adodbapi

A Python DB-API 2.0 module that makes it easy to use Microsoft ADO

for connecting with databases and other data sources

using either CPython or IronPython.

`Home page: http://sourceforge.net/projects/adodbapi`

For installation instructions, see the README.txt file, which you may also find at

https://pypi.python.org/pypi/adodbapi

About ADO and Connection Strings

_

Microsoft engineers invented ODBC as a standard way of connecting to a tabular data source, such as an SQL database server. It was such a good design that it was adopted almost universally. Therefore, Microsoft thought they should come up with something even better that everyone else did not have.

That was ADO. In truth, ADO does have some features that ODBC does not have. One powerful anti-feature is: it is proprietary. You have to have Windows to use it.

ODBC connections are made using a string, called a “data set name”, abbreviated “dsn”. A dsn can be very short. There is a utility program which stores the translations of a dsn to all of the information needed to connect to that data store. There is a more-or-less defined standard of how to pass that additional information using the dsn string for those cases where the dsn is not already known by the system.

ADO uses a completely different way of accomplishing the same goal. To make matters worse, ADO operates, by default, in ODBC mode, so it must be able to process this information in either format. The resulting mass of confusion is called a “connection string”. There is an entire web site dedicated to giving examples of connection strings in numerous combinations. See http://www.connectionstrings.com

and for more explanation: http://www.asp101.com/articles/john/connstring/default.asp

The software which connects ODBC to an engine is called a “Driver”. One which talks in ADO is called a “Provider”. Sometimes there will be a Provider for a Driver.

Driver (and Provider) download links:

Jet (ACCESS database) and other file datasets (like .xls and .csv) “ACE” Provider:

http://www.microsoft.com/en-us/download/details.aspx?id=13255 a (32 bit) Microsoft.Jet.OLEDB.4.0 Provider, which may be included with Windows. Note that you are not permitted load the 32 bit “ACE” provider if you have any 64-bit Office components installed. Conventional wisdom says that you must use 64 bit Python in this case. However, see the answer in http://stackoverflow.com/questions/12270453/ms-access-db-engine-32-bit-with-office-64-bit. If you decide to try hacking the installers, you may find http://www.pantaray.com/msi_super_orca.html to be a useful alternative to Orca. My experience is that such a hacked installer can also be used on machines where “click to buy” versions of Office have been removed, but are still blocking installation of the redistributable provider.

To use any ODBC driver from 64 bit Python, you also need the MSDASQL provider. It is shipped with Server 2008, and Vista and later. For Server 2003, You will have to download it from Microsoft.

MySQL driver http://dev.mysql.com/downloads/connector/odbc/

PostgreSQL driver http://www.postgresql.org/ftp/odbc/versions/msi/

____

PEP-249

____

This “quick reference” assumes that you are already familiar with Python, SQL, and the general principles of accessing a database using the PEP-249 api. I may write a book later. Here, I only intend to cover the extensions and special features of adodbapi. The PEP-249 database access api specification is found at:

http://www.python.org/dev/peps/pep-0249/

Module level attributes:

The PEP requires several module level attributes. Older versions of adodbapi (which was once all one big file) defined a hundred or two. I hate that, but can't break old code, so I decided to fix the problem for Python 3. If using Python3 the programmer must take the time to pick up the symbols she needs from apibase and ado_consts.

Part of the adodbapi package's __init__.py looks something like this:

```
...
    if sys.version_info < (3,0): # in Python 2, define all symbols, just like before
        from apibase import *       # using this is bad
        from ado_consts import *     # using this is worse
    else:
        # but if the user is running Python 3, then keep the dictionary clean
        from apibase import apilevel, threadsafety, paramstyle
        from apibase import Warning, Error, InterfaceError, DatabaseError, DataError
        from apibase import OperationalError, IntegrityError
        from apibase import InternalError, ProgrammingError, NotSupportedError
        from apibase import NUMBER, STRING, BINARY, DATETIME, ROWID

    from adodbapi import connect, Connection, __version__
    version = 'adodbapi v' + __version__
...
```

Please, use only those last four symbols from adodbapi. All others should be imported directly from their own sub-modules. My tests and examples all follow that rule.

Connect   (Use of the connect constructor)

As required by the PEP, the simplest way to connect is to use a “data set name”...

```
    import adodbapi
    myConn = adodbapi.connect('myDataSetName')
```

Which will work just fine, provided you (or someone) has done all of the hard work by going into “Control Panel” → “Administrative Tools” → “Data Sources (ODBC)”

and set everything up for you under “myDataSetName”.

Usually, life is not so simple...

```
import adodbapi
myhost = r".\SQLEXPRESS"
mydatabase = "Northwind"
myuser = "guest"
mypassword = "12345678"
connStr = """Provider=SQLOLEDB.1; User ID=%s; Password=%s;
    Initial Catalog=%s;Data Source= %s"""
myConnStr = connStr % (myuser, mypassword, mydatabase, myhost)
myConn = adodbapi.connect(myConnStr)
```

The PEP suggests that we should be able to create the connection using positional arguments. For us, that is difficult, because the syntax of an ADO connection string is so varied that we have to tell the module where to insert the arguments. In order to do that, use Python's old-sytle string formatting operations and label the places where the arguments should be inserted. An argument named "spam" would be located by a "%(spam)s" construct. The second, third, forth, and fifth arguments are defined (by standard) as being "user", "password", "host", and "database".

```
connStr = """Provider=SQLOLEDB.1; User ID=%(user)s;
    Password=%(password)s;"Initial Catalog=%(database)s;
    Data Source= %(host)s"""
myConn=adodbapi.connect(connStr, myuser, mypassword, myhost, mydatabase)
```

Which will work.

It would be better documented, however, to use keyword, rather than positional arguments:

```
myConn = adodbapi.connect(connStr, user=myuser, password=mypassword,
     host=myhost, database=mydatabase)
```

In adodbapi, you may also pass keywords using a dictionary structure, which also allows us to pass additional arguments:

```
conn_args = {'host' : r".\SQLEXPRESS",
    'database' : "Northwind",
    'user' : "guest",
    'password' : "12345678",
    'timeout' : 60}
myConn = adodbapi.connect(connStr,[],**conn_args)
```

Which works, but is ugly. So let's also put the connection string into the dictionary. As an extension, I allow the first (or second) positional argument to be the keyword dictionary.

```
conn_args = {'host' : r".\SQLEXPRESS",
   'database' : "Northwind",
   'user' : "guest",
   'password' : "12345678"}

conn_args['connection_string'] = """Provider=SQLOLEDB.1;
```

```
        User ID=%(user)s; Password=%(password)s;
        Initial Catalog=%(database)s; Data Source= %(host)s"""

    myConn = adodbapi.connect(conn_args)
```

Not pretty, I will admit, but it is about as readable as connection strings get. Arbitrary keywords, are also expanded using the same mechanism. For example, I often use "provider" to allow me to choose between database drivers. There are also built-in keywords which the constructor recognizes when building your connection. For example "timeout=n".

Dictionary Keywords which have meaning inside the connect method:

timeout=30 # set the ADO connection timeout to "n" seconds. (Default = 30)

paramstyle='qmark' # initialize the connection's paramstyle, like 'qmark', 'format', or 'named'. (Default = 'qmark')

autocommit=False # initialize autocommit on the connection. (Default = False)

Connection Keyword Macro Language:

Connection keywords can also be macros. See below...

Remote Connection

Included in the adodbapi package is a second db-api interface module, which is used to obtain proxy access to a host ADO system. The intended use of this is to run on a Linux computer allowing it to reach ADO data sources, such as MS SQL Server databases or "Jet" (a.k.a. ACCESS) .mdb data files.

It should take the same argument keywords as the host adodbapi server, and will pass them to it. In addition, other connection keywords are used to control the connection to the proxy server.

[Implementation note: adodbapi version 2.5 and 2.6 use PYRO4 for proxy communication. The will probably change in the future to use `ØMQ.`]

----

keywords for remote connections:

pyro_connection : 'PYRO:ado.connection@%(proxy_host)s:%(proxy_port)s'

# used to build the Pyro4 connection to the proxy. You may never need to change the default.

proxy_host : '::1' # the address of the ADO proxy server. (default = IPv6 localhost)

proxy_port : `'9099' # the IP port for the proxy connection.`

To connect to the same database as above, assuming that the Windows box running the proxy server (an SQLEXPRESS server) was at IPv4 address 10.11.12.13, you would use something like:

```
import adodbapi.remote as db
conn_args = {'host' : r".\SQLEXPRESS",
   'database' : "Northwind",
   'user' : "guest",
   'password' : "12345678"}
conn_args['connection_string'] = """Provider=SQLOLEDB.1;
     User ID=%(user)s; Password=%(password)s;
     Initial Catalog=%(database)s; Data Source= %(host)s"""
conn_args['proxy_host'] = '10.11.12.13'
myConn = db.connect(conn_args)
```

In other words, you only need to add the address of the proxy server to whatever connection string you would have used at the server itself, then connect using adodbapi.remote.connect() rather than adodbapi.connect().

Some limitations: Remote connections do not allow varientConversion customization, nor customized error handlers.

Connection Keyword Macro Language:

It often happens, when building a connection string for a remote connection, that you need to know information about your proxy host -- such as, is it running 32 bit or 64 bit Python? Keyword macros provide a method of passing that decision on to the server.

A macro is defined by using a keyword which starts with "macro_" followed by the macro name. The value associated with the keyword must be a single string, or a valid Python sequence. The string or sequence is passed to the macro processor. The first ([0]) argument will become the new keyword which the macro will produce. If the macro requires arguments, they will be in subsequent members.

The result of the macro operation will be the value of the new key.

---

macro "is64bit": Test 64-bit-ed-ness of the proxy server.

If the server is running 64 bit Python, return argument[1], otherwise return argument[2]. Example:

```
conn_keys['macro_is64bit'] =  ['provider', 'Microsoft.ACE.OLEDB.12.0',
"Microsoft.Jet.OLEDB.4.0"]
conn_keys['connection_string'] = "Provider=%(provider)s; ...and...more...stuff"
```

---

macro "getuser": Retrieve the proxy server logged-in-user's username

My systems administrator gave me a test database named after myself.

I thought it would be handy to let others do a similar thing. so:

```
conn_keys['macro_getuser'] = 'database'
conn_keys['connection_string'] = "...stuff...; Initial Catalog=%(database)s; ..."
```

---

macro "auto_security": Build ADO security string automagically.

If the username (key "user") is missing, blank, or None, use Windows login security ... otherwise use SQL Server security with "user" and "password". (It runs this code...

```
....... if macro_name == "auto_security":
            if not 'user' in kwargs or not bool(kwargs['user']):
                return new_key, 'Integrated Security=SSPI'
            return new_key, 'User ID=%(user)s; Password=%(password)s'
        # note that %(user) and %(password) are not substituted here,
        # they are put in place to be substituted before being sent to ADO.

conn_keys['macro_auto_security'] = ['secure']
conn_keys['user'] =  None                 # username here for Server Security


conn_keys['password'] = 'xys'             # ignored if "user" is blank or undefined
conn_keys[connection_string] = "...stuff...; %(secure)s"
-----------------------------------------------------------------------
```

Proxy Server

The proxy server is a module within the adodbapi package. It can be run from the command line using the "-m" switch. (*) The host address and port number can be passed on the command line or by environment variables. (You should also set the environment variable "PYRO_HMAC_KEY" to some unique string for your installation.) The environment variables are "PYRO_HOST" and "PYRO_PORT". The command line arguments are "HOST=aa.bb.cc.dd" "PORT=nnn". IPv6 addresses will also work. The default is address ::0 and port 9099 (all IPv6 interfaces).

```
    C:>python -m adodbapi.server HOST=0.0.0.0
    ado.connection server running on uri=PYRO:ado.connection@0.0.0.0:9099
```

```
(*) Except Python 2.5. To run the proxy server, execute the following main program:
>>>#!Python2.5
>>>import adodbapi.server
>>>adodbapi.server.serve()
------------------------------------------------
```

Connection class

A connection object holds an ADO connection in its .connector attribute.

A connection object is usually created using the standard api constructor.

Internally, it creates an empty connection object, fills in the attributes needed, and then call its .connect() method, which calls the ADO Open method.

Connection Methods:

```
---
The standard methods are supplied:
```

.close() # close the connection (does an ADO Close)

.commit() # commits any pending transaction(s) on the connection

.rollback() # rolls back any pending transaction. If the engine attached to the present instance does not support transactions, this method will appear to not be present (an AttributeError will be raised), as per the PEP.

.cursor() # returns a new Cursor object for the connection.

```
---
```
Connection Methods: (non-standard)

.__enter__() # the connection is a context manager for transactions

.__exit__() # if no errors occurred, .commit(), otherwise .rollback()

.get_table_names() # returns a list of table names in your database. (schema)

Connection Attributes

.errorhandler # (standard extension. See PEP.) (does not work on remote)

.messages[] # (standard extension. See PEP)

.connector # (Internal) the ADO connection object

.paramstyle # can be altered by the programmer to change the paramstyle in use. The supported values are 'quark' (the default), 'format', and 'named'. Values of 'pyformat', and 'dynamic' are also accepted (see below).

```
The connection string keyword "paramstyle" will set the default for the class for
future connections.
```

.connection_string # the complete connection string which was used to start ADO.

.dbms_name # string identifying the actual database engine from the connection.

.dbms_version # string identifying the version of the db engine.

.variantConversions # a map of ado types to the functions used to import them. `(not available on remote)`

.supportsTransactions # (bool) this driver is capable of commit()/rollback().

.dbapi # references the module defining the connection. (A proposed db-api V3 extension.) This is a way for higher level code to reach module-level attributes.

----

the Cursor class

Cursor attributes:

.description # as defined by PEP-249 -- a sequence of 7-item sequences:

```
  for each column, defines the column by:
    [0] name: ADO field.Name
    [1] type_code: ADO field.Type -- (values defined in adodbapi.ado_consts)
    [2] display_size: ADO field.ActualSize or None
    [3] internal_size: ADO field.DefinedSize
    [4] precision: ADO field.Precision
    [5] scale: ADO field.NumericScale
    [6] null_ok: ADO field.Attributes & adFldMayBeNull
```

.rowcount # -1 means "not known"

ADO recordset.RecordCount (if it works)

otherwise, the count returned by the last ADO Execute operation.

Cursor attributes (standard extensions)

--

.connection # back-link to the connection

.errorhandler # (see PEP 249 -- a function to process exceptions.)

.messages[] # (see PEP 249)

.arraysize (=1) # the default number of rows to fetch in fetchmany()

Cursor attributes (non-standard)

.paramstyle # can be altered by the user to change paramstyle processing.

```
   (default taken from connection.)  (see below)
```

.rs # the internal ADO recordset (local) or raw unpickled data (remote)

.converters[] # a list of input-conversion functions, one per column.

```
  (not available on remote)
```

.columnNames{} # a dictionary of: ((lower-cased) column name : (column number).

.numberOfColumns # number of columns in present record set.

.command # the last raw SQL command sent to the query (before any reformatting)

.query # the text of last operation, as converted by reformatting

.return_value # the result returned by a previous .callproc()

Cursor Methods (standard)

.callproc() # execute a stored procedure, returning parameter values.

```
  A "returned value" (not an "out" parameter) will be in crsr.returnValue
  --> returns the (modified) parameter list
```

.close() # close the cursor, free the recordset

```
(NOTE: non-standard: in adodbapi, it is NOT an error to re-close a closed cursor)
```

.execute(operation, parameters) # execute a query or command...

```
 -- operation: the text of the SQL.
 -- parameters: parameters for the query or command...
    if paramstyle is 'qmark' this must be a sequence.
    if paramstyle is 'named' this must be a mapping (dictionary).
    If paramstyle is 'dynamic', 'format', or 'pyformat' it could be either,
      but your SQL format must be appropriate for your choice (see below).

 --> None. There is no return value, use .fetchxxx() to see data.
```

.executemany(operation, sequence-of-parameters) # runs the SQL operation several times, once for each group of parameters in sequence-of-parameters.

```
--> .rowcount will be the sum of all .rowcounts, unless any was -1.
```

.fetchone() # get the next row from the result set.

```
    Calls ADO recordset.GetRows(1)
```

.fetchmany(size=cursor.arraysize) # get a "size" sequence of rows.

```
    Calls ADO recordset.GetRows(size)
```

.fetchall() # attempt to retrieve all remaining rows in the result set.

```
   Calls ADO recordset.GetRows() using a local cursor so may use a great deal of memory
if the query set is large.
```

.nextset() # If an operation (such as a stored procedure call) has produced multiple result sets, skip to the next available result set.

```
  --> returns None if there are no more result sets, otherwise True.
```

.setinputsizes() # pass

.setoutputsizes() # pass

Cursor methods (extensions)

.prepare(operation) # initiate an SQL prepared statement.

This method actually does very little, and the use of it may not speed up your program very much, if at all. It does store the SQL statement you pass (operation) in the cursor's self.command attribute, and sends the appropriate flag to ADO. It will cache converted paramstyle operation strings. Calling .execute() with any other string, or calling .prepare() again will invalidate the preparation.

For example: cursor.executemany() is programmed internally like:

```
def executemany(self, operation, sequence_of_parameter_sequences):
    self.prepare(operation)
    for params in sequence_of_parameter_sequences:
        self.execute(self.command, params)
```

.get_returned_parameters()

# some providers will not return the (modified) parameter `list, nor the return_value, until after the last recordset is closed.  This method can be called (_after_ calling nextset() until it returns None) to get those values.`

.next() # The cursor can be used as an iterator, each iteration does fetchone()

.__iter__()

.__enter__() # the cursor is a context manager which will auto-close

.__exit__()

PARAMSTYLEs

What???

((A quick explanation of "paramstyle" follows: if you already know this -- skip this section.))

Many SQL queries or commands require data as part of their content. For example:

```
   UPDATE cheese SET qtyonhand = 0 WHERE name = 'MUNSTER'
```

Chances are, this command will be given many times, but with different values where "0" and "MUNSTER" appear in this command. It is convenient (as well as more efficient in some cases) to pass these values as parameters.

Unfortunately, the ISO SQL standard has not defined a method for doing this, so every designer of an SQL engine is free to choose a new way of expressing it. PEP-249 recognizes several popular methods, and provides a way for the API module writer to communicate to the programmer which method he expects. It is left up to the programmer to adapt to whichever method the engine requires. According to PEP, the module author will place a string in the module attribute named ".paramstyle" telling you which method to use. The defined possibilities are as follows...

'qmark' is the default used by ADO. As far as I know, every ADO driver uses it. The above query would be altered by placing a question mark where the parameters should go. The programmer then passes the parameters (as as sequence) in the correct order for the '?'s.

```
   sql = "UPDATE cheese SET qtyonhand = ? WHERE name = ?"
   args = [0, 'MUNSTER']
   crsr.execute(sql, args)
```

'format' is used my many database engines, and blindly expected by django. The format looks like (but is emphatically not) the same as Python "%s" string substitution.

Again, the programmer supplies the parameters (as a sequence) in the correct order.

```
   sql = "UPDATE cheese SET qtyonhand = %s WHERE name = %s"
   args = [0, 'MUNSTER']
   crsr.execute(sql, args)
```

'named' is used by Oracle, and is superior because it eliminates counting. Rather than keeping track of the parameters by position, the programmer passes the arguments as a mapping (dictionary) where each name is the key to find the correct place to make the substitution. She places names in the SQL string delimited by a leading colon (":"). Each key calls up the value for that place. In the following simple example, the syntax looks needlessly complex, but in queries which take a dozen or more parameters, it really helps.

```
sql = "UPDATE cheese SET qtyonhand = :qty WHERE name = :prodname"
args = {'qty' : 0, 'prodname' : 'MUNSTER'}
crsr.execute(sql, args)
```

'pyformat' also takes a dictionary of arguments, but uses a syntax like Python's “%” string operator:

```
UPDATE cheese SET qtyonhand = %(qty)s WHERE name =%(prodname)s
args = {'qty' : 0, 'prodname' : 'MUNSTER'}
crsr.execute(sql, args)
```

[Note: in adodbapi, 'format' and 'pyformat' both use the same subroutine. It depends on the presence of “%(” in your SQL operation string to hint whether to expect a mapping or a sequence. That seems to be the way some other api adapters operate.]

The other paramstyle possibility mentioned in the PEP is: **'numeric'**, which takes a sequence. It not implemented in adodbapi. (If you want it, patches will be considered.):

```
UPDATE cheese SET qtyonhand = :1 WHERE name = :2
```

((Gurus: Start reading again here))

> **'dynamic'** paramstyle is a non-standard extension in adodbapi. It acts the same as 'qmark', unless you pass a mapping (i.e. a dictionary) of parameters to your .execute() method – in which case it will act the same as 'named'.

As an extension, adodbapi copies the module's paramstyle attribute to each connection, and then to each cursor. The programmer is allowed to change the paramstyle to the one she prefers to use. When the .execute() method runs, it checks which paramstyle is expected, then converts the operation string into 'qmark', and then makes the ADO Execute call. Extracting the appropriate SQL table values from Python objects is a complex operation. If your table receives unexpected values, try to simplify the objects you present as parameters.   Set the ".verbose" attribute > 2 to get a debug listing of the parameters.

You may switch paramstyle as often as desired. If you want to use 'named' for UPDATEs and 'qmark' for INSERTs, go right ahead. (Restriction: provide a new SQL operation string object after you change paramstyle. I don't invalidate the cache.)

Row Class & Rows Class for returned data

The PEP specifies that .fetchone() returns a "single sequence". It does not spell out what kind of a sequence. Originally, adodbapi returned a tuple. It is supposed to be up to the programmer to count columns of the returned data to select the correct thing. In the past, I often resorted to putting long lists of constants into my Python code to keep track of which column a dutum was in.

The Row class satisfies the PEP by having .fetchone() return a sequence-like “Row” object. It can be indexed and sliced like a list. This is the PEP standard method:

```
  crsr.execute("SELECT prodname, price, qtyonhand FROM cheese")
  row = crsr.fetchone()
  while row:
     value = row[1] * row[2]
     print('Your {:10s} is worth {:10.2f}'.format(row[0], value))
     row = crsr.fetchone()  # returns None when no data remains
```

As an extension, a Row object can also be indexed by column name:

```
  crsr.execute("SELECT prodname, price, qtyonhand FROM cheese")
  for row in crsr:                        # note extension: using crsr as an iterator
     value = row['price'] * row['qtyonhand']
     print('Your {:10s} is worth {:10.2f}'.format(row['prodname'], value))
```

But, <u>really</u> lazy programmers, like me, use the column names as attributes:

```
  crsr.execute("SELECT prodname, price, qtyonhand FROM cheese")
  for row in crsr:
     value = row.price * row.qtyonhand
     print('Your {:10s} is worth {:10.2f}'.format(row.prodname, value))
```

Now, isn't that easier to read and understand?

The PEP specifies that .fetchmany() and .fetchall() must return "a sequence of sequences". This as satisfied by returning a Rows object, which emits a sequence of Row objects. Both Row and Rows are actually lazy -- they do not fetch data from the queryset until the programmer asks for it.

----

variantConversions for the connection and the cursor

(((NOTE: **Remote** connections cannot support variant conversions. All values are converted to proper Python form before being pickled and sent to the remote. If you really need other conversions, you will have to customize the server module.)))

Each Connection instance has an optional .variantConversions attribute. Usually it will not be present. If it is present, it will be used in preference to the module variantConversions attribute. In order to avoid unintentional alterations, when you make an assignment to the attribute, it will be actually copied (by __setattr__) rather than merely referenced.

In older versions of adodbapi, the accepted practice for user defined conversions was to modify the module's VariantConversionMap. That still works, but please refrain.

A **variantConversionsMap** is a dictionary of ADO variable types (as defined in adodbapi.ado_consts) with the function to be used to read each type and convert it into a Python value. As a convenience, the __setitem__ for this class will accept a sequence, and will enter a mapping for each member of the sequence with the same value. So, to change several data retrieval functions:

```
    import adodbapi.ado_consts as adc
    import adodbapi.apibase as api
    conn.variantConversions = api.variantConversions  # MAGIC: will make a copy.

    conn.variantConversions[(adc.adTinyInt, adc.adChar)] = myByteFunc
```

which will be equivalent to:

```
    conn.variantConversions[adc.adTinyInt] = myByteFunc
    conn.variantConversions[adc.adChar] = myByteFunc
```

Also, there is a supply of sequences used to initialize to default map, in module adodbapi.apibase. For example:

```
adoApproximateNumericTypes = (adc.adDouble, adc.adSingle)
```

If I wish to retrieve all floating point values differently, I might use:

```
    conn.variantConversions = api.variantConversions
    conn.variantConversions[api.adoApproximateNumericTypes] = myFloatFunc
```

The cursor builds a list of conversion functions for each new query set,

(in other words, each call to .execute(), .callproc() or .nextset()).

You may override the conversion for any column by altering the function for that column:

```
    crsr.conversions[4] = myFunc # change the reader for the fifth column
    crsr.fetchone()
```

To do this by column name, use:

```
    crsr.conversions[crsr.columnNames['mycolumn']] = myFunc
```

The Examples folder:

Several small complete example programs are included:

db_print_simple.py:

```
A simple as possibe example.  Opens a local .mdb (ACCESS) datebase and reads rows from
a table.
```

db_table_names.py:

```
Opens the same database & prints out a list of the tables in it.
```

is64bit.py:

```
A copy of the one in the package, just here for import convenience.
is64bit.Python() → bool
is64bit.os() → bool
```

test.mdb:

```
The sample database for the examples.  (Also copied into the test environment temporary
folder when testing 64-bit Python programs. They cannot build .mdb files.)
```

xls_write.py:

```
Creates a simple Excel spreadsheet.
```

xls_read.py and remote_xls_read.py:

```
Read the above spreadsheet:
```

db_print.py:

The only smart program in the folder. By default, this does the same thing as db_print_simple. However, it also looks for command line arguments. Type “help” to see the options.

If you include “proxy_host=::1” it will switch to remote mode and try to connect to an ADO server at the specified address. (“::1” is the IPv6 localhost.)

If you happen to have a server running (also in the examples folder using default settings) it will print out the default report, but will do so using the server. Not exciting – but you can also do it from the other side of the planet if you type the correct IP addresses.

If you use “table_name=?” it will print a list of the tables in the .mdb.

“proxy_port=nnn” lets you change from the default 9099.

“filename=xxx” lets you open a different.mdb.

Note for Remote users: the database is opened by the server, so filename= is from the server's point of view, not the client.

Running the tests

The test folder contains a set of unittest programs. Setting them up can be a bit complex, because you need several database servers to do a complete test, and each one has a different configuration. Scripts in this folder try to work in Python 2.n or Python 3.3. You will have to run them through 2to3 to test Python 3.0 to 3.2.

`dbapi20.py:`
[The standard test for API-2.0 compliance. Do not run this, it is imported only.]

`test_adodbapi_dbapi20.py:`
Imports the above and runs it on your local machine. You may have to actually install adodbapi in your Python library for it to work exactly right, especially on Python 3.0 to 3.2. It only works correctly against a Microsoft SQL Server. SQLexpress will do.

The settings for how to reach your SQL server are in this file. Edit as needed.

`test_remote_dbapi20.py:`
Runs the dbapi20 test script using a proxy host to open the SQL server database. You must have an ADO connection server running, of course. You can type the connection information on the command line using “proxy_host=n.n.n.n” or “proxy_host=n:n” and “proxy_port=nnn” The default port is 9099. The default host is my private SQL server.

This test will run on a Linux or IronPython client. Pyro4 must be installed. There is a Pyro restriction that the server and client must be running the same major version of Python. A Python2.n client cannot talk to a Python3.n server.

`adodbapitest.py:`
This tests the special features and extensions of adodbapi. It is much more complex than the dbapi20 test. You run this program, and it imports adodbapitestconfig.py, which is where you put all of your setup information.

The full operation runs up to 12 series of tests: four different database servers, and three date-time formats, in all combinations. If the database servers are distant, this can take a while. In my present case, my PostgreSQL server is in California, my MySQL server is in Wyoming, and the SQL Server is in another city in Nigeria, so it is creepy slow – but it works.

It does some lightweight command line processing (actually the config does it).

```
“--package” tries to build a proper Python package in a temporary location and adds it
to sys.path so it can import a test version of the code.  It will run 2to3 when it does
this, if needed.

“--all” run as many of the 12 passes as possible.

“--mysql” run the MySQL tests.

“--pg” run the PostgreSQL tests.

“--time” run all time tests.  (If mx-DateTime is not installed it will be skipped.)

“--verbose=n” gives lots of information.
```

`adodbapitestconfig.py:   – E D I T   T H I S   F I L E !`
You can run this script alone to test your settings quickly. Normally it is imported.

This sets the configuration for its parent, creates a temporary work directory, reads the command line arguments, etc. You will edit this file -- a lot! Many literals are set for my test environment. You must change them to yours.

```
runtests.bat – convenient way to run the main and api tests using different Python
versions.

startserver.bat -- convenient way to start a Pyro nameserver and ado.connection server.
Also initializes a not-so temporary .mdb file for remote tests to use.
```

```
You must supply the IP address to bind the nameserver's address for your connection
server.  It cannot be generic '0.0.0.0' or ::0. The default is '::1'.
C:>startserver 192.168.25.45
```

```
setuptestframework.py:
```

If run as a main program, initialize a not-really-temporary directory for the server to use for remote testing. (Otherwise, it is a subroutine for test setup.)

```
build3test.bat: Runs 2to3 to make a Python 3.0 to 3.2 test setup. (Python 3.3 and later
do not need conversion.)
```