# Quick Start to Server side COM and Python

## Introduction

This documents how to quickly start implementing COM objects in Python. It is not a thorough discussion of the COM system, or of the concepts introduced by COM.

For more details information on Python and COM, please see the [COM Tutorial given by Greg Stein and Mark Hammond at SPAM 6 (HTML format)](http://www.python.org/windows/win32com/COMTutorial/index.htm) or download the same tutorial [in PowerPoint format.](http://www.python.org/windows/win32com/COMTutorial.ppt)

For information on using external COM objects from Python, please see [a Quick Start to Client side COM and Python](http://docs.google.com/QuickStartClientCom.html).

In this document we discuss the [core functionality](#_gjdgxs), [registering the server](#_1fob9te), [testing the class](#_3znysh7), [debugging the class](#_2et92p0), [exception handling](#_tyjcwt) and [server policies](#_3dy6vkm) (phew!)

## Implement the core functionality

### Implement a stand-alone Python class with your functionality

class HelloWorld:

def \_\_init\_\_(self):

self.softspace = 1

self.noCalls = 0

def Hello(self, who):

self.noCalls = self.noCalls + 1

# insert "softspace" number of spaces

return "Hello" + " " \* self.softspace + who

This is obviously a very simple server. In particular, custom error handling would be needed for a production class server. In addition, there are some contrived properties just for demonstration purposes.

### Make Unicode concessions

At this stage, Python and Unicode don’t really work well together. All strings which come from COM will actually be Unicode objects rather than string objects.

To make this code work in a COM environment, the last line of the "Hello" method must become:

return "Hello" + " " \* self.softspace + str(who)

Note the conversion of the "who" to "str(who)". This forces the Unicode object into a native Python string object.

For details on how to debug COM Servers to find this sort of error, please see [debugging the class](#_2et92p0)

### Annotate the class with win32com specific attributes

This is not a complete list of names, simply a list of properties used by this sample.

| **Property Name** | **Description** |
| --- | --- |
| \_public\_methods\_ | List of all method names exposed to remote COM clients |
| \_public\_attrs\_ | List of all attribute names exposed to remote COM clients |
| \_readonly\_attrs\_ | List of all attributes which can be accessed, but not set. |

We change the class header to become:

class HelloWorld:

\_public\_methods\_ = ['Hello']

\_public\_attrs\_ = ['softspace', 'noCalls']

\_readonly\_attrs\_ = ['noCalls']

def \_\_init\_\_(self):

[Same from here…]

### Registering and assigning a CLSID for the object

COM requires that all objects use a unique CLSID and be registered under a "user friendly" name. This documents the process.

#### Generating the CLSID

Microsoft Visual C++ comes with various tools for generating CLSID's, which are quite suitable. Alternatively, the pythoncom module exports the function CreateGuid() to generate these identifiers.

>>> import pythoncom

>>> print(pythoncom.CreateGuid())

{7CC9F362-486D-11D1-BB48-0000E838A65F}

Obviously the GUID that you get will be different than that displayed here.

#### Preparing for registration of the Class

The win32com package allows yet more annotations to be applied to a class, allowing registration to be effected with 2 lines in your source file. The registration annotations used by this sample are:

| **Property Name** | **Description** |
| --- | --- |
| \_reg\_clsid\_ | The CLSID of the COM object |
| \_reg\_progid\_ | The "program ID", or Name, of the COM Server. This is the name the user usually uses to instantiate the object |
| \_reg\_desc\_ | Optional: The description of the COM Server. Used primarily for COM browsers. If not specified, the \_reg\_progid\_ is used as the description. |
| \_reg\_class\_spec\_ | Optional: A string which represents how Python can create the class instance. The string is of format  [package.subpackage.]module.class  The portion up to the class name must be valid for Python to "import", and the class portion must be a valid attribute in the specified class.  This is optional from build 124 of Pythoncom., and has been removed from this sample. |
| \_reg\_remove\_keys\_ | Optional: A list of tuples of extra registry keys to be removed when uninstalling the server. Each tuple is of format ("key", root), where key is a string, and root is one of the win32con.HKEY\_\* constants (this item is optional, defaulting to HKEY\_CLASSES\_ROOT) |

Note there are quite a few other keys available. Also note that these annotations are *not* required - they just make registration simple. Helper functions in the module win32com.server.register allow you to explicitly specify each of these attributes without attaching them to the class.

The header of our class now becomes:

class HelloWorld:

\_reg\_clsid\_ = "{7CC9F362-486D-11D1-BB48-0000E838A65F}"

\_reg\_desc\_ = "Python Test COM Server"

\_reg\_progid\_ = "Python.TestServer"

\_public\_methods\_ = ['Hello']

[same from here]

#### Registering the Class

The idiom that most Python COM Servers use is that they register themselves when run as a script (ie, when executed from the command line.) Thus the standard "if \_\_name\_\_=='\_\_main\_\_\_':" technique works well.

win32com.server.register contains a number of helper functions. The easiest to use is "UseCommandLine".

Registration becomes as simple as:

if \_\_name\_\_=='\_\_main\_\_':

# ni only for 1.4!

import ni, win32com.server.register

win32com.server.register.UseCommandLine(HelloWorld)

Running the script will register our test server.

## Testing our Class

For the purposes of this demonstration, we will test the class using Visual Basic. This code should run under any version of Visual Basic, including VBA found in Microsoft Office. Any COM compliant package could be used alternatively. VB has been used just to prove there is no "smoke and mirrors. For information on how to test the server using Python, please see the [Quick Start to Client side COM](http://docs.google.com/QuickStartClientCom.html) documentation.

This is not a tutorial in VB. The code is just presented! Run it, and it will work!

## Debugging the COM Server

When things go wrong in COM Servers, there is often nowhere useful for the Python traceback to go, even if such a traceback is generated.

Rather than discuss how it works, I will just present the procedure to debug your server:

**To register a debug version of your class**, run the script (as above) but pass in a "--debug" parameter. Eg, for the server above, use the command line "testcomserver.py --debug".

**To see the debug output generated** (and any print statements you may choose to add!) you can simply select the "Remote Debug Trace Collector" from the Pythonwin Tools menu, or run the script "win32traceutil.py" from Windows Explorer or a Command Prompt.

## Exception Handling

Servers need to be able to provide exception information to their client. In some cases, it may be a simple return code (such as E\_NOTIMPLEMENTED), but often it can contain much richer information, describing the error on detail, and even a help file and topic where more information can be found.

We use Python class based exceptions to provide this information. The COM framework will examine the exception, and look for certain known attributes. These attributes will be copied across to the COM exception, and passed back to the client.

The following attributes are supported, and correspond to the equivalent entry in the COM Exception structure:

scode, code, description, source, helpfile and helpcontext

To make working with exceptions easier, there is a helper module "win32com.server.exception.py", which defines a single class. An example of its usage would be:

raise COMException(desc="Must be a string",scode=winerror.E\_INVALIDARG,helpfile="myhelp.hlp",...)

(Note the COMException class supports (and translates) "desc" as a shortcut for "description", but the framework requires "description")

## Server Policies

This is information about how it all hangs together. The casual COM author need not know this.

Whenever a Python Server needs to be created, the C++ framework first instantiates a "policy" object. This "policy" object is the gatekeeper for the COM Server - it is responsible for creating the underlying Python object that is the server (ie, your object), and also for translating the underlying COM requests for the object.

This policy object handles all of the underlying COM functionality. For example, COM requires all methods and properties to have unique numeric ID's associated with them. The policy object manages the creation of these ID's for the underlying Python methods and attributes. Similarly, when the client whishes to call a method with ID 123, the policy object translates this back to the actual method, and makes the call.

It should be noted that the operation of the "policy" object could be dictated by the Python object - the policy object has many defaults, but the actual Python class can always dictate its operation.

### Default Policy attributes

The default policy object has a few special attributes that define who the object is exposed to COM. The example above shows the \_public\_methods attribute, but this section describes all such attributes in detail.

##### \_public\_methods\_

Required list of strings, containing the names of all methods to be exposed to COM. It is possible this will be enhanced in the future (eg, possibly '\*' will be recognised to say all methods, or some other ideas…)

##### \_public\_attrs\_

Optional list of strings containing all attribute names to be exposed, both for reading and writing. The attribute names must be valid instance variables.

##### \_readonly\_attrs\_

Optional list of strings defining the name of attributes exposed read-only.

##### \_com\_interfaces\_

Optional list of IIDs exposed by this object. If this attribute is missing, IID\_IDispatch is assumed (ie, if not supplied, the COM object will be created as a normal Automation object).

and actual instance attributes:

\_dynamic\_ : optional method

\_value\_ : optional attribute

\_query\_interface\_ : optional method

\_NewEnum : optional method

\_Evaluate : optional method