Windows Programming in Python

Posted on April 19, 2006
by APRajshekhar

This article shows you how to call COM components from Python, and shows you how to use COM extensions to access the mail merge functionalities of MS Word in Python.

Generally system programming is considered to be the domain of compiled languages, especially when the system under consideration is MS Windows. However there are always exceptions to the general rule. When the scripting language in consideration is of the caliber of Python, the exception is more profound.

Python gets its caliber from the fact that for each platform there is a specific set of extensions, using which the full potential of the host system can be completely tapped into. Also there are no “semantics” or syntactic changes required when using these extensions. Thus the developer can avail himself of the power of the native APIs without leaving the highly flexible environment of Python.

The extension to the Windows platform contains three distinct sets of extensions — the COM extension, access to the native Win32 API, and the PythonWin GUI extensions. In this article, I will be discussing the COM extensions. The first section will provide an overview of the COM technology. The focus of the second section will be the steps involved in using the COM extensions. In the third segment, I will be using the COM extensions to access the mail merge functionalities of MS Word. That is the outline for this discussion.

{mospagebreak title=What is COM?}

Component Object Model is a software architecture that allows software to be built from binary components. This implies that if COM objects are used in building software, they

can communicate with each other without knowing each other's details. Even if the components are implemented in different languages, they can communicate with each other.

At the lowest level, COM is all about interfaces. It leaves the implementation details up to the class that *implements* the interface. In short, COM makes a clear distinction between the interfaces and the objects that implement these interfaces. When working with COM it is important to understand how COM views certain common terms. They are:

1. Interface

2. Objects

3. Globally Unique Identifier

4. COM Libraries

All of these together provide a framework-like implementation to COM specifications. Let's look at the details to understand how these provide the framework-like implementation.

1. Interface:

By definition, "Interface is a class in which all the methods are pure virtual methods." When seen in accordance with the COM specification, interface is a strongly typed "contract" between software components. This contract provides a small but useful set of methods. In essence COM treats interfaces as the definition of an expected behavior and responsibilities. One of the main characteristic features of a COM interface is that if new methods are added to the interface, it becomes a new interface.

The convention for naming interfaces is that the names are preceded with an "I." The best example is the **IUnkown** interface. It is defined by COM to implement some essential functionality. All COM components are required to implement the **IUnkown** interface. The specialty of the **IUnkown** interface is that by implementing it, COM objects can control their own lifespan. Thus through the **IUnknown** interface, COM provides the basis of a likely framework.

2. Objects:

In OOPs, an object is an instance of a class. But in COM, an object is a piece of compiled code that provides some specific service to rest of the software system. As described under Interface above, the COM objects (or components as they are generally called) support the **IUnknown** interface. The main difference between simple objects and COM components is that unlike other objects, one COM component can never access another COM component directly. The access must be through interface pointers.

COM defines many interfaces. One of the implemented interfaces is the IDispatch interface. It allows COM components to be called from scripting environments. Also, the components that implement IDispatch are known as automation components. The reason is that by using such a COM object, OLE automation can be implemented. As I said in the Interface section, each component has a unique identifier. It is known as GUID.

3. Globally Unique Identifier:

Globally Unique Identifier, or GUID for short, is a 128 bit integer that is guaranteed to be unique across all the COM components used the world over and created in the past or present. The human readable names are provided just for convenience, and their scope is local, which prevents accidental connections to wrong interfaces or methods. When a GUID refers to the class, it is known as CLSID, whereas GUID referring to the interfaces are called IID. The GUIDs can be created using tools such as uuidgen, provided by Microsoft.

4. COM Libraries:

COM library is a system component that provides the mechanics of COM as well as encapsulating the leg work associated with launching COM components and communicating with other components. Generally when a COM component has to be launched, the CLSID of the component is passed to the COM library, which in turn uses the CLSID to look up the associated server code in the registry and launches the component using either **CoRegisterClassFactory** or **DllGetClassFactory.**

That's about all the overview about COM. The above described four aspects form the basis for any type of COM application. In the next section I will focus on how Python maps the COM techniques into its domain.

{mospagebreak title=Calling COM components from Python}

Now that the basis of COM is clear, it's time to see how to call COM components from the Python environment. This discussion will be focused on automation objects. I will be using the Word automation object as example. There are three major steps in calling a COM component from Python, which are:

1. Importing the win32 module.

2. Creating/launching the COM component.

3. Invoking methods on the object.

All of the above are the common steps to be followed in any type of programming.
Still, while calling COM objects from the Python environment, the details of the steps are not that common.

1. Importing win32com package:

All the APIs required to work with the COM object are available in the win32com package. If COM automation objects have to be called, then the **client** module of the win32 module has to be used. If a server has to be created, then the **server** module has to be used. Hence to use COM automation objects, the import statement will be:

import win32com.client

2. Creating/launching the COM component:

As discussed in the previous section, COM components are registered with unique CLSIDs and human readable ProgIDs. For example, MS Word defines **Word.Application** as its ProgID. Using the ProgID, COM objects can be created. To use it with Python, the ProgID has to be passed to the Dispatch method of the **client** module. For example, to launch a Word automation object, the syntax would be:

```
import win32com.client
word = win32com.client.Dispatch("Word.Application")
```

word is now an object that represents MS Word. Now that we have an object of COM component, let's look at how to call the methods.

3. Invoking methods on the object:

Before actually invoking methods of the MS Word object, it would be better to understand the binding that takes place while calling a method of COM object. There are two types of binding associated with COM components (as is usual in the object-oriented world): late binding and early binding.

With compiled languages, late binding refers to runtime binding of values to the attributes of the object, whereas early binding refers to the value/method resolution at compile time. However, as Python is a scripted language, there is a bit of a twist.

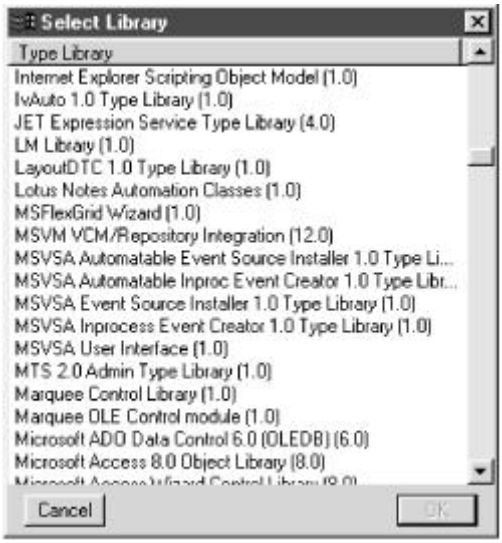
Let's take a look at late binding. In the case of COM it is known as Late Bound Automation. It means that Python doesn't have knowledge of the property or method available for an object in advance. So whenever a property is referenced, the object is queried for that property. The following example will make this more clear. The Word automation object contains a property that decides the visibility of the object. The datatype is Boolean. So when Python sees this code:

```
word.Visible=1
```

it queries the **word** object to determine whether there is a property named visible. If it is there, its value would be set to 1. This is Late Bound Automation. And it is the default behavior of COM object launched by the Python environment. In the terminology of *PythonCOM*, which is the base package of PythonWin, this behavior is known as *dynamic dispatch.*

Now let's look at early binding. It is the opposite of late binding. If the supporting source packages of any COM object are available, then there is no requirement for dynamic dispatch. For this to happen, the PyMake utility has to be used. It generates Python source code supporting the COM interface using COM type library. The whys and wherefores will be discussed in the future. There is no change in semantics in the case of Early Binding. The only change would be in the internal working and improved execution.

To create a early-bound interface, start the PyMake from the win32com directory. Once started, it would present a list of objects MakePy can use to support early binding. Selecting the Word entry from the list and clicking the OK button would cause the support interface to be generated. The list presented by PyMake would be something like this:



So each COM object has its own methods. For example, to invoke the *open* method of word object, the statement would be:

word. Open(doc_template_name)

where doc_template_name is the name of the document to be opened. If Python source is available for the interface, it would be used. Otherwise dynamic dispatch would be used. That brings us to the end of this section. In the next section I will be applying the concepts and techniques discussed until now to create a MS Word Mail-Merge application using COM.

{mospagebreak title=COM Programming in Python, in the Real World}

The application to be developed will take the advantage of early binding. So for the application to work efficiently, having the supporting interface generated would be a better choice. Moving on to the application, I will be developing the mail-merge application (without any GUI) by using the mail-merge functionality provided by the Word automation. The application requires the following to work:

1. A doc template for Word.

2. The addresses which have to be merged with the template in CSV format

3. The document where the merged document must be saved.

Now let's look at the code. First, the imports:

import os, win32com.client

The os package is required to determine the absolute path of all the required files, which is done in the next statements path using the path module:

```
import os, win32com.client

doc_template_name = os.path.abspath('template.doc')
data_source_name = os.path.abspath('record23.csv')
doc_final_name = os.path.abspath('record23.doc')
```

then create an object of Word application:

```
import os, win32com.client

doc_template_name = os.path.abspath('template.doc')
data_source_name = os.path.abspath('record23.csv')
doc_final_name = os.path.abspath('record23.doc')

app = win32com.client.Dispatch("Word.Application")
```

The next step is to obtain the document object by calling the open method of Documents object (which is contained within the returned app object) with the doc template's name.

```
import os, win32com.client

doc_template_name = os.path.abspath('template.doc')
data_source_name = os.path.abspath('record23.csv')
doc_final_name = os.path.abspath('record23.doc')

app = win32com.client.Dispatch("Word.Application")
doc_template = app.Documents.Open(doc_template_name)
```

Next we must obtain reference to the MailMerge object, which provides the mail-merge functionality. Then we must open the data source for addresses using the OpenDataSource() method of the MailMerge object. And then we will merge the data source with the document. For brevity only one record is being merged:

```
import os, win32com.client

doc_template_name = os.path.abspath('template.doc')
data_source_name = os.path.abspath('record23.csv')
```

```
doc_final_name = os.path.abspath('record23.doc')

app = win32com.client.Dispatch("Word.Application")
doc_template = app.Documents.Open(doc_template_name)

mm = doc_template.MailMerge

#attach data source to template
mm.OpenDataSource(data_source_name)

#merge just one record – this step may be redundant
mm.DataSource.FirstRecord = 1
mm.DataSource.LastRecord = 1

#send the merge result to a new document
mm.Destination = win32com.client.constants.wdSendToNewDocument

#merge
mm.Execute()
```

Before calling the execute method to merge the documents, the destination has to be specified, which is done using the COM constant- **win32com.client.constants.wdSendToNewDocument.**

Next we must obtain the reference of the newly merged document and then save it to the required file:

```
import os, win32com.client

doc_template_name = os.path.abspath('template.doc')
data_source_name = os.path.abspath('record23.csv')
doc_final_name = os.path.abspath('record23.doc')

app = win32com.client.Dispatch("Word.Application")
doc_template = app.Documents.Open(doc_template_name)

mm = doc_template.MailMerge

#attach data source to template
mm.OpenDataSource(data_source_name)
```

```
#merge just one record – this step may be redundant
mm.DataSource.FirstRecord = 1
mm.DataSource.LastRecord = 1

#send the merge result to a new document
mm.Destination = win32com.client.constants.wdSendToNewDocument

#merge
mm.Execute()

#apparently app.Documents is like a stack
doc_final = app.Documents[0]

#save our new document
doc_final.SaveAs(doc_final_name)

#cleanup
doc_final.Close()
doc_template.Close()
app.Quit()
```

Once the task is done, the cleaning up has to be done by calling close on the final document object and doc template, apart from calling quit on the Word object to stop processing.

That brings us to the end of this discussion. What I have discussed here is just the tip of the iceberg. What else can be achieved using win32com package will be discussed in the future. Till then…

Related Threads

Related Articles

Python Big Data Company Gets DARPA Funding ›

Python 32 Now Available ›

Final Alpha for Python 3.2 is Released ›

Documenting Your Project (edited – lorraine) ›

This entry was posted in Python. Bookmark the permalink.

APRajshekhar

[gp-comments width="770" linklove="off"]

← Previous Post Next Post →