size for fetches of large columns This implementation doesn't use this

used_binary_protocol() → bool
Pymonetdb-specific. Return True if the last fetch{one,many,all} for the current statement made use of the binary protocol.

Primarily used for testing.

Note that the binary protocol is never used for the first few rows of a result set. Exactly when it kicks in depends on the *replysize* setting.

1.5.2 Type conversion

functions for converting python objects to monetdb SQL format. If you want to add support for a specific type you should add a function as a value to the mapping dict and the datatype as key.

pymonetdb.sql.monetize.**convert**(*data*)
Return the appropriate convertion function based upon the python type.

pymonetdb.sql.monetize.**monet_bool**(*data*)
returns "true" or "false"

pymonetdb.sql.monetize.**monet_bytes**(*data*)
converts bytes to string

pymonetdb.sql.monetize.**monet_date**(*data*)
returns a casted date

pymonetdb.sql.monetize.**monet_datetime**(*data*)
returns a casted timestamp

pymonetdb.sql.monetize.**monet_escape**(*data*)
returns an escaped string

pymonetdb.sql.monetize.**monet_none**(_)
returns a NULL string

pymonetdb.sql.monetize.**monet_time**(*data*)
returns a casted time

pymonetdb.sql.monetize.**monet_timedelta**(*data*)
returns timedelta casted to interval seconds

pymonetdb.sql.monetize.**monet_unicode**(*data*)

functions for converting monetdb SQL fields to Python objects

pymonetdb.sql.pythonize.**Binary**(*data*)
Convert to wraps binary data

pymonetdb.sql.pythonize.**DateFromTicks**(*ticks*)
Convert ticks to python Date

pymonetdb.sql.pythonize.**TimeFromTicks**(*ticks*)
Convert ticks to python Time

pymonetdb.sql.pythonize.**TimeTzFromTicks**(*ticks*)
Convert ticks to python Time

pymonetdb.sql.pythonize.**TimestampFromTicks**(*ticks*)
Convert ticks to python Timestamp

pymonetdb.sql.pythonize.**TimestampTzFromTicks**(*ticks*)
Convert ticks to python Timestamp

pymonetdb.sql.pythonize.**convert**(*data*, *type_code*)
Calls the appropriate convertion function based upon the python type

pymonetdb.sql.pythonize.**oid**(*data*)
represents an object identifier

For now we will just return the string representation just like mclient does.

pymonetdb.sql.pythonize.**py_bool**(*data*)
return python boolean

pymonetdb.sql.pythonize.**py_bytes**(*data: str*)
Returns a bytes (py3) or string (py2) object representing the input blob.

pymonetdb.sql.pythonize.**py_date**(*data*)
Returns a python Date

pymonetdb.sql.pythonize.**py_day_interval**(*data: str*) → int
Returns a python number of days where data represents a value of MonetDB's INTERVAL DAY type which resembles a stringified decimal.

pymonetdb.sql.pythonize.**py_sec_interval**(*data: str*) → datetime.timedelta
Returns a python TimeDelta where data represents a value of MonetDB's INTERVAL SECOND type which resembles a stringified decimal.

pymonetdb.sql.pythonize.**py_time**(*data*)
returns a python Time

pymonetdb.sql.pythonize.**py_timestamp**(*data*)
Returns a python Timestamp

pymonetdb.sql.pythonize.**py_timestamptz**(*data*)
Returns a python Timestamp where data contains a tz code

pymonetdb.sql.pythonize.**py_timetz**(*data*)
returns a python Time where data contains a tz code

pymonetdb.sql.pythonize.**strip**(*data*)
returns a python string, with chopped off quotes, and replaced escape characters

1.5.3 MAPI

This is the python implementation of the mapi protocol.

class pymonetdb.mapi.**Connection**
Bases: object

MAPI (low level MonetDB API) connection

binary_cmd(*operation: str*) → memoryview
put a mapi command on the line, with a binary response.

returns a memoryview that can only be used until the next operation on this Connection object.

cmd(*operation: str*)
put a mapi command on the line

connect(*database: str, username: str, password: str, language: str, hostname: Optional[str] = None, port: Optional[int] = None, unix_socket=None, connect_timeout=-1, handshake_options_callback: Callable[[bool], List[HandshakeOption]] = <function Connection.<lambda>>*)
setup connection to MAPI server

unix_socket is used if hostname is not defined.

disconnect()
disconnect from the monetdb server

set_downloader(*downloader: Downloader*)
Register the given Downloader, or None to deregister

set_reply_size(*size*)
Set the amount of rows returned by the server.

args: size: The number of rows

set_uploader(*uploader: Uploader*)
Register the given Uploader, or None to deregister

class pymonetdb.mapi.**HandshakeOption**(*level*, *name*, *fallback*, *value*)
Bases: object

Option that can be set during the MAPI handshake

Should be sent as <name>=<val>, where <val> is *value* converted to int. The *level* is used to determine if the server supports this option. The *fallback* is a function-like object that can be called with the value (not converted to an integer) as a parameter. Field *sent* can be used to keep track of whether the option has been sent.

pymonetdb.mapi.**handle_error**(*error*)
Return exception matching error code.

args: error (str): error string, potentially containing mapi error code

returns:

tuple (Exception, formatted error): returns OperationalError if unknown error or no error code in string

pymonetdb.mapi.**mapi_url_options**(*possible_mapi_url: str*) → Dict[str, str]
Try to parse the argument as a MAPI URL and return a Dict of url options

Return empty dict if it's not a MAPI URL.

1.5.4 File Uploads and Downloads

Classes related to file transfer requests as used by COPY INTO ON CLIENT.

class pymonetdb.filetransfer.**Upload**(*mapi: MapiConnection*)
Represents a request from the server to upload data to the server. It is passed to the Uploader registered by the application, which for example might retrieve the data from a file on the client system. See pymonetdb.sql.connections.Connection.set_uploader().

Use the method send_error() to refuse the upload, binary_writer() to get a binary file object to write to, or text_writer() to get a text-mode file object to write to.

Implementations should be VERY CAREFUL to validate the file name before opening any files on the client system!

is_cancelled() → bool
Returns true if the server has cancelled the upload.

has_been_used() → bool
Returns true if .send_error(), .text_writer() or .binary_writer() have been called.

set_chunk_size(*size: int*)
After every CHUNK_SIZE bytes, the server gets the opportunity to cancel the rest of the upload. Defaults to 1 MiB.

send_error(*message: str*) → None
Tell the server the requested upload has been refused

binary_writer() → io.BufferedIOBase
Returns a binary file-like object. All data written to it is uploaded to the server.

text_writer() → io.TextIOBase
Returns a text-mode file-like object. All text written to it is uploaded to the server. DOS/Windows style line endings (CR LF, \r \n) are automatically rewritten to single \n's.

close()
End the upload succesfully

class pymonetdb.filetransfer.**Uploader**
Base class for upload hooks. Instances of subclasses of this class can be registered using pymonetdb.Connection.set_uploader(). Every time an upload request is received, an Upload object is created and passed to this objects .handle_upload() method.

If the server cancels the upload halfway, the .cancel() methods is called and all further data written is ignored.

handle_upload(*upload: pymonetdb.filetransfer.uploads.Upload*, *filename: str*, *text_mode: bool*, *skip_amount: int*)
Called when an upload request is received. Implementations should either send an error using upload.send_error(), or request a writer using upload.text_writer() or upload.binary_writer(). All data written to the writer will be sent to the server.

Parameter 'filename' is the file name used in the COPY INTO statement. Parameter 'text_mode' indicates whether the server requested a text file or a binary file. In case of a text file, 'skip_amount' indicates the number of lines to skip. In binary mode, 'skip_amount' is always 0.

SECURITY NOTE! Make sure to carefully validate the file name before opening files on the file system. Otherwise, if an adversary has taken control of the network connection or of the server, they can use file upload requests to read arbitrary files from your computer (../../)

cancel()
Optional method called when the server cancels the upload.

class pymonetdb.filetransfer.**Download**(*mapi: pymonetdb.mapi.Connection*)
Represents a request from the server to download data from the server. It is passed to the Downloader registered by the application, which for example might write the data to a file on the client system. See pymonetdb.Connection.set_downloader().

Use the method send_error() to refuse the download, binary_reader() to get a binary file object to read bytes from, or text_reader() to get a text-mode file object to read text from.

Implementations should be EXTREMELY CAREFUL to validate the file name before opening and writing to any files on the client system!

send_error(*message: str*) → None
Tell the server the requested download is refused

binary_reader()
Returns a binary file-like object to read the downloaded data from.

text_reader()
Returns a text mode file-like object to read the downloaded data from.

close()
End the download succesfully. Any unconsumed data will be discarded.

class pymonetdb.filetransfer.**Downloader**

Base class for download hooks. Instances of subclasses of this class can be registered using pymonetdb.Connection.set_downloader(). Every time a download request arrives, a Download object is created and passed to this objects .handle_download() method.

SECURITY NOTE! Make sure to carefully validate the file name before opening files on the file system. Otherwise, if an adversary has taken control of the network connection or of the server, they can use download requests to OVERWRITE ARBITRARY FILES on your computer

handle_download(*download: pymonetdb.filetransfer.downloads.Download*, *filename: str*, *text_mode: bool*)

Called when a download request is received. Implementations should either send an error using download.send_error(), or request a reader using download.text_reader() or download.binary_reader().

Parameter 'filename' is the file name used in the COPY INTO statement. Parameter 'text_mode' indicates whether the server requested text or binary mode.

SECURITY NOTE! Make sure to carefully validate the file name before opening files on the file system. Otherwise, if an adversary has taken control of the network connection or of the server, they can use file download requests to overwrite arbitrary files on your computer. (../../)

class pymonetdb.filetransfer.**SafeDirectoryHandler**(*dir*, *encoding: Optional[str] = None*, *newline: Optional[str] = None*, *compression=True*)

File transfer handler which uploads and downloads files from a given directory, taking care not to allow access to files outside that directory. Instances of this class can be registered using the pymonetb.Connection's set_uploader() and set_downloader() methods.

When downloading text files, the downloaded text is converted according to the *encoding* and *newline* parameters, if present. Valid values for *encoding* are any encoding known to Python, or None. Valid values for *newline* are *"\n"*, *"\r\n"* or None. None means to use the system default.

For binary up- and downloads, no conversions are applied.

When uploading text files, the *encoding* parameter indicates how the text is read and *newline* is mostly ignored: both \n and \r\n are valid line endings. The exception is that because the server expects its input to be \n-terminated UTF-8 text, if you set encoding to "utf-8" and newline to "\n", text mode transfers are performed as binary, which improves performance. For uploads, only do this if you are absolutely, positively sure that all files in the directory are actually valid UTF-8 encoded and have Unix line endings.

If *compression* is set to True, which is the default, the SafeDirectoryHandler will automatically compress and decompress files with extensions .gz, .bz2, .xz and .lz4. Note that the first three algorithms are built into Python, but LZ4 only works if the lz4.frame module is available.

handle_upload(*upload: pymonetdb.filetransfer.uploads.Upload*, *filename: str*, *text_mode: bool*, *skip_amount: int*)

Meta private

handle_download(*download: pymonetdb.filetransfer.downloads.Download*, *filename: str*, *text_mode: bool*)

Called when a download request is received. Implementations should either send an error using download.send_error(), or request a reader using download.text_reader() or download.binary_reader().

Parameter 'filename' is the file name used in the COPY INTO statement. Parameter 'text_mode' indicates whether the server requested text or binary mode.

SECURITY NOTE! Make sure to carefully validate the file name before opening files on the file system. Otherwise, if an adversary has taken control of the network connection or of the server, they can use file download requests to overwrite arbitrary files on your computer. (../../)

1.5.5 MonetDB remote control

class pymonetdb.control.**Control**(*hostname=None*, *port=50000*, *passphrase=None*, *unix_socket=None*, *connect_timeout=-1*)

Bases: object

Use this module to manage your MonetDB databases. You can create, start, stop, lock, unlock, destroy your databases and request status information.

create(*database_name*)
Initialises a new database or multiplexfunnel in the MonetDB Server. A database created with this command makes it available for use, however in maintenance mode (see pymonetdb lock).

defaults()

destroy(*database_name*)
Removes the given database, including all its data and logfiles. Once destroy has completed, all data is lost. Be careful when using this command.

get(*database_name*)
gets value for property for the given database, or retrieves all properties for the given database

inherit(*database_name*, *property_*)
unsets property, reverting to its inherited value from the default configuration for the given database

kill(*database_name*)
Kills the given database, if the MonetDB Database Server is running. Note: killing a database should only be done as last resort to stop a database. A database being killed may end up with data loss.

lock(*database_name*)
Puts the given database in maintenance mode. A database under maintenance can only be connected to by the DBA. A database which is under maintenance is not started automatically. Use the "release" command to bring the database back for normal usage.

neighbours()

release(*database_name*)
Brings back a database from maintenance mode. A released database is available again for normal use. Use the "lock" command to take a database under maintenance.

rename(*old*, *new*)

set(*database_name*, *property_*, *value*)
sets property to value for the given database for a list of properties, use *pymonetdb get all*

start(*database_name*)
Starts the given database, if the MonetDB Database Server is running.

status(*database_name=False*)
Shows the state of a given glob-style database match, or all known if none given. Instead of the normal mode, a long and crash mode control what information is displayed.

stop(*database_name*)
Stops the given database, if the MonetDB Database Server is running.

pymonetdb.control.**isempty**(*result*)
raises an exception if the result is not empty

pymonetdb.control.**parse_statusline**(*line*)
parses a sabdb format status line. Support v1 and v2.

1.5.6 pymonetdb Exceptions

MonetDB Python API specific exceptions

exception pymonetdb.exceptions.**DataError**
Bases: *pymonetdb.exceptions.DatabaseError*

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

exception pymonetdb.exceptions.**DatabaseError**
Bases: *pymonetdb.exceptions.Error*

Exception raised for errors that are related to the database. It must be a subclass of Error.

exception pymonetdb.exceptions.**Error**
Bases: Exception

Exception that is the base class of all other error exceptions. You can use this to catch all errors with one single 'except' statement. Warnings are not considered errors and thus should not use this class as base. It must be a subclass of the Python StandardError (defined in the module exceptions).

exception pymonetdb.exceptions.**IntegrityError**
Bases: *pymonetdb.exceptions.DatabaseError*

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

exception pymonetdb.exceptions.**InterfaceError**
Bases: *pymonetdb.exceptions.Error*

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

exception pymonetdb.exceptions.**InternalError**
Bases: *pymonetdb.exceptions.DatabaseError*

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 must be a subclass of DatabaseError.

exception pymonetdb.exceptions.**NotSupportedError**
Bases: *pymonetdb.exceptions.DatabaseError*

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 must be a subclass of DatabaseError.

exception pymonetdb.exceptions.**OperationalError**
Bases: *pymonetdb.exceptions.DatabaseError*

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 must be a subclass of DatabaseError.

exception pymonetdb.exceptions.**ProgrammingError**
Bases: *pymonetdb.exceptions.DatabaseError*

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 must be a subclass of DatabaseError.

exception pymonetdb.exceptions.**Warning**
Bases: Exception

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

1.6 Development

1.6.1 Github

We maintain pymonetdb on GitHub. If you have problems with pymonetdb, please raise an issue in the issue tracker. Even better is if you have a solution to the problem! In that case, you can make our lives easier by following these steps:

- Fork our repository on GitHub
- Add tests that will fail because of the problem
- Fix the problem
- Run the test suite again
- Commit all changes to your repository
- Issue a GitHub pull request.

Also, we try to be pep8 compatible as much as possible, where possible and reasonable.

1.6.2 Test suite

pymonetdb comes with a test suite to verify that the code works and make development easier.

Prepare test databases

Most tests use an existing MonetDB database that you must prepare beforehand. By default they try to connect to a database named "demo" but this can be configured otherwise, see below.

Some of the tests rely on a running MonetDB daemon, to test creating and destroying new databases. This daemon also needs to be prepared beforehand, and configured to allow control connections. Alternatively, you may disable the control tests by setting the environment variable *TSTCONTROL=off*.

The commands below assume an environment without any running MonetDB processes.

Create a test database farm, e.g. "/tmp/pymonetdbtest", and the "demo" database:

```
$ monetdbd create /tmp/pymonetdbtest
$ monetdbd start /tmp/pymonetdbtest
$ monetdb create demo
$ monetdb release demo
```

If you want to run the control tests (in *tests/test_control.py*), you need to set a passphrase and enable remote control:

```
$ monetdbd set control=yes /tmp/pymonetdbtest
$ monetdbd set passphrase=testdb /tmp/pymonetdbtest
$ monetdbd stop /tmp/pymonetdbtest
$ monetdbd start /tmp/pymonetdbtest
```

Note 1: Test databases created by *test_control.py* are cleaned up after the control tests have finished. However, the *demo* database and the MonetDB daemon itself are neither stopped nor destroyed.

Note 2: The above commands are also in the file *tests/initdb.sh*. Once the database farm has been created, you can use that script to do the remaining work:

```
$ tests/initdb.sh demo /tmp/pymonetdbtest
```

WARNING: *initdb.sh* will destroy the given database *demo WITHOUT* asking for confirmation!

Run tests

There are many ways to run the tests. Below we list several often-used commands. The commands should be run in the root directory of the pymonetdb source directory.

- With Python unittest:

```
$ python -m unittest # to run all tests
$ python -m unittest -f # to run all tests but stop after the first failure
$ python -m unittest -v # to run all tests and get information about individual␣
↪test
$ python -m unittest -v tests.test_policy # to run all tests of the module "tests.
↪test_policy"
$ python -m unittest -v -k test_fetch # to run the sub-test set "test_fetch*"
```

- With *pytest*:

```
$ pytest # to run all tests
$ pytest -v # to run all tests and get information about individual test
$ pytest -v tests/test_oid.py # to run one test file
```

- With *make*:

```
$ make test
```

Note: *make test* creates a *venv* in which it installs and runs *pytest*. If you get the error "Could not install packages due to an OSError: [Errno 39] Directory not empty: '_internal'", it is probably because your pymonetdb source is in a Vagrant shared folder. A simple workaround is to move your pymonetdb source to a local folder on your VM. See also vagrant.

- With *tox*:

```
$ pip install tox; tox
```

Note: If it is not listed there, you must add your Python version to the *envlist* in the *tox.ini* file.

Environment variables

Several environment variables are defined in *tests/util.py*. Many of them are self-explanatory. Here we just highlight a few:

- TSTDB is the name of the preexisting database used by most of the tests. TSTHOSTNAME, TSTUSERNAME, TSTPASSWORD and MAPIPORT control the other connection parameters. Note that for historical reasons it is MAPIPORT, not TSTPORT.
- TSTPASSPHRASE is the Merovingian passphrase you must set to run the control test (see *Prepare test databases* above).

- Some tests are skipped unless you set TSTFULL to *true*, e.g.:

```
$ TSTFULL=true python3 -m unittest -v tests/test_control.py
```

- TSTCONTROL is used to control the tests in *test_control.py*. The default *tcp,local* means run the tests over TCP/IP (e.g. on port 50000) and the Unix domain socket (e.g. "/tmp/s.merovingian.50000"). When you run MonetDB in, e.g., a Docker container, you can turn off the tests over the Unix socket using *TSTCONTROL=tcp*. If you want to turn off all Merovingian tests, you can use *TSTCONTROL=off* (actually, any string other than "tcp" and "local" will do):

```
$ TSTFULL=true TSTCONTROL=tcp  python3 -m unittest -v tests/test_control.py
```

- TSTREPLYSIZE, TSTMAXPREFETCH and TSTBINARY control the size and format of the result set transfer (see *Result set batch size*). Check out the tests in *test_policy.py* for examples of implemented data transfer policies and how setting the variables *replysize*, *maxprefetch* and *binary* affects those policies.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W