# 

# Python and COM - Implementation Details

## Introduction

This document describes the technical implementation of the COM support in Python. It is primarily concerned with the underlying C++ interface to COM, although general Python issues are touched.

This document is targeted at people who wish to maintain/enhance the standard COM support (typically by writing extension modules). For information on using Python and COM from a Python programmers perspective, please see the [documentation index](http://docs.google.com/docindex.html).

## General COM Support.

COM support in Python can be broken into 2 general areas - C++ support, and Python support. C++ support exists in the core PythonCOM module (plus any PythonCOM extension modules). Python support exists in the .py files that accompany the core module.

## Naming Conventions

The naming conventions used by Python code will be:

* The Python "New Import" (ni) module will be used, allowing packages, or nested modules.
* The package name will be "win32com".
* The core module name will be "pythoncom" (ie, "win32com.pythoncom")

The rest of the naming conventions are yet to be worked out.

## Core COM support.

This section is involved with the core C++ support in "pythoncom".

The organisation of PythonCOM support falls into 3 discrete areas.

### COM Client Support

This is the ability to manipulate other COM objects via their exposed interface. This includes use of IDispatch (eg using Python to start Microsoft Word, open a file, and print it.) but also all client side IUnknown derived objects fall into this category, including ITypeLib and IConnectionPoint support.

### COM Server Support

This is ability for Python to create COM Servers, which can be manipulated by another COM client. This includes server side IDispatch (eg, Visual Basic starting a Python interpreter, and asking it to evaluate some code) but also all supported server side IUnknown derived classes.

### Python/COM type and value conversion

This is internal code used by the above areas to managed the conversion to and from Python/COM types and values. This includes code to convert an arbitrary Python object into a COM variant, manages return types, and a few other helpers.

## COM Structures and Python Types

OLE supports many C level structures for the COM API, which must be mapped to Python.

### VARIANT

Variants are never exposed as such to Python programs. The internal framework always converts all variants to and from Python types. In some cases, type descriptions may be used, which force specific mappings, although in general the automatic conversion works fine.

### TYPEDESC

A tuple, containing the elements of the C union. This union will be correctly decoded by the support code.

### ELEMDESC

A tuple of TYPEDESC and PARAMDESC objects.

### FUNCDESC

A funcdesc is a large and unwieldy tuple. Documentation to be supplied.

### IID/CLSID

A native IID in Python is a special type, defined in pythoncom. Whenever a CLSID/IID is required, typically either an object, a tuple of type "iii(iiiiiiii)" or string can be used.

Helper functions are available to convert to and from IID/CLSID and strings.

## COM Framework

Both client and server side support have a specific framework in place to assist in supporting the widest possible set of interfaces. The framework allows external extension DLLs to be written, which extend the interfaces available to the Python user.

This allows the core PythonCOM module to support a wide set of common interfaces, and other extensions to support anything obscure.

### Client Framework

#### QueryInterface and Types

When the only support required by Python is IDispatch, everything is simple - every object returned from QueryInterface is a PyIDispatch object. But this does not extend to other types, such as ITypeLib, IConnectionPoint etc., which are required for full COM support.

For example, consider the following C++ psuedo-code:

IConnectionPoint \*conPt;

someIDispatch->QueryInterface(IID\_IConnectionPoint, (void \*\*)&conPt);

// Note the IID\_ and type of the \* could be anything!

This cast, and knowledge of a specific IID\_\* to type must be simulated in Python.

Python/COM will therefore maintain a map of UID's to Python type objects. Whenever QueryInterface is called, Python will lookup this map, to determine if the object type is supported. If the object is supported, then an object of that type will be returned. If the object is not supported, then a PyIUnknown object will be returned.

Note that PyIDispatch will be supported by the core engine. Therefore:

>>> disp=someobj.QueryInterface(win32com.IID\_Dispatch)

will return a PyIDispatch object, whereas

>>> unk=someobj.QueryInterface(SomeUnknownIID) # returns PyIUnknown

>>> disp=unk.QueryInterface(win32com.IID\_Dispatch)

>>> unk.Release() # Clean up now, rather than waiting for unk death.

Is needed to convert to an IDispatch object.

#### Core Support

The core COM support module will support the IUnknown, IDispatch, ITypeInfo, ITypeLib and IConnectionPointContainer and IConnectionPoint interfaces. This implies the core COM module supports 6 different OLE client object types, mapped to the 6 IID\_\*'s representing the objects. (The IConnection\* objects allow for Python to repsond to COM events)

A psuedo-inheritance scheme is used. The Python types are all derived from the Python IUnknown type (PyIUnknown). Therefore all IUnknown methods are automatically available to all types, just as it should be. The PyIUnknown type manages all object reference counts and destruction.

#### Extensibility

To provide the above functionality, a Python map is provided, which maps from a GUID to a Python type object.

The advantage of this scheme is an external extension modules can hook into the core support. For example, imagine the following code:

>>> import myextracom # external .pyd supporting some interface.

# myextracom.pyd will do the equivilent of

# pythoncom.mapSupportedTypes(myextracom.IID\_Extra, myextracom.ExtraType)

>>> someobj.QueryInterface(myextracom.IID\_Extra)

Would correctly return an object defined in the extension module.

### Server Framework

#### General Framework

A server framework has been put in place which provides the following features:

All Interfaces provide VTBL support - this means that the Servers exposed by Python are callable from C++ and other compiled languages.

Supports full "inproc" servers. This means that no external .EXE is needed making Python COM servers available in almost all cases.

An extensible model which allows for extension modules to provide server support for interfaces defined in that module. A map is provided which maps from a GUID to a function pointer which creates the interface.

### Python and Variant Types Conversion

In general, Python and COM are both "type-less". COM is type-less via the VARIANT object, which supports many types, and Python is type-less due to its object model.

There are a number of areas where Python and OLE clash.

#### Parameters and conversions.

For simple calls, there are 2 helpers available which will convert to and from PyObjects and VARIANTS. The call to convert a Python object to a VARIANT is simple in that it returns a VARIANT of the most appropriate type for the Python object - ie, the type of the Python object determines the resulting VARIANT type.

There are also more complex conversion routines available, wrapped in a C++ helper class. Typically, these helpers are used whenever a specific variant type is known (eg, when an ITypeInfo is available for the object being used). In this case, all efforts are made to convert the Python type to the requested variant type - ie, in this situation, the VARIANT type determines how the Python object is coerced. In addition, this code supports the use of "ByRef" and pointer paramaters, providing and freeing any buffers necessary for the call.