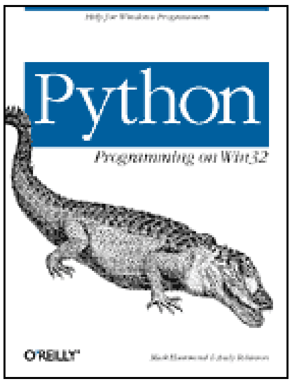


Python Programming on Win32

Help for Windows Programmers

By Mark Hammond & Andy Robinson
1st Edition January 2000
1-56592-621-8, Order Number: 6218
672 pages, $34.95

Chapter 12
Advanced Python and COM

In *Chapter 5, Introduction to COM*, we presented some basic material about Python and COM. If you have never used Python and COM together or are unfamiliar with basic COM concepts, you should review that chapter before continuing here.

In this chapter we take a more technical look at COM and using Python from COM. We initially provide a discussion of COM itself and how it works; an understanding of which is necessary if you need to use advanced features of COM from Python. We then look at using COM objects from Python in more detail and finish with an in-depth discussion of implementing COM objects using Python.

Advanced COM

In order to fully understand Python and COM, it is necessary to understand COM itself. Although Python hides many of the implementation details, understanding these details makes working with Python and COM much easier.

If you want to see how to use Python to control COM objects such as Microsoft Word or Excel, you can jump directly to the section "Using Automation Objects from Python."

Interfaces and Objects

COM makes a clear distinction between *interfaces* and *objects*. An interface describes certain functionality, while an object implements that functionality (that is, implements the interface). An interface describes how an object is to behave, while the object itself implements the behavior. For example, COM defines an `IStream` interface, a generic interface for reading and writing, in a manner similar to a file. Although COM defines the `IStream` interface, it's the responsibility of objects to implement the interface; thus, you may have an object that implements the `IStream` interface writing to and from files or an object implementing the `IStream` interface using sockets, and so forth. This is a huge advantage to users of these interfaces, because you can code to the `IStream` interface, and your code works regardless of whether your data goes to a file or out over a socket. Each COM interface has a unique 128-bit GUID known as an interface ID (IID).

An interface defines a series of methods: interfaces can't have properties. An interface is defined in terms of a C++ `vtable`. Highly experienced C++ programmers will know that a `vtable` implements virtual methods in C++.

Just as with C++, COM allows one interface to derive from, or extend, another interface; in fact, COM explicitly requires it. COM defines an interface known as `IUnknown`, which is the root (or base) of all COM interfaces; that is, all COM interfaces explicitly support the `IUnknown` interface. `IUnknown` is a simple interface defining only three methods: `AddRef()`, `Release()`, and `QueryInterface()`. `AddRef()` and `Release()` manage object lifetimes; a reference counting technique is used so a particular object knows when it is no longer needed. The Python COM framework manages this behind the scenes for you, so these will not be discussed further. `QueryInterface()` allows an object to return a specific interface, given that interface's unique IID. Thus, regardless of the object you have, you can always call its `QueryInterface()` method to obtain a new interface, such as `IStream`.

COM also defines a standard technique for identifying and creating objects themselves. Each object class is identified by a class ID (CLSID, also a GUID) that exposes interfaces, each identified by an IID. Thus, there are a number of identifiers associated with every COM object: the CLSID identifying the class that provides the object, and a series of IIDs for each interface the object supports. Each object supports at least two interfaces, the `IUnknown` interface as described previously, and some useful interface (such as `IStream`) that allows the object to perform its task.

Objects may also register a program ID, or ProgID as well as a CLSID. A ProgID is a string describing the object, suitable for use by humans. When you need to create a particular object, it's usually more convenient to use the ProgID rather than the CLSID. There is no guarantee that ProgIDs will be unique on a given system; you should choose the names of your objects carefully to avoid conflicts with other objects. For example, the Microsoft Excel object has a ProgID of `Excel.Application`.

The IDispatch Interface

The COM architecture works well for languages such as C++, where the methods you need to use are known beforehand (i.e., at compile time). You create an object using the standard COM techniques, then perform a `QueryInterface()` on the object for a particular interface. Once you have the interface, you can make calls on its methods. This architecture does have some drawbacks, notably:

- There is support for methods, but no support for properties. In many cases, properties would simplify the object model you are attempting to publish.
- It doesn't work as well when using higher-level languages than C++. There may be no compile-time step involved at all. The language in use may not support using the **.IDL** or **.H** files necessary to obtain the definition of these interfaces.

COM defines the `IDispatch` interface specifically to meet the requirements of these higher-level languages. The `IDispatch` interface allows an object to expose an object model (complete with methods and properties) and allows the user of the object to determine the methods and properties available at runtime. This means the methods or properties you need to call can be determined when you need to call them, rather than requiring them to be predefined. You should note that the object model exposed using `IDispatch` is quite distinct from the `IDispatch` interface itself; `IDispatch` is a COM interface that allows an arbitrary object model to be exposed. In other words, `IDispatch` is not the object model but is the mechanism that allows an object model to be exposed.

There are two methods `IDispatch` defines for this purpose. The first is `GetIDsOfNames()`; it allows you to ask an object "do you have a method/property named `foo` ?" If the object does have such an attribute, it returns an integer ID for the method or property. The method `Invoke()` performs the actual operation on the object--that is, either calling the method `foo`, or getting or setting a property named `foo`. The `Invoke()` method is passed the integer ID obtained from `GetIDsOfNames()`, as well as any parameters for the function or property.

In almost all languages, you don't need to use the `IDispatch` interface; your language uses `IDispatch` behind the scenes to present a natural model. For example, we'll see later that when you execute code in VB, Python, Delphi, and so forth similar to:

```
    workbook = excel.Workbooks.Add()
```

behind the scenes, there is pseudo-code similar to:

```
    propertyId = excel->GetIDsOfNames("Workbook")
    newObject = excel->Invoke(propertyId, DISPATCH_PROPERTYGET)
    methodId = newObject->GetIDsOfNames("Add")
    result = newObject->Invoke(methodId, DISPATCH_METHOD)
```

The final piece of this puzzle relates to how the arguments and results are passed around. For this purpose, COM defines a `VARIANT` data structure. A `VARIANT` is defined as a self-describing C++ union and allows a wide variety of common data-types to be passed. To create a `VARIANT`, indicate the type of data you wish to pass and set the value. When you need to use a `VARIANT` passed by someone else, first query the type of data it holds and obtain the data. If the type of the data doesn't work for you, you can either attempt a conversion or reject the call returning the appropriate error code. This implies that type checking of the parameters and results can happen only at runtime (although many tools can take advantage of type information provided by the object to flag such errors at compile-time). As with the `IDispatch` interface itself, most high-level languages hide the details of the `VARIANT` and use them invisibly behind the scenes.

Objects that expose an `IDispatch` interface to support method calls and property references are also known as *automation objects*.

Late- Versus Early-Bound IDispatch

The process described for `IDispatch` has one obvious flaw: it seems highly inefficient, and it is! In many cases, the inefficiency isn't important; the objects you need to call will often take longer to do their thing than it took to make the call.

Programs or languages that use `IDispatch` in the manner described are known as *late-bound*, because the binding of objects to methods or properties is done at the last possible moment, as the call or property reference is made.

There is, however, a technique automation objects use to publish their object model in a type library. Type libraries define a set of interfaces a program can use to determine both the methods and properties themselves, and other useful information, such as the type of the parameters or return values. Languages or environments may be capable of using this information at compile-time to provide a better interface to the objects. The key benefits of knowing this information before it's used are:

- The `GetIDsOfNames()` step described previously can be removed, as the type information includes the integer ID of each method or property.
- Better type checking can be performed.

Languages that use the `IDispatch` interface after consulting type information are known as *early-bound*.

Most COM-aware languages, including Visual Basic, Delphi, and Python have techniques that allow the programmer to choose between the binding models. Later in this chapter we discuss the differences when using Python.

Using or Implementing Objects

There is a clear distinction between using COM objects and implementing COM objects. When you use a COM object, you make method calls on an object provided externally. When you implement a COM object, you publish an object with a number of interfaces external clients can use.

This distinction is just as true for the `IDispatch` interface; programs that use an `IDispatch` object must call the `GetIDsOfNames()` and `Invoke()` methods to perform method calls or property reference. Objects that wish to allow themselves to be called via `IDispatch` must implement the `GetIDsOfNames()` and `Invoke()` methods, providing the logic for translating between names and IDs, and so forth.

In the PythonCOM world, this distinction is known as client- and server-side COM. Python programs that need to use COM interfaces use client-side COM, while Python programs that implement COM interfaces use server-side COM.

InProc Versus LocalServer Versus RemoteServer

COM objects can be implemented either in Windows DLLs or in separate Windows processes via an EXE.

Objects implemented in DLLs are loaded into the process of the calling object. For example, if your program creates an object implemented in a DLL, that object's DLL is loaded into your process, and the object is used directly from the DLL. These objects are known as *InProc objects*.

Objects implemented in their own process, obviously, use their own process. If your program creates a COM object implemented in an EXE, COM automatically starts the process for the object (if not already running) and manages the plumbing between the two processes. Objects implemented in an EXE that run on the local machine are known as `LocalServer` objects, while objects implemented in an EXE that run on a remote machine are known as `RemoteServer` objects. We discuss `RemoteServer` objects in the later section "Python and DCOM."

These options are not mutually exclusive; any object can be registered so that it runs in either, all, or any combination of these.

In most cases, you don't need to be aware of this COM implementation detail. You can simply create an object and exactly how that object is created is managed for you. There are, however, some instances where being able to explicitly control this behavior is to your advantage.

Python and COM support `InProc`, `LocalServer`, and `RemoteServer` objects, as discussed throughout this chapter.

Python and COM

The interface between Python and COM consists of two discrete parts: the `pythoncom` Python extension module and the `win32com` Python package. Collectively, they are known as PythonCOM.

The `pythoncom` module is primarily responsible for exposing raw COM interfaces to Python. For many of the standard COM interfaces, such as `IStream` or `IDispatch`, there is an equivalent Python object that exposes the interface, in this example, a `PyIStream` and `PyIDispatch` object. These objects expose the same methods as the native COM interfaces they represent, and like COM interfaces, do not support properties. The `pythoncom` module also exposes a number of COM-related functions and constants.

The `win32com` package is a set of Python source files that use the `pythoncom` module to provide additional services to the Python programmer. As in most Python packages, `win32com` has a number of subpackages; `win32com.client` is concerned with supporting client-side COM (i.e., helping to call COM interfaces), and `win32com.server` is concerned with helping Python programs use server-side COM (i.e., implement COM interfaces). Each subpackage contains a set of Python modules that perform various tasks.

Using Automation Objects from Python

As we discussed previously, automation objects are COM objects that expose methods and properties using the `IDispatch` interface. So how do we use these objects from Python? The `win32com.client` package contains a number of modules to provide access to automation objects. This package supports both late and early bindings, as we will discuss.

To use an `IDispatch`-based COM object, use the method `win32com.client.Dispatch()`. This method takes as its first parameter the ProgID or CLSID of the object you wish to create. If you read the documentation for Microsoft Excel, you'll find the ProgID for Excel is `Excel.Application`, so to create an object that interfaces to Excel, use the following code:

```
>>> import win32com.client
>>> xl = win32com.client.Dispatch("Excel.Application")
>>>
```

`xl` is now an object representing Excel. The Excel documentation also says that a boolean property named `Visible` is available, so you can set that with this code:

```
>>> xl.Visible = 1
>>>
```

Late-Bound Automation

Late-bound automation means that the language doesn't have advance knowledge of the properties and methods available for the object. When a property or method is referenced, the object is queried for the property or the method, and if the query succeeds, the call can be made. For example, when the language sees code such as:

```
xl.Visible = 1
```

the language first queries the `xl` object to determine if there is a property named `Visible`, and if so, asks the object to set the value to 1.

By default, the `win32com.client` package uses late-bound automation when using objects. In the examples we've seen so far, the `win32com.client` package has determined the `Visible` property is available as you attempt to use it. In the parlance of PythonCOM, this is known as *dynamic dispatch*.

If you look at the object, Python responds with:

```
>>> `xl`
<COMObject Excel.Application>
```

This says there's a COM object named `Excel.Application`. Python knows the name `Excel.Application` from the ProgID that created the object.

Early-Bound Automation

The PythonCOM package can also use early binding for COM objects. This means that the information about the object model (i.e., the properties and methods available for an object) is determined in advance from type information supplied by the object.

Python uses the MakePy utility to support early-bound automation. MakePy is a utility written in Python that uses a COM type library to generate Python source code supporting the interface. Once you use the MakePy utility, early binding for the objects is automatically supported; there's no need to do anything special to take advantage of the early binding.

There are a number of good reasons to use MakePy:

- The Python interface to automation objects is faster for objects supported by a MakePy module.
- Any constants defined by the type library are made available to the Python program. We discuss COM constants in more detail later in the chapter.
- There is much better support for advanced parameter types, specifically, parameters declared by COM as BYREF can be used only with MakePy-supported objects. We discuss passing parameters later in the chapter.

And there are a few reasons to avoid MakePy:

- Using a MakePy-supported object means you must run MakePy before code that requires it can be used. Although this step can be automated (i.e., made part of your program), you may choose to avoid it.
- The MakePy-generated files can be huge. The file generated for Microsoft Excel is around 800 KB, a large Python source file by anyone's standards. The time taken to generate a file of this size, and subsequently have Python compile it, can be quite large (although it's worth noting Python can then import the final *.pyc* file quickly).

Running MakePy

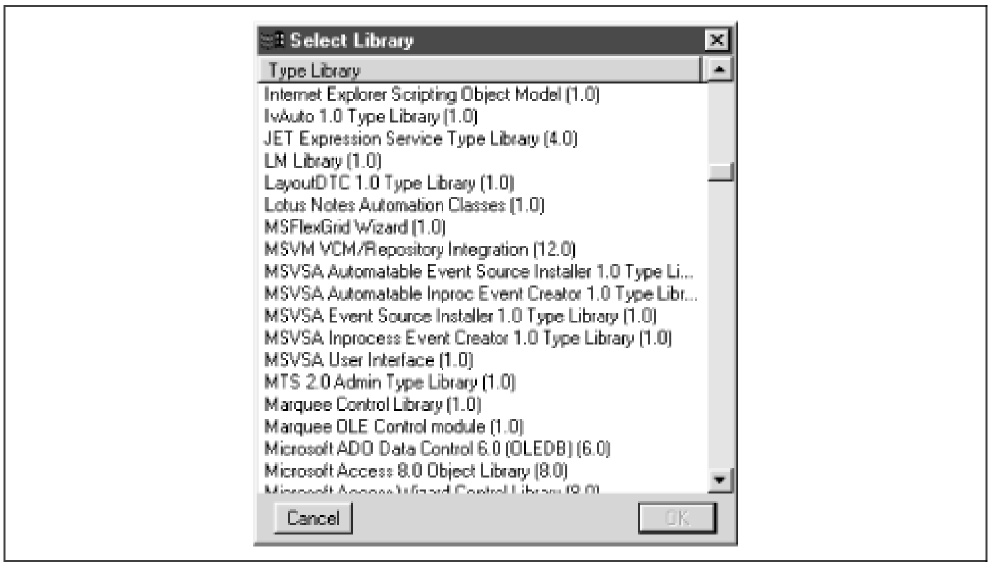
MakePy is a normal Python module that lives in the *win32com\client* directory of the PythonCOM package. There are two ways to run this script:

- Start PythonWin, and from the Tools menu, select the item COM Makepy utility.
- Using Windows Explorer, locate the client subdirectory under the main *win32com* directory and double-click the file *makepy.py*.

In both cases, you are presented with a list of objects MakePy can use to support early binding.

We will try this out, continuing our example of using Microsoft Excel. Let's start PythonWin, and select the COM Makepy utility from the Tools menu. You are then presented with a list that looks similar to that shown in Figure 12-1.

Figure 12-1.An example list of objects presented by MakePy



Internet Explorer Scripting Object Model (1.0)
IvAuto 1.0 Type Library (1.0)
JET Expression Service Type Library (4.0)
LM Library (1.0)
LayoutDTC 1.0 Type Library (1.0)
Lotus Notes Automation Classes (1.0)
MSFlexGrid Wizard (1.0)
MSVM VCM/Repository Integration (12.0)
MSVSA Automatable Event Source Installer 1.0 Type Li...
MSVSA Automatable Inproc Event Creator 1.0 Type Libr...
MSVSA Event Source Installer 1.0 Type Library (1.0)
MSVSA Inprocess Event Creator 1.0 Type Library (1.0)
MSVSA User Interface (1.0)
MTS 2.0 Admin Type Library (1.0)
Marquee Control Library (1.0)
Marquee OLE Control module (1.0)
Microsoft ADO Data Control 6.0 (OLEDB) (6.0)
Microsoft Access 8.0 Object Library (8.0)

The exact contents of the list depends on the software you have installed on your PC. Scroll down until you find the entry Microsoft Excel 8.0 Object Library (1.2, or the entry that represents the version of Excel you have installed) and press Enter. You should see a progress bar displayed as MakePy does its thing, and when complete, you should see a message in the PythonWin interactive window:

```
Generating to c:\Program Files\Python\win32com\gen_py\00020813-0000-0000-C000-00
```

Your first reaction may be one of horror: how are you supposed to use a filename that looks like that? The good news is that you don't need to; just use PythonCOM as normal, but from now on, all references to the Excel object model use the early binding features generated by MakePy.

Now we have created MakePy support; let's see it in action. You can either use the existing PythonWin session, or start a new session and use the same code used earlier to create the `Excel.Application` object:

```
>>> import win32com.client
>>> xl=win32com.client.Dispatch("Excel.Application")
>>>
```

And you can still set the `Visible` property:

```
>>> xl.Visible=1
>>>
```

At this stage, the only difference is when you print the `xl` object:

```
>>> `xl`
<win32com.gen_py.Microsoft Excel 8.0 Object Library._Application>
>>>
```

If you compare this with the output Python presented in the previous example, note that Python knows more about the object; it has referenced the name of the type library (Microsoft Excel 8.0 Object Library) and the name of the object as defined by Excel itself (`_Application`).

How MakePy works

In most cases, you don't need to know how MakePy works, but in certain cases, particularly when tracking down problems, it is handy to know.

The `makepy` module generates Python source code into a standard **.py** source file. The items in this file may include:

- A standard Python class for each automation object included in the type library
- A set of constants exposed by the type library

The Python class has one method for each of the methods defined by the object and a list of properties supported by the object. Let's take a look at some generated code.

Let's open the source file you generated previously for Microsoft Excel. The simplest way to open this file is to copy the name of the file from the PythonWin interactive window, then paste it into the File Open dialog of your favorite editor.

To find the class definition for the Excel `_Application` object, you can perform a search for `class _Application`, and locate code similar to this:

```
class _Application(DispatchBaseClass):
  CLSID = pythoncom.MakeIID('{000208D5-0000-0000-C000-000000000046}')
  def ActivateMicrosoftApp(self, Index=defaultNamedNotOptArg):
    return self._ApplyTypes_(0x447, 1, (24, 0), ((3, 1),), \
                         'ActivateMicrosoftApp', None, Index)

  def AddChartAutoFormat(self, Chart=defaultNamedNotOptArg, \
                           Name=defaultNamedNotOptArg, \
                           Description=defaultNamedOptArg):
    return self._ApplyTypes_(0xd8, 1, (24, 0), ((12, 1), (8, 1), (12, 17)),\
                      'AddChartAutoFormat', None, Chart, Name, Description)
```

There are many more methods. Each method includes the name of each parameter (including a default value). You will notice the series of magic numbers passed to the `_ApplyTypes_()` method; these describe the types of the parameters and are used by the PythonCOM framework to correctly translate the Python objects to the required `VARIANT` type.

Each class also has a list of properties available for the object. These properties also have cryptic type information similar to the methods, so properties also benefit from the increased knowledge of the parameters.

At the end of the generated source file, there is a Python dictionary describing all the objects supported in the module. For example, our module generated for Excel has entries:

```
CLSIDToClassMap = {
  '{00024428-0000-0000-C000-000000000046}' : _QueryTable,
  '{00024423-0001-0000-C000-000000000046}' : ICustomView,
  '{00024424-0001-0000-C000-000000000046}' : IFormatConditions,
  '{00024425-0001-0000-C000-000000000046}' : IFormatCondition,
  '{00024420-0000-0000-C000-000000000046}' : CalculatedFields,
  # And many, many more removed!
}
```

This dictionary is used at runtime to convert COM objects into the actual classes defined in the module. When the PythonCOM framework receives an `IDispatch` object, it asks the object for its Class ID (CLSID), then consults the map for the class that provides the interface to the object.

Forcing Early or Late Binding

When you use the `win32com.client.Dispatch()` method, the PythonCOM framework automatically selects the best available binding method; if MakePy support for an object exists, it provides early binding; otherwise the dynamic dispatch method provides late binding. In some cases, you may wish to get explicit control over the binding method.

The `win32com.client.Dispatch()` method achieves this functionality by initially checking to see if MakePy support exists for the object. If MakePy support doesn't exist, the Python module `win32com.client.dynamic` is called to perform the late-bound functionality. To force late binding for your objects, use the `win32com.client.dynamic` module directly, bypassing any MakePy-generated objects.

The `win32com.client.dynamic` module contains only one function designed to be used by Python programmers, `win32com.client.dynamic.Dispatch()`. This function is used in the same way as `win32com.client.Dispatch()`, except that MakePy support is never used for the returned object.

To force the use of early binding to access COM objects, you must force the MakePy process in your code. Once you have ensured the MakePy support exists, use `win32com.client.Dispatch()` as usual. It always returns the MakePy-supported wrappers for your COM object.

To force the MakePy process, the `win32com.client.gencache` module is used. This module contains the code that manages the directory of MakePy-generated source files: the generated cache, or *gencache*. There are a number of useful functions in this module, and you are encouraged to browse the source file if you need to perform advanced management of these generated files.

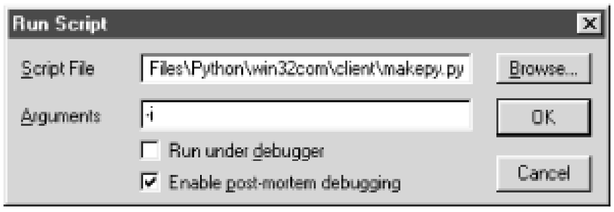
To generate a MakePy file at runtime, you need to know the unique ID of the type library (a CLSID) and its version and language identifier. This information is usually not easy to find, so the MakePy module supports a convenient method to obtain this information.

If you run the MakePy script with a `-i` parameter, instead of generating the source module, it prints the information necessary to force the MakePy process at run-time. The easiest way to do this is to perform the following steps:

1. Start PythonWin and select File → Run.
2. Click on the Browse button and locate the file *makepy.py* in the *win32com\client* directory.
3. Enter `-i` in the arguments control.

Your dialog should now look something like Figure 12-2.

Figure 12-2.Running MakePy with the -i argument


Run Script

Script File: Files\Python\win32com\client\makepy.py

Arguments: -i

☐ Run under debugger

☑ Enable post-mortem debugging

Click on the OK button and again select the entry Microsoft Excel 8.0 Object Library (1.2). You should see the following text printed in the PythonWin interactive window:

```
{00020813-0000-0000-C000-000000000046}, lcid=0, major=1, minor=2
 >>> # Use these commands in Python code to auto generate .py support
 >>> from win32com.client import gencache
 >>> gencache.EnsureModule('{00020813-0000-0000-C000-000000000046}', 0, 1, 2)
```

Let's tie all this together in a file that demonstrates what we've covered so far.

The following example first creates a late-bound object for Microsoft Excel, then forces

MakePy to be run over the type library and create an early-bound object. You do nothing with the object; simply print the object to the output window:

```
# LateAndEarly.py - Demonstrates how to force
# late or early binding of your COM objects.

import win32com.client
import win32com.client.dynamic

print "Creating late-bound Excel object"
xl = win32com.client.dynamic.Dispatch("Excel.Application")
print "The Excel object is", `xl`


print "Running makepy for Excel"
# NOTE - these 2 lines are copied verbatim from the output
# of makepy.py when run with the -i parameter.
from win32com.client import gencache
gencache.EnsureModule('{00020813-0000-0000-C000-000000000046}', 0, 1, 2)

xl = win32com.client.Dispatch("Excel.Application")
print "The Excel object is", `xl`
```

Note that you copied the output of `makepy -i` verbatim into your source code.

Before running this code, remove the existing cache of *.py* files. If you run this code with a previously generated MakePy file for the Excel object, it won't be created again. To delete the cache of *.py* files, locate the *Python\win32com\gen_py* directory and delete it. You can delete the entire directory or just the files in the directory. Be sure to remove all files, not just the *.py* files.

If you run this code, notice that a progress bar is displayed as the *.py* file is generated, and this newly generated module is used for the early-bound object. If you then run this code a second time, notice you get the same output, but no generation process; this demonstrates you can force late-bound objects to be used, even when early-bound MakePy support exists for the object.

The output from this script should be:

```
Creating late-bound Excel object
The Excel object is <COMObject Excel.Application>
Running makepy for Excel
The Excel object is <win32com.gen_py.Microsoft Excel 8.0 Object
    Library.Application>
```

Differences Between Early and Late Binding

There are a number of differences between using early and late binding within Python. All these changes are specific to Python and not to COM itself. These differences are most significant when moving from code that uses late binding to code that uses early binding.

The key difference is the handling of parameters; in fact, these differences are so significant that we discuss them separately later in the section "Passing and Obtaining Python Objects from COM."

Another fundamental difference is case sensitivity. Late binding is generally not sensitive to the case of methods and properties, while early binding is. To see an example of this, create a late-bound Excel object and adjust its `Visible` property. As discussed in the previous section, you force a late-bound object even if MakePy support exists for the object:

```
>>> import win32com.client.dynamic
>>> xl=win32com.client.dynamic.Dispatch("Excel.Application")
>>> xl.Visible=1
>>> print xl.VISIBLE
1
```

```
>>>
```

You can use both `Visible` and `VISIBLE` in this context.

Now let's try the same example using early bindings. Assume that you have generated MakePy support for Microsoft Excel and use the same code:

```
>>> import win32com.client
>>> xl=win32com.client.Dispatch("Excel.Application")
>>> xl.Visible=1
>>> print xl.VISIBLE
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "c:\Program Files\Python\win32com\gen_py\00020813-0000-0000-C000-00000000
 line 1462, in _    _getattr_    _
    raise AttributeError, attr
AttributeError: VISIBLE
```

Note that using `VISIBLE` fails. The reason is simple; when using late binding, Python asks Excel for a `Visible` property and then for a `VISIBLE` property. Excel itself is case-insensitive, so it happily indicates both properties are OK. When using early binding, Python source code is generated, and all property and method references are handled by Python itself. Because Python is case-sensitive, it allows only the case that Excel reports for the property--in this case `Visible`. When the early-binding code attempts to use `VISIBLE`, Python raises the exception without consulting Excel.

Using COM Constants

Many COM type libraries also include enumerations, which are named constants and used with the type library. For example, the type library used with Microsoft Excel includes constants named `xlAscdending`, `xlDescending`, and so forth, and are used typically as parameters to (or return values from) methods or properties.

These are made available from the Python object `win32com.client.constants`, for example, `win32com.client.constants.xlAscending`.

It's important to note that the constants for a package don't exist until the MakePy-generated module has been imported; that is, until you create or use an object from the module. You can see this in action if you start Python and attempt to reference a constant from the Microsoft Excel type library:

```
>>> from win32com.client import constants, Dispatch
>>> constants.xlAscending
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File " win32com\client\_    _init_    _.py", line 53, in _    _getattr_    _
    raise AttributeError, a
AttributeError: xlAscending
>>>
```

You can see that attempting to use these constants results in an attribute error. However, if you first create an `Excel.Application` object, the constants become available:

```
>>> xl=Dispatch("Excel.Application")
>>> constants.xlAscending
1
```

Of course, because these constants are read from a type library, they aren't available when you use late-bound (or dynamic dispatch) objects. In this case, you must use integer literals, rather than named constants in your source code.

Passing and Obtaining Python Objects from COM

COM supports a variety of argument types, using the VARIANT data structure. The types that can be passed to COM functions include integers and floats of various sizes, strings, date/time values, COM objects, or arrays of any of these types.

In many cases, PythonCOM can translate between Python objects and VARIANT structures seamlessly. When you call a COM object and pass a Python object, PythonCOM automatically creates a VARIANT of the right type and passes the VARIANT to COM. In the absence of any hints, PythonCOM performs the translations as listed in Table 12-1 and Table 12-2. In Table 12-1, for example, you can see that a Python integer is automatically converted to a VARIANT type VT_I4.

Table 12-1:Default Python Object to VARIANT Translation

Python Object Type	VARIANT Type
Integer	VT_I4
Long Integer	VT_I4 if the value is less than 232, or VT_I8 if greater
String/Unicode	VT_BSTR
Float	VT_R8
PyTrue/PyFalse	VT_BOOL
None	VT_NULL
win32com.client.Dispatch instance	VT_DISPATCH
PyIDispatch	VT_DISPATCH
All other PyI* PythonCOM objects	VT_UNKNOWN
Pywintypes PyTIME object	VT_DATE
Any other Python sequence	An array of VARIANTs; each element of the sequence is translated using this table

&nbsp:

Table 12-2:Default Python Object to VARIANT Translation

VARIANT Type	Python Object
VT_BOOL VT_I2 VT_I4 VT_ERROR	Integer
VT_R4 VT_R8	Float
VT_DISPATCH	PyIDispatch
VT_UNKNOWN	PyIUnknown
VT_BSTR	PyUnicode
VT_NULL VT_EMPTY	None
VT_DATE	Pywintypes PyTIME object

In some cases, these translations aren't suitable; for example, a COM object may be picky about the VARIANT types passed and accept only a VT_I2 integer, not a VT_I4 integer. This should be considered a bug in the COM object, but it does happen. In this case, you must use early-bound

COM by using MakePy. The code generated by MakePy includes information about the types of parameters expected by the COM object, and the PythonCOM framework correctly coerces the Python object into the required `VARIANT` type. If you can't use MakePy for your COM object, you must get your hands dirty and use the `PyIDispatch.InvokeTypes()` method manually; this is how MakePy gets the behavior it does. The use of `InvokeTypes()` is beyond the scope of this book.

Just as with the C and Visual Basic languages, it's possible in COM to pass objects by value or by reference. Passing by value means the value is passed, and changes to the value aren't reflected in the calling object. Passing by reference means a pointer to the value is passed, so changes to the value are reflected in the calling object.

Python doesn't support this concept; it's not possible to pass simple parameters by reference in Python. The common pattern is for Python to return the values from the function.

Fortunately, parameters passed by reference tend to be avoided. The Microsoft Office suite doesn't use them, nor do any of the other type libraries you could reasonably assume are installed on your PC. This makes demonstrating the problem using real code somewhat difficult, but as an example, let's assume you need to call a function that in C looks like:

```
BOOL GetSize( int *left, int *right, int *top, int *bottom);
```

Your C code to call this function looks like this:

```
int left, right, top, bottom;
BOOL ok;
ok = GetSize( &left, &right, &top, &bottom);
```

Or in Visual Basic, the code looks like:

```
Declare GetSize( ByRef left as integer, ByRef right as integer, _
                 ByRef top as integer, ByRef bottom as integer) as Integer
...
ok = GetSize(left, right, top, bottom);
```

In Python, the code looks something like:

```
left, right, top, bottom = GetSize() # Exception indicates error.
```

Note that the output parameters have been converted to the function result; the same style is used for PythonCOM. It's critical to note, however, that ByRef-style parameters may be detected only when using early-bound dispatch. If you haven't used MakePy for the type library, PythonCOM may not detect that the parameters are indeed marked as by reference and therefore may not work as expected.

The moral of the story is this: for anything other than simple arguments, it's highly recommended you use MakePy to force early-bound dispatch. If you have any problems with COM parameters and aren't using MakePy, try switching to it, and your problem is likely to go away.

Using Other COM Interfaces

So far, we have only discussed using `IDispatch` (or automation) COM objects from Python and only via Python helper classes. Although this is the most common way to use COM objects, more advanced applications often need native COM interfaces.

To illustrate this contingency, we will demonstrate the use of native interfaces with a little utility to dump statistics from a Microsoft Office application file (e.g., a Word document or Excel spreadsheet).

COM provides a technology known as *structured storage*. This is a set of functions and interfaces that allows you to store rich, hierarchical streams of data inside a single file, often referred to as a "filesystem within a file."

Part of this implementation provides for standard properties about the file--the author of the file, for example. Windows Explorer is aware of these interfaces and can display the properties without any knowledge of the application that created the file. Microsoft Office stores its documents in structured storage files, and therefore the Windows Explorer can display rich information about Office documents.

To access these properties, call a COM function to open the structured storage file. This operation results in a `PyIStorage` object, a Python object that wraps the COM `IStorage` interface. If the document has standard properties, you get these through the COM `IPropertySetStorage` interface, which means you should perform a `QueryInterface()` on the `PyIStorage` object to get the needed interface. Then open the property set you want and query for the properties.

We won't discuss the `IPropertySetStorage` and `IPropertyStorage` interfaces in any detail; the focus for this example is how to work with COM interfaces from Python, not what these particular interfaces do:

```
# DumpStorage.py - Dumps some user defined properties
# of a COM Structured Storage file.

import pythoncom
from win32com import storagecon # constants related to storage functions.

# These come from ObjIdl.h
FMTID_UserDefinedProperties = "{F29F85E0-4FF9-1068-AB91-08002B27B3D9}"

PIDSI_TITLE               = 0x00000002
PIDSI_SUBJECT             = 0x00000003
PIDSI_AUTHOR              = 0x00000004
PIDSI_CREATE_DTM          = 0x0000000c

def PrintStats(filename):
    if not pythoncom.StgIsStorageFile(filename):
        print "The file is not a storage file!"
        return
    # Open the file.
    flags = storagecon.STGM_READ | storagecon.STGM_SHARE_EXCLUSIVE
    stg = pythoncom.StgOpenStorage(filename, None, flags )

    # Now see if the storage object supports Property Information.
    try:
        pss = stg.QueryInterface(pythoncom.IID_IPropertySetStorage)
    except pythoncom.com_error:
        print "No summary information is available"
        return
    # Open the user defined properties.
    ps = pss.Open(FMTID_UserDefinedProperties)
    props = PIDSI_TITLE, PIDSI_SUBJECT, PIDSI_AUTHOR, PIDSI_CREATE_DTM
    data = ps.ReadMultiple( props )
    # Unpack the result into the items.
    title, subject, author, created = data
    print "Title:", title
    print "Subject:", subject
    print "Author:", author
    print "Created:", created.Format()

if __name__=='__main__':
    import sys
    if len(sys.argv)<2:
        print "Please specify a file name"
    else:
        PrintStats(sys.argv[1])
```

The first step is to check whether the file is indeed a structured storage file, then call `pythoncom.StgOpenStorage()` to obtain a Python `PyIStorage` interface object. You call the Python interface objects just like normal Python objects, as you'd expect. The `QueryInterface()` method can be used on any Python interface object, and returns a new interface object or throws an exception.

The output of running the example over the Microsoft Word document that contains this chapter is:

```
C:\Scripts>python.exe DumpStorage.py "Python and COM.doc"
Title: Python and COM
Subject:
Author: Mark Hammond
Created: 03/04/99 00:41:00

C:\Scripts>
```

A final note on native interfaces: Python can't support arbitrary COM interfaces; the `pythoncom` module (or a `pythoncom` extension) must have built-in support for the interface. Fortunately, there are tools `pythoncom` developers use that largely automate the process of supporting new interfaces.

Error Handling

COM uses three schemes to report error information to client applications:

- All COM interface methods return an integer status code (known as an `HRESULT`), with COM defining many common values for these `HRESULT`s. There is an `HRESULT` to indicate success and a number of `HRESULT`s that indicate warnings. All other `HRESULT` values indicate an error status.

- COM defines two special interfaces that report extended error information--`ISupportErrorInfo` and `IErrorInfo`. When any method fails, the client can perform a `QueryInterface()` to determine if the interface supports providing additional error information.

- `IDispatch` (automation) interfaces have a standard technique for reporting COM exceptions. When an `IDispatch` object encounters an error, it fills out an exception structure and returns it to the caller.

The PythonCOM framework combines all these error-reporting mechanisms into a single, Python-exception mechanism. This means you can effectively ignore the three techniques listed: PythonCOM unifies them, so you never need to know how the details of an error were obtained.

All COM errors are reported to Python programs as `pythoncom.com_error` exceptions. The exception value has the following parts:

- The `HRESULT` of the COM function.

- A text representation of the `HRESULT`. For example, if the `HRESULT` is `E_NOINTERFACE`, the text representation is (for English users) "No such interface supported."

- Additional exception information as described later in this chapter, or `None` if no additional information is supported.

- If the error is due to a parameter to a function, an integer indicating the parameter in error. This may be `None` or -1 if no information about the argument in error can be determined.

The error codes are worthy of discussion. The COM rules state that if additional exception information is available, the `HRESULT` should be `win32con.DISP_E_EXCEPTION`. However, not all COM objects meet this requirement, so the behavior shouldn't be relied on. If additional exception information is available, it will be a tuple of:

- An additional error code for the error (the `wCode`)
- The source of the error as a string, typically the application name
- A text description of the error
- The name of a Windows help file with additional information about the error
- A help context to identify the topic in the Windows help file
- Yet another error code for the error (the `sCode`)

As mentioned, if this exception information is available, the `HRESULT` should be `win32con.DISP_E_EXCEPTION`. In this case, either the `wCode` or the `sCode` contains the actual error. One of these must be zero, but it depends on the object implementing the error exactly which is used.

Let's see some code that catches a COM exception. For this example, we'll write a function to open an Excel spreadsheet. If this function fails, we print all the details known about the error. First, let's define the function:

```
>>> from win32com.client import Dispatch
>>> import pythoncom
>>> def OpenExcelSheet(filename):
...     try:
...         xl = Dispatch("Excel.Application")
...         xl.Workbooks.Open(filename)
...     except pythoncom.com_error, (hr, msg, exc, arg):
...         print "The Excel call failed with code %d: %s" % (hr, msg)
...         if exc is None:
...             print "There is no extended error information"
...         else:
...             wcode, source, text, helpFile, helpId, scode = exc
...             print "The source of the error is", source
...             print "The error message is", text
...             print "More info can be found in %s (id=%d)" % (helpFile, helpIc
...
>>>
```

As you can see, there's a Python `except` block to catch all COM errors. The first thing to do is print the generic information about the message, then check for extended information. If the extended information exists, decode and print that too.

To try this function, you could use the following code (assuming, of course, you don't have an Excel spreadsheet named *foo.xls* lying around):

```
>>> OpenExcelSheet("foo.xls")
The Excel call failed with code -2147352567: Exception occurred.
The source of the error is Microsoft Excel
The error message is 'foo.xls' could not be found. Check the spelling of the
file name, and verify that the file location is correct.

If you are trying to open the file from your list of most recently used
files on the File menu, make sure that the file has not been renamed,
moved, or deleted.
More info can be found in XLMAIN8.HLP (id=0)
```

The first line of output displays the raw `HRESULT` for the function. In this case, it's `winerror.DISP_E_EXCEPTION`, and we do have extended error information, so Excel is following the COM rules. The second line displays the application that generated the error. The full error text is large: in this case five lines long! The error messages have been designed to be placed directly in a message box for the user. The last line of the text tells us the name of the Windows help file that contains further information about the error.

Implementing COM Objects in Python

Implementing a COM object using Python means you expose a Python object to be used by any COM-aware environment, such as Visual Basic or Delphi.

In Chapter 5, we presented a simple example of a Python class exposed as a COM object. In this section, we provide a more detailed picture of exposing Python objects via COM.

Implementing a COM Server

In Chapter 5 we presented a sample COM server. This example recaps that code:

```
# SimpleCOMServer.py - A sample COM server - almost as small as they come!
#
# We simply expose a single method in a Python COM object.
class PythonUtilities:
    _public_methods_ = [ 'SplitString' ]
    _reg_progid_ = "PythonDemos.Utilities"
    # NEVER copy the following ID
    # Use "print pythoncom.CreateGuid()" to make a new one.
    _reg_clsid_ = "{41E24E95-D45A-11D2-852C-204C4F4F5020}"

    def SplitString(self, val, item=None):
        import string
        if item != None: item = str(item)
        return string.split(str(val), item)

# Add code so that when this script is run by
# Python.exe, it self-registers.
if _    _name_    _=='_    _main_    _':
    print "Registering COM server..."
    import win32com.server.register
    win32com.server.register.UseCommandLine(PythonUtilities)
```

The main points from the example are:

- Most COM servers are implemented as Python classes. These classes have special attribute annotations that indicate how the object is published via COM; our sample uses the minimum possible to register and expose a COM server.
- The `win32com` package automatically registers and unregisters the COM server.

The list of annotation attributes can be broken into two sets: those that expose the object via COM and those that allow the object to be registered via COM. Table 12-3 lists the annotations used at runtime; registration attributes are covered in the next section.

Table 12-3:Runtime-Related Annotations on COM Objects

Attribute	Description
`_public_methods_`	A list of strings that indicate the names of the public methods for the object. COM objects can use only methods listed here, the rest are considered private. This is the only required attribute; all others are optional.
`_public_attrs_`	A list of strings that indicate the public attributes (or properties) for the object. Any attributes not listed here are considered private. Any attributes listed here but not in `_readonly_attrs_` can be read or written. It is possible to list the name of Python methods here, in which case the property is implemented by calling the Python method rather than fetching the attribute directly.
`_readonly_attrs_`	A list of attributes that should be considered read-only. All names in this list should also be in `_public_attrs_` , otherwise they shouldn't be exposed.
`_value_`	A method (not the name of a method) that provides the default value for the object. We present an example of this in the sample code. Because this is a method, the typical way to implement this is to add a method to your class named `_value_`.
`_NewEnum`	A method (not the name of a method) that's used to when the client using this object requests an enumerator. This function must provide a function that conforms to the enumerator specification.
`_Evaluate`	Used when the client using this object requests to evaluate it. This appears to be a rarely used COM concept.

Registering Your COM Server

Although our sample object implements registration of the object, we haven't discussed it in detail.

Registering an object is the process of allowing the object to be independently created; once an object is registered, a language can use its standard techniques for creating COM objects to access it, e.g., `CreateObject()` in Visual Basic or `win32com.client.Dispatch()` in Python.

There are many cases where you wish to implement a COM object but don't need it registered. For example, let's assume you are designing an object model for an editor that has a root `Application` object, and inside this `Application` object there are a number of `Document` objects. In this case, you would typically want to register the `Application` object (clients need to be able to create this object directly) but not register the `Document` object (making requests on the `Application` object creates these). In this case, you don't need to specify any registration information for the `Document` object.

To prepare an object for registration, you need to provide additional attribute annotations on the object. The registration process uses these annotations to provide the correct information in the Windows registry for the object. The full list of registration related attributes can be found in Table 12-4.

Table 12-4:Registration-Related Attributes on COM Objects

Attribute	Description
`_reg_progid_`	The ProgID for the object. This is the name of the COM object clients use to create the object.
`_reg_desc_`	Optional description of the COM object. If not specified, `_reg_progid_` is used as the description.
`_reg_classspec_`	An optional string identifying the Python module and the object in the module. The PythonCOM framework uses this string to instantiate the COM object. If neither this nor `_reg_policyspec_` are provided, the COM framework determines the value from the command line.
`_reg_policyspec_`	An optional string identifying the PythonCOM policy to be used for this object. If not provided, the default policy is used. See the section "Policies" later in this chapter.
`_reg_verprogid_`	The version-dependent ProgID. This is typically the ProgID with a version number appended. For example, the second version of a particular server may have a ProgID of `Python.Object` and a VerProgId of `Python.Object.2`.
`_reg_icon_`	The default icon for the COM object.
`_reg_threading_`	The default threading model for the COM object. This must be one of the COM-defined values acceptable for the ThreadingModel key of the COM server, e.g., `Apartment`, `Free`, or `Both`. If not specified, `Both` is used. See *Appendix D, Threads,* for a discussion on COM threading models.
`_reg_catids_`	A list of category IDs for the server. See the COM documentation on categories for more information.
`_reg_options_`	A dictionary of additional keys to be written to the registry for the COM object. The PythonCOM framework doesn't define the values for this; it's up to the author to specify meaningful values.
`_reg_clsctx_`	The contexts defining how this object is to be registered. This attribute defines if the COM object is registered as an `InProc` object (i.e., implemented by a DLL) or a `LocalServer` object (i.e, implemented in an EXE). If not specified, the default of `CLSCTX_LOCAL_SERVER | CLSCTX_INPROC_SERVER` is used (i.e., the object is registered as both `InProc` and `LocalServer`).
`_reg_disable_pycomcat_`	A boolean flag indicating if the COM object should be associated with the list of PythonCOM servers installed on the machine. If not specified, the object is associated with the PythonCOM servers.
`_reg_dispatcher_spec_` `_reg_debug_dispatcher_spec_`	The dispatcher for the COM object. Dispatchers are largely a debugging aid, allowing you to snoop on your COM object as calls are made on it. Dispatchers are closely related to policies but aren't covered in this book.

The module `win32com.server.register` contains many utilities for registering and unregistering COM servers. The most useful of these functions is `UseCommandLine()`, which allows you to register any number of Python classes. Using this function is a no-brainer; pass to this function the class objects you wish to expose.

In the COM example, we include the following code:

```
if __name__=='__main__':
    print "Registering COM server..."
    import win32com.server.register
    win32com.server.register.UseCommandLine(PythonUtilities)
```

The `PythonUtilities` object is the class to register. Adding this functionality allows the COM object to be registered or unregistered from the command line as detailed in Table 12-5.

Table 12-5:Command-Line Options Recognized by UseCommandLine

Command-Line Option	Description
	The default is to register the COM objects.
`--unregister`	Unregisters the objects. This removes all references to the objects from the Windows registry.
`--debug`	Registers the COM servers in debug mode. We discuss debugging COM servers later in this chapter.
`--quiet`	Register (or unregister) the object quietly (i.e., don't report success).

Each option uses a double hyphen. For example, if your COM objects are implemented in *YourServer.py*, use the following commands.

To register objects:

```
C:\Scripts> Python.exe YourServer.py
```

To unregister the objects:

```
C:\Scripts> Python.exe YourServer.py --unregister
```

To register the objects for debugging:

```
C:\Scripts> Python.exe YourServer.py --debug
```

With a standard Python setup, double-clicking on a Python COM server script in Explorer has the same effect as the first example and registers the server.

Error Handling for COM Servers

When you implement a COM object, it's often necessary to return error information to the caller. Although Python has a powerful exception mechanism, the caller of your objects is likely to be Visual Basic or Delphi, so standard Python exceptions don't really work.

To support this, the `win32com.server.exception` module exposes the `COMException` Python object in order to raise an exception to COM. This object allows you to specify many details about the error, including the error message, the name of the application generating the error, the name of a help file in which the user can find additional information, etc. See the `win32com.server.exception` module for more details.

The PythonCOM framework makes the assumption that all Python exceptions other than `COMException` indicate a bug in your code. Thus, your object shouldn't allow normal Python exceptions to be raised when calling your methods, but should take steps to handle these Python exceptions and translate them to an appropriate `COMException`.

As an example, let's assume you want to publish a method called `sqrt()` that returns the square root of its argument. If you use the following code:

```
def sqrt(self, val):
    return math.sqrt(val)
```

you have a potential problem; in fact, a few of them. If you pass anything other than a positive number to your function, the code fails, and a Python exception is raised. This is considered a bug in your COM object. To improve this function, use the following code:

```
def sqrt(self, val):
    try:
        return math.sqrt(val)
    except (TypeError, ValueError):
        raise COMException("The argument must be a positive number", \
                           winerror.DISP_E_TYPEMISMATCH)
```

This version of the code does the right thing: it traps the exceptions that may be raised by the `math.sqrt()` function and raises a `COMException` object with a useful message and value.

Policies

PythonCOM policies are an advanced topic and typically don't need to be understood to successfully use Python and COM. However, if you need to perform advanced techniques using Python and COM, this information is valuable. You may wish to skip this section and come back to it when the need arises.

A PythonCOM policy determines how Python objects are exposed to COM; the policy dictates which attributes are exposed to COM and the IDs these attributes get. The policy actually sits between COM and your object and is responsible for responding to the `IDispatch` interface's `GetIDsOfNames()` and `Invoke()` functions. The policy dictates how these `IDispatch` calls are translated into the references to your Python object.

The default PythonCOM policy is suitable in most cases, and all the examples to date have used the default policy. The policy implemented is:

- All methods named in the `_public_methods_` attribute are exposed to COM. Any method not listed in `_public_methods_` is considered private.
- All properties named in the `_public_attrs_` attribute are exposed to COM. If the property name also appears in the attribute `_readonly_attrs_`, it can be read, but not written; otherwise, users of this object can change the property.
- Other special attributes can obtain advanced behavior. You can review the full list of attributes in Table 12-4.

What this means is that the PythonCOM framework itself doesn't determine how an object is exposed via COM; it's determined by the policy.

The PythonCOM package provides two useful policies: the default policy (known as the DesignatedWrapPolicy, because the attributes exposed via COM must be explicitly designated) and the DynamicPolicy that implements a far more liberal approach to publishing objects. These policies are implemented in the `win32com.server.policy` module.

The DynamicPolicy requires your Python class to implement a single function named `_dynamic_` , and this function must implement the logic to determine if the COM call is requesting a property reference or a method call.

To demonstrate the DynamicPolicy, we present a more advanced COM server in the following example. The aim of this COM server is to expose the entire Python string module. Anyone using Visual Basic or Delphi can then use all the string-processing functions available to Python.

Before we look at the code, there are a couple of points:

- The Unicode strings bite us again! As COM passes all strings as Unicode, you need to convert them to Python strings before the string module can work with them.
- Most of the registration data is the same as discussed previously, except there is a new attribute, `_reg_policy_spec_`. This attribute identifies that you need to use the DynamicPolicy rather than the default DesignatedWrapPolicy.
- The handling of `IDispatch.GetIDsOfNames()` has been done for you (the `_dynamic_` methods deal with attribute names), and the policy has dealt with the IDs for the attributes.
- There is some extra internal knowledge of COM needed to implement the `_dynamic_` method. Specifically, you need to differentiate between a property reference and a method call. Also remember that VB is not case-sensitive, while Python is.

```
# DynamicPolicy.py -- A demonstration of dynamic policies in PythonCOM
import string
import pythoncom
import pywintypes
import winerror
import types
from win32com.server.exception import COMException

def FixArgs(args):
    # Fix the arguments, so Unicode objects are
    # converted to strings.  Does this recursively,
    # to ensure sub-lists (ie, arrays) are also converted
    newArgs = []
    for arg in args:
        if type(arg)==types.TupleType:
            arg = FixArgs(arg)
        elif type(arg)==pywintypes.UnicodeType:
            arg = str(arg)
        newArgs.append(arg)
    return tuple(newArgs)

class PythonStringModule:
    _reg_progid_ = "PythonDemos.StringModule"
    _reg_clsid_ = "{CB2E1BC5-D6A5-11D2-852D-204C4F4F5020}"
    _reg_policy_spec_ = "DynamicPolicy"

    # The dynamic policy insists that we provide a method
    # named _dynamic_, and that we handle the IDispatch::Invoke logic.
    def _dynamic_(self, name, lcid, wFlags, args):
        # Get the requested attribute from the string module.
        try:
            item = getattr(string, string.lower(name))
        except AttributeError:
            raise COMException("No attribute of that name", \
                               winerror.DISP_E_MEMBERNOTFOUND)
        # Massage the arguments...
        args = FixArgs(args)
        # VB will often make calls with wFlags set to
        # DISPATCH_METHOD | DISPATCH_PROPERTYGET, as the VB
        # syntax makes the distinction impossible to make.
        # Therefore, we also check the object being referenced is
        # in fact a Python function
        if (wFlags & pythoncom.DISPATCH_METHOD) and \
           type(item) in [types.BuiltinFunctionType, types.FunctionType]:
            return apply(item, args)
        elif wFlags & pythoncom.DISPATCH_PROPERTYGET:
            return item
        else:
            raise COMException("You can not set this attribute",
                               winerror.DISP_E_BADVARTYPE)

# Add code so that when this script is run by
# Python.exe, it self-registers.
```

```
if __name__=='__main__':
    import win32com.server.register
    win32com.server.register.UseCommandLine(PythonStringModule)
```

To test the COM object, use the following VBA code:

```
Sub Test()
    ' Create the Python COM objects.
    Set stringmod = CreateObject("PythonDemos.StringModule")
    ' Call string.split
    response = stringmod.Split("Hello from VB")
    For Each Item In response
        MsgBox (Item)
    Next
    ' Call string.join
    MsgBox "The items joined are " & stringmod.join(response)
    ' Get string.uppercase
    MsgBox "The upper case character are" & stringmod.uppercase

    ' Attempt to set a property - this should fail.
    stringmod.uppercase = "Hi"

End Sub
```

When you run this code, you should see a series of message boxes, followed by an error dialog. As mentioned in the code, the attempt to set `string.uppercase` should fail, and indeed it does.

As you can see, the DynamicPolicy has given you the tools to wrap any arbitrary Python object, rather than requiring you to explicitly declare the public interface. Depending on your requirements, this may or may not serve your purpose better than the default policy, but if neither of these policies meet your requirements, just write your own! The `Python.Dictionary` sample COM object (implemented in the module `win32com.servers.dictionary`) implements its own specialized policy, so it's a good starting point if you need to go this route.

Wrapping and Unwrapping

Whenever you expose a Python object via COM, you actually expose an `IDispatch` object. As described previously, the `IDispatch` interface is used to expose automation objects. Thus, whenever a Visual Basic program is using a Python COM object, VB itself is dealing with a COM `IDispatch` object. The Python COM framework provides the `IDispatch` object that wraps your Python COM class instance. Whenever the COM framework creates a new Python COM object, the general process is:

- An instance of the selected policy for the object is created.
- The policy creates an instance of your Python class.
- An `IDispatch` object is created that wraps the Python policy (which in turn wraps your instance).
- The `IDispatch` object is returned to the client (e.g., Visual Basic).

Thus, when you need to create an `IDispatch` from a Python class instance, you should wrap the object. Unwrapping is the reverse of this process; if you have an `IDispatch` object that wraps a Python instance, you can unwrap the `IDispatch` object, returning the underlying Python instance.

In many cases you don't need to worry about this. When you expose a COM object that can be directly created via COM, the Python COM framework handles all the wrapping for you. However, there are a number of cases where the explicit wrapping of objects is necessary.

The most common scenario is when you need to expose a COM object via some sort of factory method; that is, rather than allowing the user to create your object directly using VB's `CreateObject()`, you return the object from another object. Microsoft Office provides good examples of this behavior: the object model defines `Cell` or `Paragraph` objects, but you can't create them directly. You must create the `Application` object and use it to create or reference `Cells` and `Paragraphs`.

We will use a contrived example to demonstrate the wrapping and unwrapping of objects. But before that, we take a small digression into techniques that debug the COM objects. These debugging techniques demonstrate wrapping and unwrapping and also how to use `IDispatch` objects passed as parameters.

Debugging Python COM Objects

When you use COM clients such as Excel from Python, you can employ the same debugging techniques as for any Python code; you are simply calling Python objects. However, when you implement COM objects in Python, things become more difficult. In this case, the caller of your Python code isn't Python, but another application, such as Visual Basic or Delphi. These applications obviously have no concept of a Python exception or a Python traceback, so finding bugs in your Python code can be a problem.

In a nutshell: register your COM objects using `-debug` on the command line. Then use the Trace Collector Debugging Tool item on the PythonWin Tools menu to see any print statements or Python exceptions. The rest of this section is devoted to how this works.

To assist with the debugging problem, the Python COM framework has the concept of a *dispatcher*. A dispatcher is similar to a policy object, but dispatches calls to the policy. The `win32com` package provides a number of useful dispatchers.

When you register the COM Server with `--debug` (note the double hyphen), the registration mechanism also registers a dispatcher for your object. The default dispatcher is known as `DispatcherWin32trace,` although you can specify a different dispatcher using the `_reg_debug_dispatcher_spec_` attribute on your object, as described in the earlier section "Registering Your COM Server."

The default `DispatcherWin32trace` uses the `win32trace` module to display its output. To see the output of a COM server when debugging, use the Trace Collector Debugging Tool item on the PythonWin Tools menu.

The Final Sample

A final sample COM server demonstrates wrapping and unwrapping objects, how to use `IDispatch` objects when passed as parameters to COM functions, and also how to debug your COM servers. This example is contrived and does nothing useful other than demonstrate these concepts.

The sample exposes two COM objects, a `Parent` object and a `Child` object. The `Parent` object is registered with COM so VB code can use `CreateObject` to create it. The `Child` object isn't registered and can be created only by calling the `CreateChild()` method on the `Parent`. The `Child` object has no methods, just a `Name` property.

The Parent object also has a method called `KissChild()`, that should be called with a `Child` object previously created by the parent. The `KissChild()` method demonstrates how to use the `IDispatch` passed to the method and also how to unwrap the `IDispatch` to obtain the underlying Python object.

Finally, the code has a number of `print` statements and a lack of error handling. We use the debugging techniques to see these `print` statements and also a Python exception raised:

```
# ContrivedServer.py
#
# A contrived sample Python server that demonstrates
# wrapping, unwrapping, and passing IDispatch objects.

# Import the utilities for wrapping and unwrapping.
from win32com.server.util import wrap, unwrap

import win32com.client

# Although we are able to register our Parent object for debugging,
# our Child object is not registered, so this won't work. To get
# the debugging behavior for our wrapped objects, we must do it ourself.
debugging = 1
if debugging:
    from win32com.server.dispatcher import DefaultDebugDispatcher
    useDispatcher = DefaultDebugDispatcher
else:
    useDispatcher = None

# Our Parent object.
# This is registered, and therefore creatable
# using CreateObject etc from VB.
class Parent:
    _public_methods_ = ['CreateChild', 'KissChild']
    _reg_clsid_ = "{E8F7F001-DB69-11D2-8531-204C4F4F5020}"
    _reg_progid_ = "PythonDemos.Parent"

    def CreateChild(self):
        # We create a new Child object, and wrap
        # it using the default policy
        # If we are debugging, we also specify the default dispatcher
        child = Child()
        print "Our Python child is", child
        wrapped = wrap( child, useDispatcher=useDispatcher )
        print "Returing wrapped", wrapped
        return wrapped

    def KissChild(self, child):
        print "KissChild called with child", child
        # Our child is a PyIDispatch object, so we will attempt
        # to use it as such.  To make it into something useful,
        # we must convert it to a win32com.client.Dispatch object.
        dispatch = win32com.client.Dispatch(child)
        print "KissChild called with child named", dispatch.Name

        # Now, assuming it is a Python child object, let's
        # unwrap it to get the object back!
        child = unwrap(child)
        print "The Python child is", child

# Our Child object.
# This is not registered
class Child:
    _public_methods_ = []
    _public_attrs_ = ['Name']
    def _    _init_    _(self):
        self.Name = "Unnamed"

if _    _name_    _=='_    _main_    _':
    import win32com.server.register
    win32com.server.register.UseCommandLine(Parent, debug=debugging)
```

Before you register your class, we must mention some of the debugging-related code. Near the top of the source file, we declare a variable named `debugging`. If this variable is `true`, you then load the `DefaultDebugDispatcher` and assign it to a variable. In the `CreateChild()` method, you pass this dispatcher to the `wrap` function. This is due to a quirk in the debug mechanism. As mentioned previously, the registration mechanism allows you to register an object for debugging by using `--debug` on the command line. While this works fine for objects you register, recall that our `Child` object isn't registered, so it doesn't benefit from this mechanism;

this code enables it for both objects. Also note that you pass this debugging variable to the `UseCommandLine()` function. This allows you to control the debugging behavior totally from your debugging variable. If set, debugging is enabled for all objects, regardless of the command line. If not set, you don't get debugging for either object.

So you can register this COM server like the other COM servers; no need for anything special on the command line. You register the server by running the script (either from within PythonWin, or using Windows Explorer). After registration, you should see the message:

```
Registered: PythonDemos.Parent (for debugging)
```

The next step is for some VB code to use your object. For this demonstration, you can use the following Visual Basic for Applications code from either Excel or Word:

```
Sub DebuggingTest()
  Set ParentObj = CreateObject("PythonDemos.Parent")

  Set child = ParentObj.CreateChild()
  MsgBox "Child's name is " & child.Name

  ParentObj.KissChild child
  MsgBox "I kissed my child"

  ' Now lets pass a non-python object!
  ' As we are using VBA (either Word or Excel)
  ' we just pass our application object.
  ' This should fail with an InternalError.
  Set app = Application
  ParentObj.KissChild (app)
End Sub
```

This code is simple: it creates a `Parent` object, then calls the `CreateChild()` method, giving a `Child` object. You fetch the name of the child, then display it in a message box. The next step is to call the `KissChild()` method and display another message box when the kiss is complete. The final step is to call the `KissChild()` method, but pass a different object, in this case the Excel (or Word) `Application` object.

Before running this code, you should open the window that displays the debugging output and select the Trace Collector Debugging Tool item on the PythonWin Tools menu.

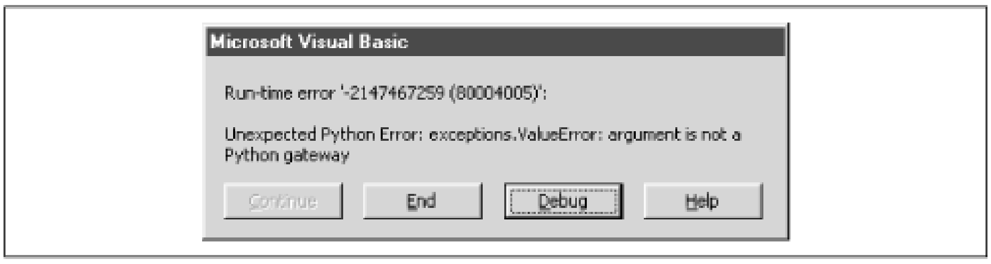
Now, let's run the Visual Basic code and stop at the first message box. The debug window should now display:

```
Object with win32trace dispatcher created (object=None)
in _GetIDsOfNames_ with '('CreateChild',)' and '1033'
in _Invoke_ with 1000 1033L 3 ()
Our Python child is <ContrivedServer.Child instance at 2a8d5e0>
Object with win32trace dispatcher created (object=<ContrivedServer.Child instanc
Returing wrapped <PyIDispatch at 0x2a80254 with obj at 0x2a801c0>
in _GetIDsOfNames_ with '('Name',)' and '1033'
in _Invoke_ with 1000 1033L 3 ()
```

The first thing to note is there are a number of unexpected lines; the Python COM framework has printed some extra debugging information for you. The first three lines show how internally `GetIDsOfNames()` and `Invoke()` have been translated by the Python COM framework. The fourth line is one of yours: it's the `print` statement in the `CreateChild()` method. Wrapping the `Child` object causes the Python COM framework to print the next output line, which is followed by the next `print` statement.

If you now dismiss the next couple of message boxes in the VB example, you should see an error message that looks like that in Figure 12-3.

Figure 12-3.Visual Basic error dialog when running the sample



Microsoft Visual Basic

Run-time error '-2147467259 (80004005)':

Unexpected Python Error: exceptions.ValueError: argument is not a Python gateway

As you can see, you have the "Unexpected Python Error," discussed previously, which means you have an unhandled Python exception in the COM server. If you look in the debugging window, the last few lines are:

```
in _Invoke_ with 1001 1033L 1 (<PyIDispatch at 0x2a7bbb4 with obj at 0xcdd4f8>,)
KissChild called with child <PyIDispatch at 0x2a7bbb4 with obj at 0xcdd4f8>
KissChild called with child named Microsoft Word
Traceback (innermost last):
  File "L:\src\pythonex\com\win32com\server\dispatcher.py", line 40, in _Invoke_
    return self.policy._Invoke_(dispid, lcid, wFlags, args)
  File "L:\src\pythonex\com\win32com\server\policy.py", line 265, in _Invoke_
    return self._invoke_(dispid, lcid, wFlags, args)
  File "L:\src\pythonex\com\win32com\server\policy.py", line 486, in _invoke_
    return S_OK, -1, self._invokeex_(dispid, lcid, wFlags, args, None, None)
  File "L:\src\pythonex\com\win32com\server\policy.py", line 498, in _invokeex_
    return apply(func, args)
  File "L:\docs\Book\Python and COM\ContrivedServer.py", line 49, in KissChild
    child = unwrap(child)
  File "L:\src\pythonex\com\win32com\server\util.py", line 36, in unwrap
    ob = pythoncom.UnwrapObject(ob)
ValueError: argument is not a Python gateway
```

Here are the full details of the Python exception. This makes tracking the error much simpler: it should be obvious that the error occurs when you attempt to unwrap the Microsoft Excel application object. The unwrap fails because there is no Python object behind this interface.

One final note relates to how an `IDispatch` object is used as a parameter to a COM function. In the debugging window, locate the following messages:

```
KissChild called with child <PyIDispatch at 0x2a86624 with obj at 0x2a865b0>
KissChild called with child named in _GetIDsOfNames_ with '('Name',)' and '0'

in _Invoke_ with 1000 0L 2 ()
Unnamed
The Python child is <ContrivedServer.Child instance at 2a7aa90>
```

The raw parameter to `KissChild()` is in fact a `PyIDispatch` object. You may recall from the start of this chapter that a `PyIDispatch` object only has methods `GetIDsOfNames()` and `Invoke()`. To turn the object into something useful, you must use a `win32com.client.Dispatch()` object. Once you have the `win32com.client.Dispatch` object, use it like any other COM object; in the example, we called the `Name` property.

Python and DCOM

Microsoft has recently enhanced COM to support distributed objects. These enhancements are known as distributed COM, or DCOM. The term *distributed objects* refers to objects that operate on different machines: a COM client and COM server may be on different parts of the network.

There are a number of reasons why this may be appealing. It allows you to host your objects close to your data; for example, on a server with high-speed access to a database server. Microsoft also has a product available called the Microsoft Transaction Server (MTS) that provides additional facilities for large-scale distributed applications. Alternatively, DCOM may allow you to use specific hardware installed on a remote machine, by running on that machine and controlling it from your own workstation.

One of the key strengths of the DCOM architecture is that in many cases, the objects don't need to have special code to support distributed objects. DCOM manages all this behind the scenes, and neither the local or remote object need be aware they are not running locally.

DCOM comes with a tool that allows you to configure the DCOM characteristics of the local machine and of each specific object registered on your machine. For each individual object, you can specify the machine where that object should be executed. In addition, the code that creates a new COM object (i.e., the COM client) can specify its own settings by making slight changes to the creation process.

To demonstrate DCOM, let's use the standard `Python.Interpreter` COM object and configure it to be created on a remote machine. Here's the process:

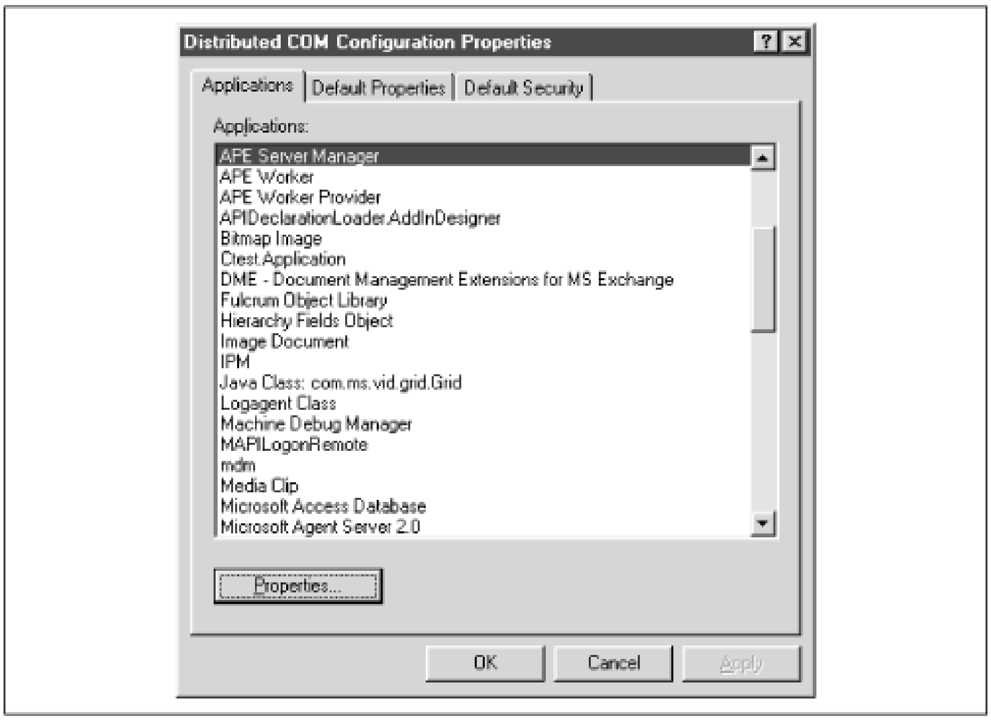
1. Configure DCOM on the remote machine.
2. Configure DCOM and our object on the local machine.
3. Test the remote object using normal COM.
4. Test the remote object using DCOM-specific calls.

Configure DCOM on the Remote Machine

The first step is to configure DCOM on the remote machine, where the object will actually run.

To configure DCOM, start the DCOM configuration tool by selecting the Run option from the Windows start menu, and enter `dcomcnfg`. The mail display is shown in Figure 12-4.

Figure 12-4.The DCOM configuration tool


Distributed COM Configuration Properties

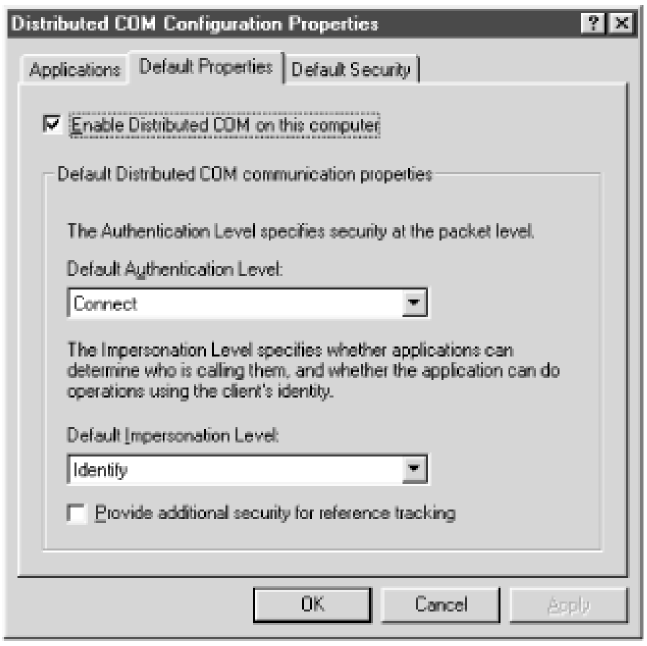
Applications | Default Properties | Default Security

Applications:

APE Server Manager
APE Worker
APE Worker Provider
APIDeclarationLoader.AddInDesigner
Bitmap Image
Ctest.Application
DME - Document Management Extensions for MS Exchange
Fulcrum Object Library
Hierarchy Fields Object
Image Document
IPM
Java Class: com.ms.vid.grid.Grid
Logagent Class
Machine Debug Manager
MAPILogonRemote
mdm
Media Clip
Microsoft Access Database
Microsoft Agent Server 2.0

Now select the Default Properties tab and ensure that DCOM is enabled for this computer, as shown in Figure 12-5.

Figure 12-5.DCOM configuration tool with DCOM enabled



☑ Enable Distributed COM on this computer

Default Distributed COM communication properties

The Authentication Level specifies security at the packet level.

Default Authentication Level: Connect

The Impersonation Level specifies whether applications can determine who is calling them, and whether the application can do operations using the client's identity.

Default Impersonation Level: Identify

☐ Provide additional security for reference tracking

No additional configuration options are required on the remote machine, but you do need to ensure the COM object itself is installed on this computer. There's nothing special about registering your object for DCOM; perform a normal registration process for the object, as described in the previous section "Registering Your COM Server."

The `Python.Interpreter` object you use is part of the standard Python COM distribution, so it should have been registered when the Python COM extensions were installed. However, it can't hurt to reregister it. To register the `Python.Interpreter` object, perform the following steps on the remote machine:

1. Start PythonWin.
2. Select File → Run, select the Browse button, and locate the file *win32com\servers\interp.py*.
3. Select OK.

The PythonWin window should report:

```
Registering COM server...
Registered: Python.Interpreter
```

Configure DCOM and the Object on the Local Machine

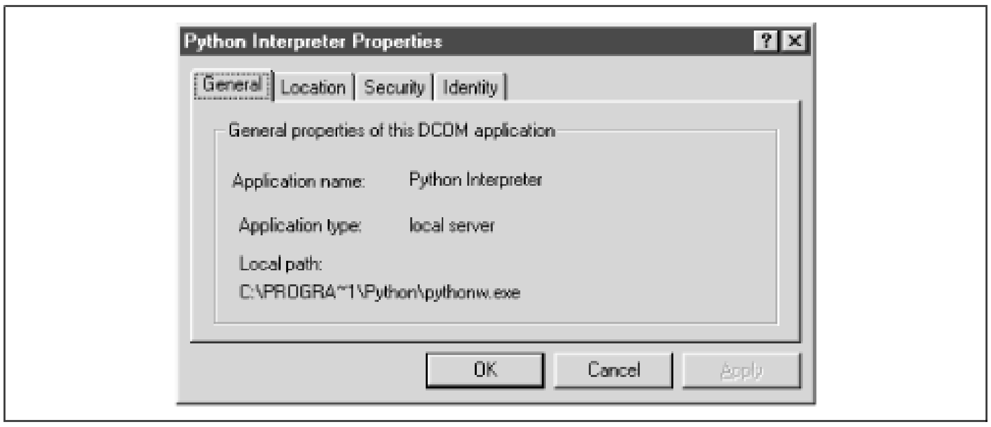
The next step is to configure your local machine, where you actually create and use the object running on the remote machine.

First, ensure the object is registered on the local machine; although you don't need to start the object on this local machine, the object must be registered locally so the COM and DCOM architecture know about the object and how to redirect it. To register the object locally, perform the same process you did for registering the object on the remote machine.

Now start the DCOM configuration tool to configure the object on the local machine. Use the same process you used to start the tool on the remote machine, but this time at the local machine.

The same initial dialog in Figure 12-4 is displayed. Locate the `Python.Interpreter` object in the list and select the Properties button. The default properties for the object should look something like Figure 12-6.

Figure 12-6.Default DCOM properties for the object


Python Interpreter Properties

General | Location | Security | Identity

General properties of this DCOM application

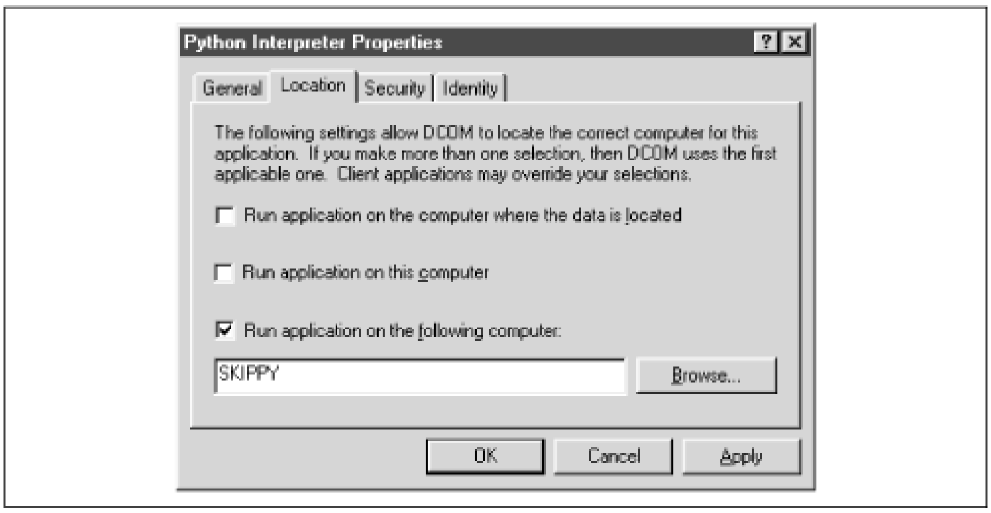
Application name: Python Interpreter

Application type: local server

Local path:
C:\PROGRA~1\Python\pythonw.exe

If you select the Location tab, you see the available options. The default setting should indicate that the application runs only on this computer. Disable the local computer option and enable the "Run application on the following computer" setting. The remote computer is named *SKIPPY*. Now enter the name of your remote machine. The dialog should now look something like Figure 12-7.

Figure 12-7.Python.Interpreter object ready for remote testing


Python Interpreter Properties

General | Location | Security | Identity

The following settings allow DCOM to locate the correct computer for this application. If you make more than one selection, then DCOM uses the first applicable one. Client applications may override your selections.

☐ Run application on the computer where the data is located

☐ Run application on this computer

☑ Run application on the following computer:

SKIPPY

Select the OK button, and you're ready to go.

Testing the object using normal COM

Before testing, there is a slight complication that needs to be addressed. If you recall the discussion at the beginning of the chapter regarding `InProc`, `LocalServer`, and `RemoteServer` objects, it should come as no surprise that remote DCOM objects must be hosted by an executable. It's not possible to use an object implemented in a DLL, since the object is running on a different computer than the user of the object, so it must be in a different process. This is not a problem, since by default all Python COM objects are registered with both executable and DLL support.

The complication is on the local machine. Although you configured DCOM not to run the object on the local computer, this applies only to `LocalServer` objects. If there is an `InProc` object registered locally, this object is used regardless of the DCOM settings. As your object is registered on the local machine as an `InProc` application, you need to take action to ensure this version of the object isn't loaded.

It's worth noting that this complication is provided courtesy of COM and DCOM. There is nothing Python-specific about this problem; it exists for all COM objects regardless of their implementation language. Python is slightly unique in that the default registration for its objects are for both `LocalServer` and `InProc`; most languages force you to choose one or another quite early in the development process. There are two ways to solve this dilemma:

- Modify the COM object so it supports only `LocalServer` operations by setting the `_reg_clsctx_` attribute in the class. See the earlier section "Registering Your COM Server" for more details.
- Make a slight change to the object creation code to explicitly exclude the `InProc` version of the object from being used.

Because we are using the existing `Python.Interpreter` example, we won't modify the it, but will go for the second option. You do this by specifying the `clsctx` parameter to the `win32com.client.Dispatch()` function. If you decide to change your COM object to support only `LocalServer` operations, this step isn't necessary, and the object creation code is identical to the normal object creation process.

To execute the object remotely, start Python or PythonWin on the local computer. First, let's prove the name of the local machine:

```
>>> import win32api
>>> win32api.GetComputerName()
'BOBCAT'
>>>
```

Now, let's create a `Python.Interpreter` object. As discussed, you pass a custom `clsctx` parameter to the `Dispatch()` function. Because the `clsctx` parameter is not the second parameter, specify it by name:

```
>>> import pythoncom, win32com.client
>>> clsctx=pythoncom.CLSCTX_LOCAL_SERVER
>>> i=win32com.client.Dispatch("Python.Interpreter", clsctx=clsctx)
>>>
```

Now, let's use this object. Ask it to report what machine it is on. Then ask the remote interpreter to import the `win32api` module and print the value of `win32api.GetComputerName()`. Because the object is running remotely, expect to see the name of the remote computer:

```
>>> i.Exec("import win32api")
>>> i.Eval("win32api.GetComputerName()")
'SKIPPY'
>>>
```

If you view the Task Manager for the remote machine, notice a new process **pythonw.exe**. This is the process hosting the remote object. If you release the reference to the object, you should see the process terminate. To release the reference, execute:

```
>>> i=None
>>>
```

And within a few seconds the process on the remote server terminates.

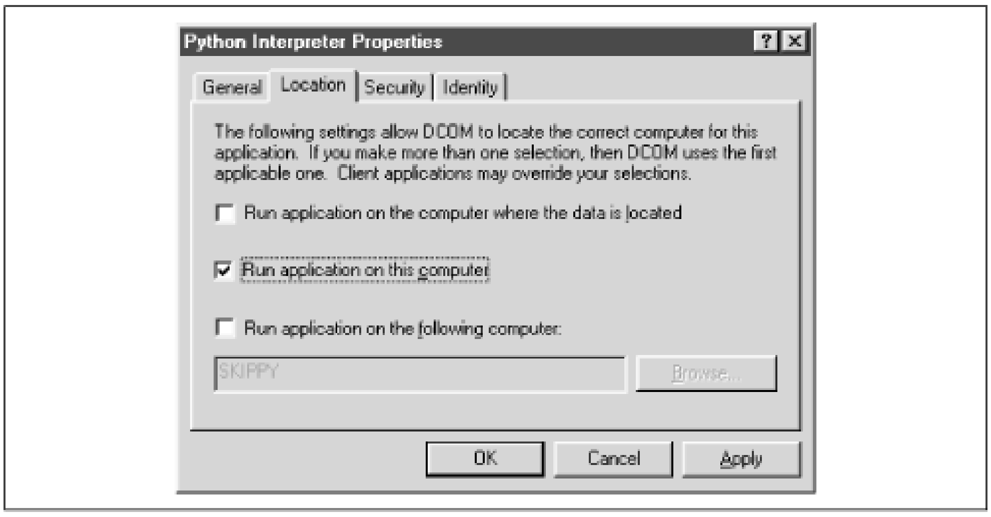
As you can see, it worked! Setting up the machines for DCOM is quite painless, and using the remote object is as simple as if it were a local object.

Testing a Remote Object Using DCOM-Specific Calls

You may have noticed that the DCOM configuration dialog states "Client applications may override your selection." So when you configure DCOM, you really just provide default values for non-DCOM-aware programs. In fact, if you want to make your client code DCOM-aware, it isn't necessary to configure DCOM on the local machine at all; your client code provides the information needed.

To demonstrate this, let's restore the DCOM configuration on the local machine to the defaults. Restart the DCOM configuration tool and again select the `Python.Interpreter` object. Select the Location tab and restore the settings to how you first found them. The dialog should look similar to Figure 12-8.

Figure 12-8.Back to the default DCOM configuration for the object



The following settings allow DCOM to locate the correct computer for this application. If you make more than one selection, then DCOM uses the first applicable one. Client applications may override your selections.

- [ ] Run application on the computer where the data is located
- [✓] Run application on this computer
- [ ] Run application on the following computer:

SKIPPY

Select the OK button to apply the changes.

First, let's check that the object is indeed restored correctly. Execute the same lines of code used previously, but because the DCOM configuration has been restored, the object should be local:

```
>>> i=win32com.client.Dispatch("Python.Interpreter", clsctx=clsctx)
>>> i.Exec("import win32api")
>>> i.Eval("win32api.GetComputerName()")
'BOBCAT'
>>>
```

As expected, the local machine name is now being used, and you should be able to locate a *pythonw.exe* process running on the local computer.

The `win32com.client.DispatchEx()` function allows you to override the DCOM defaults for your object.

The first parameter to `DispatchEx()` is the ProgID for the object you wish to create, the same ProgID you would pass to `Dispatch()`. The second parameter is the name of the machine to create the object on. If the machine name isn't specified, the call operates identically to `win32com.client.Dispatch()`.

Let's test drive this function. Although the DCOM configuration for the local machine is set up to run locally, use the following code to force a remote server:

```
>>> i=win32com.client.DispatchEx("Python.Interpreter", "skippy", clsctx=clsctx)
>>> i.Exec("import win32api")
>>> i.Eval("win32api.GetComputerName()")
'SKIPPY'
>>>
```

Note that the same `clsctx` complications exist here. If the object is registered as an `InProc` server locally, all DCOM settings (including the explicit machine name) are ignored.

Conclusion

In this chapter we took a whirlwind tour of COM. We covered the important COM concepts and how they relate to Python and the Python COM package. We discussed the differences between COM interfaces and COM automation objects that expose an object model using the `IDispatch` interface.

The `pythoncom` module and the `win32com` package were introduced, and we discussed how to use COM objects from Python and how to use native COM interfaces from Python.

Implementing COM objects using Python was discussed in detail. We covered some simple examples and covered some advanced topics, such as wrapping and unwrapping your COM objects, PythonCOM policies, and how to debug your COM servers.

Finally, we showed you how you can distribute your Python COM objects across various machines on your network using DCOM.

Back to: Sample Chapter Index

Back to: Python Programming on Win32

oreilly.com Home | **O'Reilly Bookstores** | **How to Order** | **O'Reilly Contacts**
International | **About O'Reilly** | **Affiliated Companies** | **Privacy Policy**