ADO简明教程

Chapter 0: Introduction-ADO Programming with Python Tutorial

Python is an object-oriented programming language that has gained a lot of popularity in recent years. ADO (or ActiveX Data Objects) is a Microsoft concept that has also gained popularity on Windows platforms. However, there is precious little documentation on the web that deals with both ADO and Python. This tutorial aims to remedy that and provide some ideas about how to use COM objects with Python as well.

The tutorial will start off with a quick walkthrough of some ADO concepts, followed by practical Python code. The reader is assumed to have basic familiarity with SQL and Python.

DISCLAIMER: While every precaution as been taken in the preparation of this tutorial, the author assumes no responsibility for any errors or omissions, or for damages resulting from the use of the information contained herein.

Chapter 1: What is ADO?

ActiveX Data Objects or ADO, as it is more popularly known, is a Microsoft technology that they introduced during the last quarter of 1996. Originally, it was intended to provide a database independent method of accessing data from a wide range of databases. This means you can use the same functions whether you are using Access, Microsoft SQL Server, Sybase, Oracle etc. as your database engine. Microsoft had already pioneered the ODBC (Object DataBase Connectivity) interface standard years ago and it was widely used in the industry, but it was outdated and the cracks were beginning to show. Hence, Microsoft went for a new programming standard and thus ADO was born.

Some other languages, notably Perl and Delphi, also provide database independent methods of accessing data via their DBI or BDE interfaces. However, ADO was also intended to be somewhat language independent, provided your language had support for COM interfaces. Thus, you can use ADO with Python, Delphi, VB, ASP, Visual C++, any .NET language, Borland C++ Builder etc. Originally ADO was only intended to work with relational databases, but its role has been enhanced in recent years. With version 2.5, ADO also gained the capability of dealing with folders, e-mail messages, text files etc. With version 2.7 came the ability to work with XML and engines like SQL Server 2000. This tutorial will deal with using ADO to talk to databases alone and leave the other stuff for another article perhaps.

One disadvantage of ADO is that its usage is currently restricted to Microsoft operating systems (Windows 98, 2000, ME, XP etc.) alone. Their earlier ODBC standard was gradually accepted by other vendors and thus you can find ODBC drivers for UNIX platforms as well. Hopefully the ADO model will move in that direction one day. For now though, it means that a client program that uses ADO can only run on Windows. However, the database server that the client program connects to, can run on any platform. Hence, this is not a severe issue because this is the setup that most sites have -- (i.e.) client machines running on Windows connecting to a database server that may be running on Windows, *NIX or whatever.

Advantages

Designed to be database independent.

Language independent, provided language has a COM interface. Python, Perl, Visual Basic, Visual C++, Delphi, ASP, C++ Builder, Ruby, Java, C# and other .NET languages all support COM interfaces and can thus use ADO. Wide variety of drivers available.

Disadvantages

A program that uses ADO to connect to the database can only be on a Microsoft operating system currently. The database itself can be on a non-microsoft platform though. In fact, the MySQL examples in this tutorial were executed by the author from a Windows 2000 Pro client computer connecting to a Linux server running MySQL.

Now that you have a basic idea of what ADO is and what it can do for you, we will begin to explore the various objects in the next chapter.

Chapter 2: Basic ADO Objects

There are quite a few types of ADO objects but our discussion will be restricted to five main ones. In this section, we will quickly cover five objects and then discuss each one in detail in succeeding chapters. Additional information may be obtained from the Microsoft

MSDN website.

Connection Object

The connection object is used to manage a connection to a data source. You typically supply connection parameters such as the driver name, address of database server, username, password and various connection options to this object and then instruct it to connect to the database server. All other ADO objects are then hooked to this connection object. The connection object also has a method to execute statements directly without the need for any other ADO object. We will discuss this method, but not use it heavily for reasons explained later.

RecordSet Object

The RecordSet object is typically used to iterate through the results of a query or a stored procedure. The RecordSet object can access a row at a time and has a collection of Fields, which represent individual columns in the row. The RecordSet object also has methods to move to the next row, previous row, first row and last row of a set of results. Additionally, if a stored procedure returns multiple sets of results with different number of columns and column types (SQL Server can do this), the RecordSet object can also jump between each set of results. The RecordSet can also execute its own query/stored procedure (after being attached to a Connection of course) and obtain the returned results. We will discuss this method, but not use it heavily in our examples. Instead we will usually assign a RecordSet object to the result of a Command (or a Connection) object and use the RecordSet to process the rows of results returned by the Command object.

Field Object

The Field object represents a column in a row of results. As there are usually multiple columns returned as part of a SQL query, a RecordSet object will contain as many Field objects as there are columns. A Field object contains information about the column that it represents, such as value of the column, type of column (integer, character, float, money etc.), maximum length of the field, precision etc. We will mostly deal with Fields as part of the RecordSet object in this tutorial.

Command Object

The Command object is typically attached to a Connection object and then used to execute queries and stored procedures or access tables through the Connection. If there are results returned by the Command object (say when you run a SQL query), these are assigned to a RecordSet object, which is then used to iterate through the rows. Most of our examples will use this paradigm. A major feature of the Command object is the ability to use parameters for queries and stored procedures. We will see why this is a good thing in this chapter and why it is better to use this than to execute queries directly via the Connection or RecordSet object.

Parameter Object

A Parameter object is typically attached to a Command object. It is used to pass parameter values to the Command object. A Command object can have multiple Parameter objects attached to it. There are several benefits to using Parameter objects though novice programmers almost never use them. In the next section, we will discuss the reasons and advantages of using Parameter objects.

Chapter 3: What is a Parameter and Why Use It?

Parameters are a model that is used by several interfaces such as DBI, BDE, ODBC etc. ADO also provides for using them. However novice programmers usually don't know about them and never bother to use them at all. We will investigate several reasons why they are such a good thing in this chapter.

Consider the following situation. You need to read several lines from a text file and insert each one into a SQL table. A naive way to do this would be:

1. Open the text file.
2. Read a line from the text file.
3. Prepare a SQL statement using string concatenation:

```
1   sql = "INSERT INTO tablename(column1) VALUES (' " + line + " ')"
```

4. Execute the SQL statement that we prepared above. The database engine validates our statement as legal SQL, compiles it and then executes it.

5. Check if there are any more lines in the file. If there are any more lines, repeat from step 2.

A person who has dabbled with SQL before might note a problem with the above approach. If the line read in step 2 contains a quote character ('), then the SQL string in step 3 will not be formatted correctly. It will have similar problems if the line contains other reserved characters such as %, ?, newlines etc. To prevent this, we will need to rewrite step 3 as follows:

3. Prepare a SQL statement using string functions to format the string properly:

```
sql = "INSERT INTO tablename(column1) VALUES (' " + EscapeSpecialChars(line) + " ')"
```

where EscapeSpecialChars() is a function you write to escape (or remove) the characters that may cause problems. Typically, this function should replace single quote (') with two single quotes ('') and similarly handle %, ? etc. The trouble is that different database engines might have different ideas of what characters are considered special and how they should be escaped.

An additional problem with this approach is the fact that the SQL statement is being altered every time a line is read in. The only change to the SQL statement is the value of the variable "line", while the rest of it stays the same. The novice approach rebuilds the SQL string from scratch each time, which takes up more time. First, you have the overhead of joining strings together to form the SQL statement. Second, the database server will validate and compile the statement each time before executing it. This is a wasteful approach since the only thing that changes is the value of the line variable and there's no need to revalidate or recompile the statement after the first time around. Thirdly, there is the possiblity of SQL Injection, where a malicious person could run their own SQL statements on your server.

A better approach is to use the parameter facility. Modern database servers are capable of taking a SQL statement with parameters and compiling it ahead of time. You can then pass a parameter value to it and execute it each time for new parameter values. This saves time because the database engine does not need to validate/recompile the statement each time. Instead it has the statement validated and compiled and simply puts the parameter value into it each time.

When using parameters, the same problem can be solved as follows:

1. Prepare a SQL statement using parameters and pass this to the database engine ahead of time. The database engine validates the statement as valid SQL, compiles it into bytecode and returns a handle which we can use to access this statement.

2. Open the text file.

3. Read a line from the text file.

4. Execute the SQL statement via the handle we obtained in step 1 and pass the line as a parameter value. The database engine already has the statement validated and compiled, so all it does is substitute the parameter with the value that you pass to it.

5. Check if there are any more lines in the file. If there are any more lines, repeat from step 3.

Since the database engine has already validated/compiled our statement once, it doesn't bother to do it again for each new execution of the statement. When you have a lot of lines, this can result in increased performance. As an additional bonus, step 4 automatically knows to escape characters properly, so you don't have to write your own EscapeSpecialChars() function to do so. This makes SQL injection attacks impossible.

It is possible to prepare a statement that takes multiple parameter arguments too. This allows you to pass values for multiple columns in a statement. You can also assign default values to parameters when creating them. Thus if you don't explicitly set the value of a parameter when executing the SQL statement, it will use the default value for that parameter. This is very handy if you need to insert multiple rows where the values of some columns only change sometimes.

Parameters can also be used to supply values to a stored procedure object. This tutorial will demonstrate usage of parameters to execute both SQL queries as well as stored procedures. As you can see, the use of parameters has several advantages that make them essential for any serious database programmer.

Chapter 4: ADO and Python Basics

As mentioned in a previous chapter, Python can access ADO via a COM interface. To allow Python to work with COM interfaces, you will need Mark Hammond's excellent set of Python for Windows Extensions, in particular his win32com module. If you don't already have

the extensions installed, you should download and install them now. Oh yeah, did I mention you need to be running on a Windows platform already :).

In addition to Mark Hammond's extensions, you may need my list of ADO constants that I put in a Python module called adoconstants.py. This module is released under a BSD style license. This module merely contains a list of constant values that you can use. The list has several commonly used values but is by no means complete.

Once you've downloaded and installed the above, it is time to start exploring the different ADO objects. All the programs presented in the following chapters are assumed to have the following lines at the top of the file:

```
from win32com.client import Dispatch
from adoconstants import *
```

We're using "from adoconstants import *" instead of "import adoconstants" only because it keeps the examples uncluttered.

Another issue is the matter of Late Binding vs. Early Binding in COM objects. This is not ADO specific, but rather an issue of using any COM object with Python. Basically, there are two ways that a Python COM object can access its methods and properties. These two methods are called Late Binding and Early Binding. If a Python COM object uses Late Binding, then every time you access a method or property of the object, it goes through the IDispatch interface to find the method/property, even if it is the same one being called each time. With Early Binding, we let Python know ahead of time what methods and properties are available to an object. This speeds up things significantly, especially inside loops, and the performance gains are actually quite substantial. To enable Early Binding for ADO objects, we need to import the ADO library. To do this:

In PythonWin, go to Tools --> COM Makepy Utility

In the dialog box that pops up, scroll down till you reach Microsoft ActiveX Data Objects Library. If there are multiple versions, simply pick the latest one.

Click on the OK button. The PythonWin environment will freeze for a little bit, while the library is being imported. In a little while, you should see a message on the Interactive Window that says that a file was generated.

You've successfully generated a Python type library for ADO. From now on, Python will automatically use the type library to early-bind any ADO objects.

You can get by without importing the ADO library, but the performance gains are well worth it. Now, let's move on to exploring how to use the different ADO objects with Python.

Chapter 5: The ADO Connection Object

The Connection object is our starting point to obtaining data from a source. All the other ADO objects use the Connection object to point to the data source. Hence, this is the first object that needs to be created. We will first study some of the properties and methods that a Connection object provides and then see how they can be called from Python.

The following table outlines some of the common properties of the Connection object. Most of these properties should be set before a connection is made. The properties control how the connection is made. The bold green text indicates the most useful ones.

Property	Description
Attributes	Sets/returns the attributes.
CommandTimeout	Sets/returns the maximum number of seconds to wait for a command to complete. This value will be inherited by other ADO components that use this connection object, unless they override their value explicitly.
ConnectionString	Sets/returns the parameters used for the connection. This is probably the most important property of them all, as it can be used to set all other properties instead of setting them one at a time. We will cover some examples below.
CursorLocation	Sets/returns the location of the cursor service. This property only applies to certain database engines.
DefaultDatabase	Sets/returns the default database name that the object switches to when it connects to the datasource.

Property	Description
IsolationLevel	Sets/returns the transaction isolation level. Again, this may apply to only certain database engines.
Mode	Sets/returns the access permissions
Provider	Sets/returns the provider name (i.e.) the driver to be used to connect to the DB.
State	Returns the state of the connection object, whether connected or not connected to the datasource.
Version	Returns the version number of the ADO objects

Next, we will study the methods that the Connection object has. As before, the the bold green text indicates the most useful ones.

Property	Description
BeginTrans	Begins a new transaction. Only useful if the datasource actually supports transactions, which not all engines do.
Cancel	Cancels the currently executing statement.
Close	Closes the connection.
CommitTrans	Commits all the changes made in the current transaction and then ends the transaction. You should have called BeginTrans first to start the transaction. Not all engines support transactions.
Execute	Executes a query, stored procedure or a statement specific to the datasource.
Open	Opens a connection to the datasource.
OpenSchema	Returns schema information about the datasource. This is not supported by all database engines, so you may end up with no information if your database doesn't support it.
RollbackTrans	Cancels any changes made in the current transaction and ends the transaction. Only applies if your database engine supports transactions. You should have called BeginTrans first also.

The way to use a Connection object is to first create the object, then set the various properties (depending on your database) and then call Open() to connect to the database. The ConnectionString property is the most important property of them all. Instead of setting individual properties like this:

```
oconn.Provider = "SQLOLEDB"
oconn.CommandTimeout = 60
```

you can instead set all the properties in one shot with the ConnectionString property. All you need to do is separate each argument with a semicolon(;) character.

```
oconn.ConnectionString = "provider=SQLOLEDB; CommandTimeout=60; ... more properties ..."
```

The ADO Connection piggybacks on top of other connection methods such as ODBC, OLE DB, RDS etc. Aside from the ConnectionString though, the other ADO components have no idea of what specific connection method is being used. This is one of the advantages of abstraction that ADO provides to the user.

Now, on to some real Python code. We will assume we're connecting to SQL server for this example using OLEDB and then show some connection strings for other database engines. The code is very simple and the comments actually outweigh the code, so you should have no trouble understanding what is going on here.

```
# First import two useful modules

from win32com.client import Dispatch
from adoconstants import *

```

```
# Create the ADO Connection object via COM.
oConn = Dispatch('ADODB.Connection')

# Now set the connection properties via the ConnectionString
# We're connecting to a SQL Server on 192.168.1.100 using OLEDB.

oConn.ConnectionString = "Provider=SQLOLEDB.1;Data Source=192.168.1.100;" + \
   "uid=my_user_name;pwd=my_password;database=my_database_name"


# Now open the connection
oConn.Open()

# Instead of setting the ConnectionString and then calling Open, it is also
# possible to call the Open method directly and pass the connection string
# as an argument to the method. {i.e.)

# oConn.Open("Provider=SQLOLEDB.1; Data Source=.....")


if oConn.State == adStateOpen:
   # Do something here
   print "We've connected to the database."

   # Execute a stored procedure.
   oConn.Execute("myStoredProcedure")

   # Execute an INSERT statement
   oConn.Execute("INSERT INTO table(col1, col2) VALUES (2, 'Test String')")
 else:
   print "We failed to connect to the database."

 # Close up the connection and unload the COM object
 if oConn.State == adStateOpen: oConn.Close()
 oConn = None

```

That was extremely easy to work with, wasn't it? Now you may not have access to a box running SQL Server, but not to worry. You can make it connect to just about any database engine by just altering the ConnectionString property. The following table illustrates connection strings for some common engines. If your engine is not in the list below, it is fairly easy to find it by using Google and searching for "ADO ConnectionString ".

Database Engine	ConnectionString
DBASE (using ODBC)	Driver={Microsoft dBASE Driver (*.dbf)};DriverID=277;Dbq=C:\path\to\database
Excel (using ODBC)	Driver={Microsoft Excel Driver (*.xls)};DriverID=790;Dbq=C:\path\to\spreadsheet;DefaultDir=C:\path\to\defaultdir
Excel (using OLE DB)	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\Path\To\sheet.xls;

Database Engine	ConnectionString
Access (using ODBC)	Driver={Microsoft Access Driver (*.mdb)};Dbq=C:\path\to\database.mdb;Uid=username;Pwd=password You can also pass additional options -- for example Exclusive=1; sets it to be opened in exclusive mode.
Access (using OLE DB)	**Using standard security** Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\path\to\database.mdb;User Id=username;Password=somepassword; **Using Workgroup security** str="Provider=Microsoft.Jet.OLEDB.4.0;DataSource=database.mdb;JET OLEDB:System Database=mysystem.mdw;" oConn.Open(str, "my_user_name", "my_password")
Firebird	**Remote Database** Provider='LCPI.IBProvider';Data Source='remotehost:C:\path\to\database.fdb';User ID='username';Password='pwd';Auto Commit=true; The above is reported to work for a Firebird database by Edward Diamond (ediamond at water dot ca dot gov). I would presume that it could work on a local Firebird server, simply by removing "remotehost" from the string above. Edward reports that even simple queries don't work without the "Auto Commit" part in the connection string.
MySQL (using ODBC)	**Local Database** Driver={MySQL ODBC 3.51 Driver};Server=localhost;User=username;Password=mypassword;Database=mydatabase; **Remote Database** Driver={MySQL ODBC 3.51 Driver};Server=192.168.1.100;Port=3306;User=username;Password=mypassword;Database=mydatabase; There are more parameters that can be set (for example, Option, which controls several connection properties such as logging, packet size limits etc.) See section 3.3 (Connection Parameters) of theMyODBC manual for more information.
MySQL (using OLE DB)	Provider=MySQLProv;Server=192.168.1.100;Port=3306;User=username;Password=mypassword;Database=mydatabase; If you have the datasource already set up: Provider=MySQLProv;Data Source=name_of_datasource; You will need to download and install MyOleDB first. Last time I checked MyOLEDB was no longer maintained.
Oracle (using ODBC)	Driver={Microsoft ODBC for Oracle};Server=MyOracleServer;Uid=username;Pwd=password See the MSDN library for additional options.
Oracle (using OLE DB)	**Using OLE DB provider from Microsoft** Provider=MSDAORA;Data Source=MyOracleDB;User Id=username;Password=password See the MSDN library for additional options. **Using OLE DB provider from Oracle** Provider=OraOLEDB.Oracle;Data Source=MyOracleDB;User Id=username;Password=password
SQL Server (using ODBC)	**Standard Security** Driver={SQL Server};Server=192.168.1.100;Uid=username;Pwd=password;Database=dbname; **Trusted Connection** Simply add Trusted_Connection=yes to the above string. See MSDN Library for more options.
SQL Server (using OLE DB)	Provider=SQLOLEDB.1;Data Source=192.168.1.100;Uid=username;Pwd=password;Database=dbname; See MSDN Library for more options.

Chapter 6: The ADO RecordSet Object

The RecordSet object is used to iterate through a set of results from a query or a stored procedure. A RecordSet Object must always be assigned a Connection object, where it can retrieve its data from. This is usually done by setting its ActiveConnection property. A complete glossary of all the properties and methods of a RecordSet object is available from the MSDN Library and so we won't repeat all that information here. We will see how to perform some basic selections using the RecordSet object. The following examples assume a MySQL database, but the same code could very well apply to any other database engine (Access, SQL Server, Oracle etc.), merely by changing the ConnectionString property. All the other code can remain the same irrespective of the actual database engine being used.

Opening a Table and Accessing Fields

```
from win32com.client import Dispatch
from adoconstants import *
```

```
3
4
5  # Create Connection object and connect to database.
6  oConn = Dispatch('ADODB.Connection')
7  oConn.ConnectionString = "Driver={MySQL ODBC 3.51 Driver};" + \
8                "Server=192.168.0.50;Port=3306;" + \
9                "User=foobar;Password=bigsecret;Database=mytestdb"
10
11 oConn.Open()
12
13 # Now create a RecordSet object and open a table
14 oRS = Dispatch('ADODB.RecordSet')
15 oRS.ActiveConnection = oConn  # Set the recordset to connect thru oConn
16 oRS.Open("zipcode")       # Open a table called zipcode
17
18 # Can also use oRS.Open("zipcode", oConn) instead of setting ActiveConnection
19
20 while not oRS.EOF:
21   # Access individual fields by name -- all 4 methods are equivalent
22   print oRS.Fields.Item("city").Value, oRS.Fields("state"), \
23      oRS.Fields("zip").Value, oRS.Fields.Item("city")
24
25   # Access individual fields by position -- all 4 methods are equivalent
26   print oRS.Fields.Item(0), oRS.Fields(1), \
27      oRS.Fields(2).Value, oRS.Fields.Item(0).Value
28
29   # Move to the next record in the RecordSet
30   oRS.MoveNext()
31
32 # Close and clean up
33 oRS.Close()
34 oRS = None
35
36 oConn.Close()
37 oConn = None
```

As you can see, the above example is fairly self-explanatory. The next few examples will demonstrate how to execute queries and stored procedures.

Using a RecordSet for Query / Stored Procedure

The code to open a query or a stored procedure is very similar to the example above. All you have to do is replace the line where we open the table above (i.e. oRS.Open("zipcode")) with one of the lines below. The rest of the code can remain the same.

```
1 oRS.Open("SELECT zip, city, state FROM zipcode ORDER BY id")
2 oRS.Open("SELECT zip, city, state FROM zipcode ORDER BY id", oConn)
3
4 # Assuming we have an engine that supports stored procedures which return result sets,
5 # such as SQL Server, we can call a stored proc as well. Here, the name of the stored
6 # proc is assumed to be selZipCode
7
8 oRS.Open("selZipCode")
9 oRS.Open("selZipCode", oConn)
```

Another way to create a RecordSet object is to use the result from an Execute method of a Connection object. The Execute method of the Connection object returns a RecordSet object, if you pass it a query or stored procedure that returns a recordset. If you use this method, you do not need to create a new RecordSet object or assign its ActiveConnection property.

```
# oRS = Dispatch('ADODB.RecordSet')
# oRS.ActiveConnection = oConn

# Execute a query that returns a RecordSet
(oRS, result) = oConn.Execute("SELECT zipcode, city, state FROM zipcode ORDER BY id")

# Execute a stored proc that returns a RecordSet
(oRS, result) = oConn.Execute("selZipCode")

while not oRS.EOF:
    # rest of code
```

The above techniques can be used if the query is a simple one. However, if you have a query where you need to build the string like this:

```
sql = "SELECT zip, city, state FROM zipcode WHERE city = '" + cityvar + "' ORDER BY id"
```

you may run into problems if cityvar has a quote character in it. As we discussed in chapter 3 of this tutorial, the best way to get around this is to use parameters. We will demonstrate how to use parameters using a Command object in the next chapter.

Chapter 7: The ADO Command Object

The Command object is used to perform a query or execute a stored procedure. Thus, it can be used to create, delete, update or select records. If it is used to retrieve data, then the data will be returned via a RecordSet object. One of the major advantages of a Command object is the ability to use parameters. A complete glossary of all the properties and methods of a Command object is available from the MSDN Library and so we won't repeat all that information here. As before, the engine is assumed to be MySQL for this example, but it could easily apply to any other database engine by merely changing the ConnectionString property. The rest of the code can remain the same for any database engine.

Opening a Table and Accessing Fields

```
from win32com.client import Dispatch
from adoconstants import *

# Create Connection object and connect to database.
oConn = Dispatch('ADODB.Connection')

oConn.ConnectionString = "Driver={MySQL ODBC 3.51 Driver};" + \
              "Server=192.168.0.50;Port=3306;" + \
              "User=foobar;Password=bigsecret;Database=mytestdb"

oConn.Open()

# Now create a Command object and open a table
oCmd = Dispatch('ADODB.Command')
oCmd.ActiveConnection = oConn   # Set the Command to connect thru oConn
oCmd.CommandType = adCmdTable  # Set the CommandType to open a table
oCmd.CommandText = "zipcode"   # called zipcode
oCmd.Prepared = True       # Set the Command object to prepare the query ahead of time

# Now execute the command object and collect the results in a recordset
(oRS, result) = oCmd.Execute()

while not oRS.EOF:
   # Access individual fields by name -- all 4 methods are equivalent
```

```
   print oRS.Fields.Item("city").Value, oRS.Fields("state"), \
      oRS.Fields("zip").Value, oRS.Fields.Item("city")

   # Access individual fields by position -- all 4 methods are equivalent
   print oRS.Fields.Item(0), oRS.Fields(1), \
      oRS.Fields(2).Value, oRS.Fields.Item(0).Value

   # Move to the next record in the RecordSet
   oRS.MoveNext()

# Close and clean up
oRS.Close()
oRS = None
oCmd = None
oConn.Close()
oConn = None
```

Nothing very different from the previous page where we opened a table. However, we are only just beginning to explore the Command object capabilities. Instead of opening a table, let's now execute a query. This can be done by just changing two lines in the above code, where we set the CommandType and CommandText properties.

```
oCmd.CommandType = adCmdText   # Set the CommandType to open a query
oCmd.CommandText = "SELECT city, state, zip FROM zipcode"
```

As you can see, the code is still very similar to how we used a RecordSet object. In fact, we could have done this by using the Open() method of a RecordSet object that we used in the previous chapter. Now comes the REALLY BIG advantage of using a Command object to execute queries. Let's say you need to work with a query like this:

```
SELECT city, state, zip FROM zipcode WHERE city = 'some_city_entered_by_user'
```

If you use a RecordSet object, you will need to build a new SQL string each time the user enters a new city name. Also, you will need to perform some checking to see if the string entered by the user contains quotes, percent signs or any other special characters and escape them out accordingly.

However, with a Command object using parameters, you can get around both these problems. Recall in chapter 3, we discussed how parameters can make our queries much more efficient. Now let's put that theory into practice here. We're going to assume a case where the user is prompted to enter two zip codes and the program will return all the places between the two zipcodes.

```
from win32com.client import Dispatch
from adoconstants import *


# Create DB and connect to database.
oConn = Dispatch('ADODB.Connection')
oConn.ConnectionString = "Driver={MySQL ODBC 3.51 Driver};" + \
             "Server=192.168.0.50;Port=3306;" + \
             "User=foobar;Password=bigsecret;Database=mytestdb"
oConn.Open()

# Now create a command object and prepare the query
oCmd = Dispatch('ADODB.Command')
oCmd.ActiveConnection = oConn

# Create a query that accepts two parameters
oCmd.CommandType = adCmdText
```

```
oCmd.CommandText = "SELECT * FROM zipcode WHERE zipcode >= ? AND zipcode <= ?"

# Now create the Parameter objects
oParamZip1 = oCmd.CreateParameter('minzip', adInteger, adParamInput)
oParamZip2 = oCmd.CreateParameter('maxzip', adInteger, adParamInput)
oCmd.Parameters.Append(oParamZip1)
oCmd.Parameters.Append(oParamZip2)

# Request the query to be prepared ahead of time
oCmd.Prepared = True

while 1:
  # Ask the user for input
  min = raw_input("Enter min zip code: ")
  if (min == ""):
    break
  max = raw_input("Enter max zip code: ")

  # Set the parameter values and execute our query
  oParamZip1.Value = min
  oParamZip2.Value = max
  (oRS, result) = oCmd.Execute()

  # Print out the query results
  while not oRS.EOF:
    # Access individual fields by name -- all 4 methods are equivalent
     print oRS.Fields.Item("city").Value, oRS.Fields("state"), \
       oRS.Fields("zip").Value, oRS.Fields.Item("city")

    # Access individual fields by position -- all 4 methods are equivalent
    print oRS.Fields.Item(0), oRS.Fields(1), \
       oRS.Fields(2).Value, oRS.Fields.Item(0).Value

    oRS.MoveNext()

# Close and clean up
oRS.Close()
oRS = None
oCmd = None
oConn.Close()
oConn = None
```

As you can see in the above code, we are asking the user for input and executing the query using the input values as parameters. It is not necessary to rebuild the SQL statement again for different input values. All we have to do is change the values of the two Parameter objects and call the Execute method again. Hence, if there's a need to execute the same query over and over again with different values, using this method provides some significant performance advantages.

Also note that we created two parameters of type adInteger above. Different parameter types require different number of arguments to CreateParameter(). For example, creating a parameter of type adVarChar requires CreateParameter() to be called with four arguments. Details about creating various parameter types are presented in Chapter 10 and you will also encounter them in the next few chapters.

In the next chapter, we will explore how to use the Command object to execute Insert and Update statements.

Chapter 8: More on the Command Object

In the previous chapter, we saw how to use a Command object to perform select queries on a database. We also saw how it was possible to pass parameters to the query and optimize it. In this chapter, we will explore techniques to insert, update and delete data using queries. First we will deal with the quick (albeit dirty and naive) way to insert data into a database. We will write a small program

that allows us to input a person's first name, last name, age and income into a table. We will assume that the first name and last name fields are of type varchar, age is an integer and income is a numeric(6, 2) field. In this example, we will use the Execute method of the Connection object to add new rows to the database. The example assumes a MySQL database engine, but it could easily be modified for just about any database engine (Access, SQL Server, Oracle, PostGres etc.) by merely changing the ConnectionString. The rest of the code remains the same for any database engine.

Quick-N-Dirty Insert Query

```
from win32com.client import Dispatch
from adoconstants import *

# Create Connection object and connect to database.
oConn = Dispatch('ADODB.Connection')
oConn.ConnectionString = "Driver={MySQL ODBC 3.51 Driver};" + \
             "Server=192.168.0.50;Port=3306;" + \
             "User=foobar;Password=bigsecret;Database=mytestdb"

oConn.Open()

# Now ask the user for input
while 1:
  # Ask the user for input

  fname = raw_input("Enter First Name: ")
  if (fname == ""):
    break
  lname = raw_input("Enter Last Name: ")
  age = raw_input("Enter your age: ")
  income = raw_input("Enter your income: ")

  # Now prepare the SQL statement to do the data insertion.
  sql = "INSERT INTO person(first_name, last_name, age, income) " + \
     "VALUES ('" + fname + "', '" + lname + "', " + age + ", " + income + ")"

  # Now execute the SQL statement that we prepared above.
  oConn.Execute(sql)

# Close up and clean up
oConn.Close()
oConn = None
```

At first, the method seems fairly straightforward. The program asks the user for input, then prepares an appropriate SQL statement and executes it. However on closer observation, a few flaws come to light. For one thing, we are rebuilding the SQL statement each time, even though the only thing that has changed are the values of the fields. The second problem is much more serious. If the user were to enter, say, "O'Bannon" for the last name, then the sql statement that we prepared would read something like this:

```
sql = "INSERT INTO person(first_name, last_name, age, income) VALUES ('Roy', 'O'Bannon', 21, 1250.25)"
```

which is clearly the wrong thing, since the apostrophe will cause problems if you try to execute the statement. You'll need to replace a single apostrophe character with a doubled one ('') to make it process correctly. Now let's see how this same job can be done using a Command object and Parameter objects.

Better Insert Query

```
from win32com.client import Dispatch
from adoconstants import *
```

```
3
4  # Create DB and connect to database.
5  oConn = Dispatch('ADODB.Connection')
6
7  oConn.ConnectionString = "Driver={MySQL ODBC 3.51 Driver};" + \
8              "Server=192.168.0.50;Port=3306;" + \
9              "User=foobar;Password=bigsecret;Database=mytestdb"
10 oConn.Open()
11
12 # Now create a command object and prepare the query
13 oCmd = Dispatch('ADODB.Command')
14 oCmd.ActiveConnection = oConn
15
16 # Create a query that accepts four parameters
17 oCmd.CommandType = adCmdText
18 oCmd.CommandText = "INSERT INTO person(first_name, last_name, age, income) " + \
19           "VALUES (?, ?, ?, ?)"
20
21 # Now create the Parameter objects
22 oParam1 = oCmd.CreateParameter('first_name', adVarChar, adParamInput, 50)
23 oParam2 = oCmd.CreateParameter('last_name', adVarChar, adParamInput, 50)
24 oParam3 = oCmd.CreateParameter('name', adInteger, adParamInput)
25 oParam4 = oCmd.CreateParameter('income', adNumeric, adParamInput)
26 oParam4.Precision = 6
27 oParam4.NumericScale = 2
28 oCmd.Parameters.Append(oParam1)
29 oCmd.Parameters.Append(oParam2)
30 oCmd.Parameters.Append(oParam3)
31 oCmd.Parameters.Append(oParam4)
32
33 # Request the query to be prepared ahead of time
34 oCmd.Prepared = True
35
36 while 1:
37   # Ask the user for input
38   fname = raw_input("Enter First Name: ")
39   if (fname == ""):
40     break
41   lname = raw_input("Enter Last Name: ")
42   age = raw_input("Enter your age: ")
43   income = raw_input("Enter your income: ")
44
45   # Set the Parameter values and execute our query
46   oParam1.Value = fname
47   oParam2.Value = lname
48   oParam3.Value = age
49   oParam4.Value = income
50   oCmd.Execute()
51
52 # Close and clean up
53 oCmd = None
54 oConn.Close()
55 oConn = None
```

In the above code, we create a Command object and assign it a SQL query which contains four parameters. We then create four Parameter objects to correspond to the four variables in the SQL statement. Then we prepare the statement, so that the database server knows ahead of time that we intend to execute this statement multiple times. Then we request the user for input, set the appropriate parameter values and execute the statement. This is more efficient because we are not rebuilding the SQL statement each time we get a

new set of values. Also, since we're using Parameter objects, the code automatically escapes any apostrophe characters and other special characters by itself. Hence an input of "O'Bannon" will be accepted and processed correctly. Thus we can get around the two biggest issues of the Quick-N-Dirty method.

Note that in the above code, we created parameters of type adVarChar, adInteger and adNumeric. If you look carefully, you'll notice that CreateParameter() was called with different numbers of arguments depending on the parameter type and also that some parameter types need additional properties to be set. Chapter 10 details how to create different parameter types.

You can also use the same techniques to handle UPDATE and DELETE statements using parameters. For instance, you can use a CommandText like this to update a row:

```
oCmd.CommandText = "UPDATE person SET income = income * 1.5 WHERE age > ? AND age < ?"
```

or delete rows like this:

```
oCmd.CommandText = "DELETE FROM person WHERE age > ?"
```

In the next section we will explore how to call stored procedures with parameters.

Chapter 9: Command Objects and Stored Procedures

In the previous chapter, we saw how to use a Command object to perform insert, update and delete queries on a database. In this chapter we will explore how to use a Command object to execute stored procedures. The assumption here is that you have a database that supports stored procedures (such as SQL Server or Oracle). First we will examine the Quick-N-Dirty method to do things. The example assumes a SQL Server database engine, but it could easily be modified for just about any database engine that supports stored procedures (Oracle, InterBase, Firebird etc.) by merely changing the ConnectionString. The rest of the code remains the same for any database engine. Assuming a SQL Server database engine and a stored procedure called addPerson that takes 4 arguments, a naive way to handle stored procedures would go like this:

Quick-N-Dirty Stored Procedures

```
from win32com.client import Dispatch
from adoconstants import *

# Create Connection object and connect to database.
oConn = Dispatch('ADODB.Connection')
oConn.ConnectionString = "Provider=SQLOLEDB.1;Data Source=192.168.1.100;" + \
  "uid=my_user_name;pwd=my_password;database=my_database_name"

oConn.Open()


# Now ask the user for input
while 1:
  # Ask the user for input
  fname = raw_input("Enter First Name: ")
  if (fname == ""):
    break
  lname = raw_input("Enter Last Name: ")
  age = raw_input("Enter your age: ")
  income = raw_input("Enter your income: ")

  # Now prepare the stored procedure statement.
  sql = "addPerson @first_name = '" + fname + "', " + \
     "@last_name = '" + lname + "', " + \
     "@age = " + age + ", " + \
```

```
      "@income = " + income

  # Now execute the stored proc statement that we prepared above.
  oConn.Execute(sql)

# Close up and clean up
oConn.Close()
oConn = None
```

As in the previous chapter, this naive method suffers from the same two issues -- first, we're building a new statement each time we receive input from the user when the only thing changing is the parameter values. Secondly, a name like "O'Bannon" will still cause problems when we build the SQL statement, unless we escape the characters properly ourselves. Now, let's see how to overcome both problems using a Command object and Parameter objects.

Better Method for Stored Procedures

```
from win32com.client import Dispatch
from adoconstants import *

# Create DB and connect to database.
oConn = Dispatch('ADODB.Connection')
oConn.ConnectionString = "Provider=SQLOLEDB.1;Data Source=192.168.1.100;" + \
  "uid=my_user_name;pwd=my_password;database=my_database_name"

oConn.Open()

# Now create a command object and prepare the query
oCmd = Dispatch('ADODB.Command')
oCmd.ActiveConnection = oConn

# Create a query that accepts four parameters
oCmd.CommandType = adCmdStoredProc
oCmd.CommandText = "addPerson @first_name = ?, @last_name = ?, @age = ?, @income = ?"

# Now create the Parameter objects
oParam1 = oCmd.CreateParameter('@first_name', adVarChar, adParamInput, 50)
oParam2 = oCmd.CreateParameter('@last_name', adVarChar, adParamInput, 50)
oParam3 = oCmd.CreateParameter('@age', adInteger, adParamInput)
oParam4 = oCmd.CreateParameter('@income', adNumeric, adParamInput)
oParam4.Precision = 6
oParam4.NumericScale = 2
oCmd.Parameters.Append(oParam1)
oCmd.Parameters.Append(oParam2)
oCmd.Parameters.Append(oParam3)
oCmd.Parameters.Append(oParam4)

# Request the query to be prepared ahead of time
oCmd.Prepared = True

while 1:
  # Ask the user for input
  fname = raw_input("Enter First Name: ")
  if (fname == ""):
    break
  lname = raw_input("Enter Last Name: ")
  age = raw_input("Enter your age: ")
  income = raw_input("Enter your income: ")
```

```
    # Set the Parameter values and execute our stored procedure
    oParam1.Value = fname
    oParam2.Value = lname
    oParam3.Value = age
    oParam4.Value = income
    oCmd.Execute()

  # Close and clean up
  oCmd = None
  oConn.Close()
  oConn = None
```

In the above code, we create a Command object and assign it a SQL query which contains four parameters. We then create four Parameter objects to correspond to the four stored procedure parameters. Then we prepare the statement so that the database server knows ahead of time that we intend to execute this statement multiple times. Then we request the user for input, set the appropriate parameter values and execute the statement. This is more efficient because we are not rebuilding the SQL statement each time we get a new set of values. Also, since we're using Parameter objects, the code automatically escapes any apostrophe characters and other special characters by itself. Hence an input of "O'Bannon" will be accepted and processed correctly.

Note that in the above code, we created parameters of type adVarChar, adInteger and adNumeric. If you look carefully, you'll notice that CreateParameter() was called with different numbers of arguments depending on the parameter type and also that some parameter types need additional properties to be set. Chapter 10 details how to create different parameter types.

Command Objects, Stored Procedures and RecordSets

Now we will examine how to execute a stored procedure that accepts parameters and returns a RecordSet, using a Command object. We already saw how to do this with a Connection object back when we examined the RecordSet object. This method is superior since it accepts Parameters.

```
from win32com.client import Dispatch
from adoconstants import *


 # Create DB and connect to database.
 oConn = Dispatch('ADODB.Connection')
 oConn.ConnectionString = "Provider=SQLOLEDB.1;Data Source=192.168.1.100;" + \
   "uid=my_user_name;pwd=my_password;database=my_database_name"

 oConn.Open()

 # Now create a command object and prepare the query
 oCmd = Dispatch('ADODB.Command')
 oCmd.ActiveConnection = oConn


 # Create a query that accepts a parameter
 oCmd.CommandType = adCmdStoredProc
 oCmd.CommandText = "selPersons @age = ?"

 # Now create the Parameter object
 oParam1 = oCmd.CreateParameter('@age', adInteger, adParamInput)
 oCmd.Parameters.Append(oParam1)

 # Request the query to be prepared ahead of time
 oCmd.Prepared = True

 while 1:
```

```
  age = raw_input("Enter your age: ")
  if (age == ""):
    break

  # Set the Parameter values and execute our stored procedure
  oParam1.Value = age
  (oRS, status) = oCmd.Execute()

  # Print out the query results
  while not oRS.EOF:
    print oRS.Fields("first_name"), oRS.Fields("last_name")

  oRS.MoveNext()


  oRS.Close()
  oRS = None

# Close and clean up
oCmd = None
oConn.Close()
oConn = None
```

TECH NOTE: On SQL Server, it may happen that your stored procedure executes fine from the query analyzer and returns a result set, but doesn't work when you try to use it from ADO. Instead, it may give you an error that oRS.EOF cannot be evaluated on a closed object. If this happens, you need to add the line: SET NOCOUNT ON at the top of your procedure and SET NOCOUNT OFF at the bottom of your procedure. If you don't have these two lines, the program will complain that oRS is not open. The author spent a long time trying to figure out where the bug lay in his code, before a little googling found the true solution. This is an issue with ADO and SQL Server that is not specific to using Python alone.

In the previous few examples, we've created different parameter types. The next chapter will merely revisit how to create different parameter types, so that they are all in one page.

Chapter 10: Creating Parameters

In the previous chapters, we created various types of parameter objects (i.e.) varchar, integer etc. If you read the code carefully, you'll notice some parameter objects take more arguments than others. This page details how to create some common parameter object types.

Creating ADO Parameters of Various Types

```
from win32com.client import Dispatch
from adoconstants import *

# We assume oCmd is an already created ADODB.Command object

# Creating a parameter of type integer
oIntParam = oCmd.CreateParameter('intparam', adInteger, adParamInput)

# Creating a parameter of type varchar (assuming max size is 20 chars)
oStrParam = oCmd.CreateParameter('stringparam', adVarChar, adParamInput, 20)

# Another way to do the same thing.
oStrParam = oCmd.CreateParameter('stringparam', adVarChar, adParamInput)
oStrParam.Size = 20

# Creating a parameter of type char (assuming max size is 30 chars)
oStrParam = oCmd.CreateParameter('stringparam', adChar, adParamInput, 30)
```

```
18
19  # Another way to do the above
20  oStrParam = oCmd.CreateParameter('stringparam', adChar, adParamInput)
21  oStrParam.Size = 30
22
23  # Creating a parameter of type widechar (assuming max size is 20 chars)
24  oStrParam = oCmd.CreateParameter('stringparam', adWChar, adParamInput, 20)
25
26  # Creating a parameter of type wide varchar (assuming max size is 30 chars)
27  # widechar and wide varchar are suitable for holding Unicode characters.
28  oStrParam = oCmd.CreateParameter('stringparam', adVarWChar, adParamInput, 30)
29
30  # Another way to do the above
31  oStrParam = oCmd.CreateParameter('stringparam', adVarWChar, adParamInput)
32  oStrParam.Size = 30
33
34  # Creating a parameter of type DBDate
35  oDateParam = oCmd.CreateParameter('dateparam', adDBDate, adParamInput)
36
37  # Creating a parameter of type numeric (assuming field is numeric(6,2))
38  oNumParam = oCmd.CreateParameter('numparam', adNumeric, adParamInput)
39  oNumParam.Precision = 6
40  oNumParam.NumericScale = 2
```

As you can see in the above code, different parameter types take different arguments or may need certain other attributes (such as Size, Precision, NumericScale etc.) to be set. If you don't set all the attributes for a parameter correctly, then when you try to Append() it to the command object (or Execute() the command object):

```
1   oCmd.Parameters.Append(oStrParam)
```

python will throw ugly COM error messages like the following:

```
1  com_error: (-2147352567, 'Exception occurred.', (0, 'ADODB.Parameters',
2  'Parameter object is improperly defined. Inconsistent or incomplete information was provided.',
3  'C:\WINNT\HELP\ADO270.CHM', 1240657, -2146824580), None)
4
5  com_error: (-2147352567, 'Exception occurred.', (0, 'Microsoft OLE DB Provider for SQL Server',
6  'The precision is invalid.', None, 0, -2147467259), None)
```

If you see these types of messages, you now know what the problem is.

This brings the tutorial to an end. We have covered the usage of some of the common ADO objects and how to access them from Python. All that remains is to post a few handy links and credits for this article.

Appendix: Links and Credits

Useful Links

The following links contain useful information and were instrumental in the writing of this tutorial. Most of them contain additional information that wasn't covered here and it'll do the reader a world of good to actually study them. Official Python Website Python for Windows Extensions adoconstants.py -- my own module containing handy ADO constants MSDN reference for ADO

Credits

The author would like to thank the following people: Suzanne for inspiration and always being there for me. :) A. Vivaldi, J.S. Bach, C.P.E. Bach, J. Pachelbel, C. Orff, W.A. Mozart, Jimi Hendrix, The Scorpions, Iron Maiden, Motorhead, Rainbow, Helloween, Mago de Oz, Blind Guardian, Rhapsody, Deep Purple, Judas Priest, Black Sabbath, Led Zeppelin, Pink Floyd -- this list goes on. Iron Chef, Most Extreme Elimination Challenge and P.G. Wodehouse for the great laughs. Suzanne, Grim Archon, MasterChief, netytan and shahenshah for their excellent proof-reading. Joe Nguyen for inspiring the chapter on Creating Parameters. Edward Diamond for the Firebird database connection string. The echarcha crew for their feedback. Len Remmerswaal for suggesting that I add prevention of SQL injections as another reason to use parameters

Article is from : http://www.mayukhbose.com/python/ado/index.php