An Overview of the PythonCOM & Win32COM Libraries

When a new installation of Python is placed in a system, there aren't many Windows-specific functionalities that come along with it. To increase the array of tools in your arsenal, a set of extensions were developed to fill in the missing pieces; these extensions came to be known as the Python for Windows Extensions.

With these extensions, we can now access different window-specific features like creating and running COM objects, direct access to the Win32 API, Windows GUI, and much more. In this tutorial, we will focus on one of two of these extensions, the PythonCOM and Win32COM library.

Overview of COM Objects

Being that we will be working with COM objects throughout this tutorial, it is vital to have an idea of how they work and what they are. Microsoft introduced the Component Object Model (COM) in 1993; this new framework would allow the reuse of objects (components) regardless of the language they were implemented in. For example, I might write an object in C++, but I would be able to use and interact with it inside of another programming language, like Python.

This is one of the best features of a COM object; we can write objects in one language and use them in another without having to know anything about how they were implemented. How is this achieved you may ask? Through the use of an **interface**.

Microsoft does a great job of explaining an interface, so let's leverage what they have. **An interface defines a set of methods that an object can support, without dictating anything about the implementation.** The interface marks a clear boundary between code that calls a method and the code that implements the method. **In computer science terms, the caller is decoupled from the implementation.**

An Introduction to the Win32 COM Library

Have you ever wanted to access different applications from Python? For example, maybe you have some data in Excel that would like to load into Python for further manipulation. To the surprise of many, the entire VBA model is accessible from Python. However, to access this model, we need to access it by using COM objects.

This can quickly be done by leveraging a built-in library for Windows's users. This library is called the **Win32Com Library** and with it, we have a framework where we can use and publish our COM objects regardless of the language they were implemented in. **To be clear, with the Win32COM library we can access all sorts of different COM objects on our system, not just Office Applications. For example, we can access Adobe Applications, Visual Studios, and many more.**

A Simple Example With Excel

Let's get a feel for the Win32COM library by walking through a simple example that works with the Excel Application from Python. This way, we can see how to call methods and properties of the different COM object and understand how tasks are taking place behind the scenes. This will not go into every single detail, but more set the stage for further discussion down the road.

Section One: Creating an Instance Of Excel

The first thing we need to do is create an instance of our Excel application because we are assuming that we don't have an active instance running. The `Win32com.client` module makes this easy with the use of `Dispatch()` method, which creates a new COM object that will represent our Excel Application.

How do we tell `Win32COM` which COM object to create? Well, there are two ways we can do this, either through the use of a ProgID or a CLISD.

For completeness, here are the definitions for the CLSID and ProgID provided by Microsoft. **A programmatic identifier (ProgID)** is a registry entry that can be associated with a CLSID. Like the CLSID, the ProgID identifies a class but with less precision because it is not guaranteed to be globally unique.**The CLSID** is a globally unique identifier that identifies a COM class object.

The main take away from this is that the CLSID is unique for every COM object, whereas the ProgID can technically appear more than once, for example, I could have multiple versions of Excel installed on my system.

For Excel, the ProgID and CLSID appear as the following.

- The **ProgId** being **Excel.Application**
- The **CLSID** being **{00024500-0000-0000-C000-000000000046}**

Down below, I have provided both methods for creating an instance of the Excel Application. Now by default just because the object has been created doesn't mean we can see it. However, because we have a reference to the COM object now, we can start invoking different methods and properties. Let's set the `Visible` property to `True`, so it makes the application visible.

In [4]:
```
# import the library
import win32com.client

# create an instance of Excel using the ProgID
ExcelApp = win32com.client.Dispatch("Excel.Application")
ExcelApp.Visible = True

# create an instance of Excel using the CLSID
# ExcelAppCLSID = win32com.client.Dispatch("{00024500-0000-0000-C000-000000000046}")
# ExcelAppCLSID.Visible = True

display(ExcelApp)
# display(ExcelAppCLSID)
```

```
<win32com.gen_py.Microsoft Excel 16.0 Object Library._Application instance at 0x2575380955264>
```

Section Two: Add Objects To Our Application

Okay we have our Excel Application, and it behaves exactly how it would behave in our VBA model, so let's add some more objects we can modify. The first object we will create is a Workbook object, this means we will go into our `ExcelApp`, call the `Workbooks` Collection property, and then the `Add()` method to add a new workbook to our collection. Let's do the same for a worksheet, but this time we have to go to our `ExcelWorkbook` object and then the `Worksheets` collection. We have to do this because the `Worksheets` collection exists in our `Workbook` object. Finally, let's define a range of cells and store them in a variable. From here, call the `Value` property and set the value of each cell to `200`.

What exactly is happening behind the scenes for this to happen? Well we explore it in more detail later but at a high level, we are interacting with our **DispatchInterface** and asking it to provide some information in the form of different methods it makes available (GetIDsOfNames, Invoke). For example, when we added a new workbook it did something like the following:

- **propertyIDForWorkbook** = ExcelApp -> GetIDsOfNames("Workbooks")
- **WorkbookCollectionObject** = ExcelApp -> Invoke(propertyIDForWorkbook, DISPATCH_PROPERTYGET)
- **methodIDForAdd** = WorkbookCollectionObject -> GetIDsOfNames("Add")
- **AddedWorkbook** = WorkbookCollectionObject -> Invoke(methodIDForAdd, DISPATCH_METHOD)

Don't worry if this seems confusing at first, we will implement this later, but hopefully, you caught on to a pattern. We ask if the property/method exists for that particular object using the **GetIDsOfNames** if it does return the ID, and finally invoke the method/property using the returned ID and **Invoke Method**.

In [5]:
```
#create a new workbook in the Excel app
ExcelWorkbook = ExcelApp.Workbooks.Add()

#create a new sheet in the workbook
ExcelWrkSht = ExcelWorkbook.Worksheets.Add()

# set a reference to a range of cells & then set their value to 200.
ExcRng1 = ExcelWrkSht.Range("A1:A10")
ExcRng1.Value = 200

display(ExcelWorkbook)
display(ExcelWrkSht)
display(ExcRng1)
display(ExcRng1.Value)
```

```
<win32com.gen_py.None.Workbook>

<win32com.gen_py.Microsoft Excel 16.0 Object Library._Worksheet instance at 0
x2575387092752>

<win32com.gen_py.Microsoft Excel 16.0 Object Library.Range instance at 0x2575
387094040>

((200.0,),
 (200.0,),
 (200.0,),
 (200.0,),
 (200.0,),
 (200.0,),
 (200.0,),
 (200.0,),
 (200.0,),
 (200.0,))
```

Section Three: Looping Through Cells

Now that we have a range of cells let's explore how we can loop through them. Well, the first thing we need to recognize is that our large range of cells is made up of smaller individual cells that have their unique properties and methods. In this example, I'll define a second range of cells but this time using the `Cells` object. Once, I have my range of cells; I'll define the value of those cells using the `Value` property. From here, we will print out some info about our range of cells, like the number of cells in that particular range. Then we will use a for loop to loop through each cell, display the address, or display the value of that cell. I also chose, to apply to separate loops because I want to drive home the concept that you can use either loop through each cell or define a range of numbers that goes from 0 to the number of cells in our range.

In [6]:
```
# set a reference to a range of cells
Cell1 = ExcelWrkSht.Cells(1,4)
Cell2 = ExcelWrkSht.Cells(7,4)
ExcRng2 = ExcelWrkSht.Range(Cell1,Cell2)
ExcRng2.Value = 100

# count the cells in our range
print(ExcRng2.Cells.Count)

# get the cell address using a for loop
for cellItem in ExcRng2:
    print(cellItem.Address)

# get the cell value using a for loop but this time using a range object.
for i in range(len(ExcRng2)):
    print(cellItem.Value)
```

```
7
$D$1
$D$2
$D$3
$D$4
$D$5
$D$6
$D$7
100.0
100.0
100.0
100.0
100.0
100.0
100.0
```

Interfaces

Remember up above where we talked about how we interact with COM objects? It was through the use of Interfaces, in this section we will cover interfaces in more detail. The interface is simple terms, describes how an object will behave, however, it is up to the object itself to implement the behavior. There are multiple types of interfaces, the base interface that derives all other interfaces is the `IUnkown` interace. If we want to return a specific interface we can use the `QueryInterface()` method to return a specific interface. This is done through the use of a IID, which is a unique 128-bit GUID known as an interface ID (IID) that is unique for each COM interface.

To meet the requirements of higher-level languages, the IDispatch interface was defined to allow the exposure of an object model that contain both the methods and properties of an object and defines a mechanism that will allow us to determine which methods and properties are available during runtime.

I do want to note here, that IDispatch is simply the tool that is exposing the object model, which is it's own object separate from the IDispatch interface. There are two methods that allow this to take place, `GetIDsOfNames` and `Invoke()`. The `GetIDsOfNames` allows us to ask a given object whether a they have a given property or method. If it does, then it returns an integer ID for the method or property.

With the ID we can then use the `Invoke()` method to pass through the ID and then invoke the action which is either calling the method or getting/setting the property.

Let's see how this would all look in code and also cover a few of the remaining topics that we need to know with interfaces.

Okay, first thing we are going to do is impor the `pythoncom` which is a module, encapsulating the OLE automation API. The API is a feature that programs use to expose their objects to development tools, macro languages, and other programs that support Automation.

From here, we will call the `new()` which creates a new instance of an OLE automation server, in this case Excel. Finally, we display the type of object which is a PyIDispatch, a python version of an IDispatch interface for our OLE automation client.

In [7]:
```
import pythoncom

# create a new Excel COM object.
xlCom = pythoncom.connect('Excel.Application')

# let's see what type of Object it is
display(type(xlCom))
```

```
PyIDispatch
```

Okay so we have an IDispatch interface object which exposes the object model for our Excel Application. Let's start by seeing what methods this PyIDispatch object has.

In [8]:
```
# let's take a look at our PyIDispatch Object.
help(xlCom)
```

```
Help on PyIDispatch object:

class PyIDispatch(PyIUnknown)
 |  Define the behavior of a PythonCOM Interface type.
 |
 |  Method resolution order:
 |      PyIDispatch
 |      PyIUnknown
 |      interface-type
 |      object
 |
 |  Methods defined here:
 |
 |  GetIDsOfNames(...)
 |
 |  GetTypeInfo(...)
 |
 |  GetTypeInfoCount(...)
 |
 |  Invoke(...)
 |
 |  InvokeTypes(...)
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __setattr__(self, name, value, /)
 |      Implement setattr(self, name, value).
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

```
|
|  ----------------------------------------------------------------------
|  Methods inherited from PyIUnknown:
|
|  QueryInterface(...)
```

There are a few methods here, but I only want to talk about a few of them, and then we will come back to the rest when we need additional information about our object. The first one we will discuss is `GetIDsOfNames` which takes in a string that is either is a property or method of our object. If there is one, it returns the DISPID, which is the ID for that specific method or property. Formal documentation can be found here:

- *Link to Win32COM:* http://timgolden.me.uk/pywin32-docs/PyIDispatch__GetIDsOfNames_meth.html (http://timgolden.me.uk/pywin32-docs/PyIDispatch__GetIDsOfNames_meth.html)
- *Link to Microsoft:* https://docs.microsoft.com/en-us/windows/desktop/api/oaidl/nf-oaidl-idispatch-getidsofnames (https://docs.microsoft.com/en-us/windows/desktop/api/oaidl/nf-oaidl-idispatch-getidsofnames)

GetIDsOfNames:

The (int, ...) or int = GetIDsOfNames(name)

Description Get the DISPID for the passed names.

Parameters

- *name:* string or `[name1, name2, name3]`
- *desc:* A name to query for

Comments Currently, the LCID cannot be specified, and LOCALE_SYSTEM_DEFAULT is used.

Return Value If the first parameter is a sequence, the result will be a tuple of integers for each name in the series. If the first parameter is a single string, the result is a single integer with the ID of the requested item.

In [9]:
```
# let's grab the visible property
visible_id = xlCom.GetIDsOfNames('Visible')
display(visible_id)

# let's grab the Quit method.
quit_id = xlCom.GetIDsOfNames('Quit')
display(quit_id)
```

```
558

302
```

Okay so we have an IDispatch interface object which exposes the object model for our Excel Application, and we have a few IDs we can pass along in our Invoke methods, so let's start invoking methods and properties and seeing what we get back. The PyIDisptach has two different invoke methods, `Invoke` and `InvokeTypes`, there isn't a difference between these two other than the fact that with `InvokeTypes` you can define the return type. You'll see this later when talking about early versus late binding in COM. As always, here is some documentation:

- *Link to Win32COM:* http://timgolden.me.uk/pywin32-docs/PyIDispatch__Invoke_meth.html (http://timgolden.me.uk/pywin32-docs/PyIDispatch__Invoke_meth.html)
- *Link to Microsoft:* https://docs.microsoft.com/en-us/windows/desktop/api/oaidl/nf-oaidl-idispatch-invoke (https://docs.microsoft.com/en-us/windows/desktop/api/oaidl/nf-oaidl-idispatch-invoke)

InvokeTypes

Syntax: expression.InvokeTypes(dispid, lcid , wFlags , resultTypeDesc , typeDescs , args)

Description Invokes a DISPID, using the passed arguments and type descriptions.

Parameters

name: dispid

type: int

desc: The dispid to use. Typically this value will come from PyIDispatch::GetIDsOfNames or from a type library.

name: lcid

type: int

desc: The locale id to use.

name: wflags

type: int

desc: The flags for the call. The following flags can be used.

name: resultTypeDesc

type: tuple

desc: A tuple describing the type of the result.

name: typeDescs

type: tuple

desc: A sequence of tuples describing the types of the parameters for the function. See the comments for more information.

name: args

type: object

desc: The arguments to be passed.

Comments: The Microsoft documentation for IDispatch should be used for all params except `resultTypeDesc` and `typeDescs` .

name: resultTypeDesc

desc: describes the return value of the function, and is a tuple of (type_id, flags).

name: typeDescs

desc: describes the type of each parameters, and is a list of the same (type_id, flags) tuple.

```
item: type_id
desc: A valid "variant type" constant (eg, VT_I4 | VT_ARRAY, VT_DATE, etc - see VAR
IANT at MSDN).

item: flags
desc: One of the PARAMFLAG constants (e.g., PARAMFLAG_FIN, PARAMFLAG_FOUT, etc. - s
ee PARAMFLAG at MSDN).
```

Return Value: object

In [10]:
```
# does the Excel Application have a property called "Range"?
RangeID = xlCom.GetIDsOfNames('Range')
print(RangeID)

# define the LCID (The locale id) to use.
LCID = 0x0
print(LCID)

# define the flags for call, in this case we are getting a property so we will
use DISPATCH_PROPERTYGET
wFlags = pythoncom.DISPATCH_PROPERTYGET
print(wFlags)

# define the resultTypeDesc, this particular method returns a dispatch object
 (pythoncom.VT_DISPATCH) and
# and the type of the contained field is undefined (pythoncom.VT_EMPTY).
resultTypeDesc = (pythoncom.VT_DISPATCH, pythoncom.VT_EMPTY)
print(resultTypeDesc)

# define the type descriptions of the arguments we are passing in.
# 2 arguments - given by two tuples
#               first and second argument will be a variant (pythoncom.VT_VARI
ANT), and the parameter passes
#               information from the caller to the callee pythoncom.PARAMFLAG_
FIN
typeDescs = ((pythoncom.VT_VARIANT, pythoncom.PARAMFLAG_FIN), (pythoncom.VT_VA
RIANT, pythoncom.PARAMFLAG_FIN))
print(typeDescs)

# define our two arguments, the range property takes two arguments cell1 and c
ell.
cell1 = 'A1'
cell2 = 'A2'

# print everything together.
print((RangeID, LCID, wFlags, resultTypeDesc, typeDescs, cell1, cell2))

# grab the range object
xlCom.InvokeTypes(RangeID, LCID, wFlags, resultTypeDesc, typeDescs, cell1)
```

```
197
0
2
(9, 0)
((12, 1), (12, 1))
(197, 0, 2, (9, 0), ((12, 1), (12, 1)), 'A1', 'A2')
```

Out[10]:
```
<PyIDispatch at 0x000002579B9558E0 with obj at 0x000002579F918C28>
```

Okay so in that example we were working with a property, how would it look if we wanted to use a method? Well, for the most part, the structure is precisely the same; we just have to change the arguments.

In [ ]:
```
# let's grab the DISPID of the Undo method.
UndoID = xlCom.GetIDsOfNames('Undo')

# define the LCID (The locale id) to use.
LCID = 0x0

# define the flags for call, in this case we are using a method so we will use
DISPATCH_METHOD
wFlags = pythoncom.DISPATCH_METHOD

# define the resultTypeDesc, this particular method does not return anything
 (pythoncom.VT_VOID or 24) and
# and the parameter passes information from the caller to the callee (pythonco
m.PARAMFLAG_FIN or 1).
# could also be written as (pythoncom.VT_VOID, pythoncom.PARAMFLAG_FIN)
resultTypeDesc = (24, 1)

# proof they are the same.
display((pythoncom.VT_VOID, pythoncom.PARAMFLAG_FIN))

# there are no arguments to pass through in this example, so we don't need to
 define their types
# so we will pass through an empty tuple.
typeDescs = ()

# no arguments - SIMPLY A PLACE HOLDER
args = ''

# Let's put all the pieces together and invoke the method.
xlCom.InvokeTypes(UndoID, LCID, wFlags, resultTypeDesc, typeDescs)
```

Here is an example of using the `Invoke` method using the `PyIDispatch` interface.

Invoke

- Win32COM Documentation: http://timgolden.me.uk/pywin32-docs/PyIDispatch__Invoke_meth.html (http://timgolden.me.uk/pywin32-docs/PyIDispatch__Invoke_meth.html)
- Microsoft Documentation: https://docs.microsoft.com/en-us/windows/desktop/api/oaidl/nf-oaidl-idispatch-invoke (https://docs.microsoft.com/en-us/windows/desktop/api/oaidl/nf-oaidl-idispatch-invoke)

Syntax expression.Invoke(dispid, lcid , flags , bResultWanted , params, ...)

Description Invokes a DISPID, using the passed arguments.

Parameters

name: dispid

type: int

desc: The dispid to use. Typically this value will come from PyIDispatch::GetIDsOfNames or from a type library.

name: lcid

type: int

desc: The locale id to use.

name: flags

type: int

desc: The flags for the call. The following flags can be used.

name: bResultWanted

type: int

desc: Indicates if the result of the call should be requested.

name: params

type: object

desc: The parameters to pass.

Comments: If the bResultWanted parameter is False, then the result will be None. Otherwise, the result is determined by the COM object itself (and may still be None)

Return Value: object

In [ ]:
```
# let's grab the DISPID of the Range Property.
RangeID = xlCom.GetIDsOfNames('Range')

# define the LCID (The locale id) to use. We just need to define the LOCALUSER
LCID = 0x0

# define the flags for call, in this case we are using a property so we will u
se DISPATCH_PROPERTYGET
wFlags = pythoncom.DISPATCH_PROPERTYGET

# do we want the results back? False means we get NONE True means we get the R
esult back if there is one.
bResultWanted = True

# no parameters to pass - SIMPLY A PLACE HOLDER
cell1 = 'A1'
cell2 = 'A2'

# Let's put all the pieces together and invoke the method.
myRangeObject = xlCom.Invoke(RangeID, LCID, wFlags, bResultWanted, cell1, cell
2)
display(myRangeObject)
```

Here are the possible flags you can set for the calls.

---

name: DISPATCH_METHOD

desc: The member is invoked as a method. If a property has the same name, both this and the DISPATCH_PROPERTYGET flag may be set.

name: DISPATCH_PROPERTYGET

desc: The member is retrieved as a property or data member.

name: DISPATCH_PROPERTYPUT

desc: The member is changed as a property or data member.

name: DISPATCH_PROPERTYPUTREF

desc: The member is changed by a reference assignment, rather than a value assignment. This flag is valid only when the property accepts a reference to an object.

If you were like me, you saw the code up above, and you asked the question, "Well that's great and all, but what if I don't know the property or methods of the object? What if I want to see all the methods and properties?" Well, there is a way to see all the methods and properties of an object, but it requires us using a different method of our PyIDispatch interface object. The `GetTypeInfo` method returns an `PyITypeInfo` Object which contains information about the COM object. Getting the ITypeInfo object is quite easy.

With this object, we can work with a few of the methods to get additional information about our COM object. The first one we will need to use is the `GetTypeAttr` method, which returns a `TypeAttr` object that contains different information about our COM object.

In [12]:
```
# Grab the Type Info
xlTypeInfo = xlCom.GetTypeInfo()

# let's get some info about our TypeInfo Object. This will be covered in an ad
ditional Video
help(xlTypeInfo)
```

```
Help on PyITypeInfo object:

class PyITypeInfo(PyIUnknown)
 |  Define the behavior of a PythonCOM Interface type.
 |
 |  Method resolution order:
 |      PyITypeInfo
 |      PyIUnknown
 |      interface-type
 |      object
 |
 |  Methods defined here:
 |
 |  GetContainingTypeLib(...)
 |
 |  GetDocumentation(...)
 |
 |  GetFuncDesc(...)
 |
 |  GetIDsOfNames(...)
 |
 |  GetImplTypeFlags(...)
 |
 |  GetNames(...)
 |
 |  GetRefTypeInfo(...)
 |
 |  GetRefTypeOfImplType(...)
 |
 |  GetTypeAttr(...)
 |
 |  GetTypeComp(...)
 |
 |  GetVarDesc(...)
 |
 |  __delattr__(self, name, /)
 |      Implement delattr(self, name).
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __ne__(self, value, /)
```

```
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __setattr__(self, name, value, /)
|      Implement setattr(self, name, value).
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  __hash__ = None
|
|  ----------------------------------------------------------------------
|  Methods inherited from PyIUnknown:
|
|  QueryInterface(...)
```

With TypeAttr object we can now call the attributes of our object, I've put a list of all the properties that exist in the TypeAttr object if you want you can always call `help(xlTypeAttr)` to see the docstring.

In [13]:
```
# the TypeInfo object contains the GetTypeAttr, callthe GetTypeAttr() method t
o return the object.
xlTypeAttr = xlTypeInfo.GetTypeAttr()

# The IID
# At Index 0
display('-'*100)
display('The IID')
display(xlTypeAttr.iid)

# The LCID
# At Index 1
display('-'*100)
display('The LCID')
display(xlTypeAttr.lcid)

# The ID of Constructor
# At Index 2
display('-'*100)
display('The ID of the Constructor')
display(xlTypeAttr.memidConstructor)

# The ID of Destructor
# At Index 3
display('-'*100)
display('The ID of the Destructor ')
display(xlTypeAttr.memidDestructor)

# cbSizeInstance - The size of an instance of this type
# At Index 4
display('-'*100)
display('The cbSizeInstance')
display(xlTypeAttr.cbSizeInstance)

# typekind - The kind of type this information describes. One of the win32con.
TKIND_* constants.
# At Index 5
display('-'*100)
display('The TypeKind')
display(xlTypeAttr.typekind)

# cFuncs - Number of functions.
# At Index 6
display('-'*100)
display('The Number of Functions')
display(xlTypeAttr.cFuncs)

# cVars - Number of variables/data members.
# At Index 7
display('-'*100)
display('The Number of Variables/Data members')
display(xlTypeAttr.cVars)

# cImplTypes - Number of implemented interfaces.
# At Index 8
display('-'*100)
```

```
display('The Number of Implemented Interfaces')
display(xlTypeAttr.cImplTypes)

# cbSizeVft - The size of this type's VTBL
# At Index 9
display('-'*100)
display('The cbSizeVft')
display(xlTypeAttr.cbSizeVft)

# cbAlignment - Byte alignment for an instance of this type.
# At Index 10
display('-'*100)
display('The cbAlignment')
display(xlTypeAttr.cbAlignment)

# wTypeFlags - One of the pythoncom TYPEFLAG_* constants
# At Index 11
display('-'*100)
display('The wTypeFlags')
display(xlTypeAttr.wTypeFlags)

# wMajorVerNum - Major version number.
# At Index 12
display('-'*100)
display('The Major Version Number')
display(xlTypeAttr.wMajorVerNum)

# wMinorVerNum - Minor version number.
# At Index 13
display('-'*100)
display('The Minor Version Number')
display(xlTypeAttr.wMinorVerNum)

# tdescAlias - If TypeKind == pythoncom.TKIND_ALIAS, specifies the type for wh
ich this type is an alias.
# At Index 14
display('-'*100)
display('The tdescAliar')
display(xlTypeAttr.tdescAlias)

# idldescType - IDL attributes of the described type.
# At Index 15
display('-'*100)
display('The IDL Attributes')
display(xlTypeAttr.idldescType)
```

```
'-------------------------------------------------------------------------------
------------------------'

'The IID'

IID('{000208D5-0000-0000-C000-000000000046}')

'-------------------------------------------------------------------------------
------------------------'

'The LCID'

0

'-------------------------------------------------------------------------------
------------------------'

'The ID of the Constructor'

-1

'-------------------------------------------------------------------------------
------------------------'

'The ID of the Destructor '

-1

'-------------------------------------------------------------------------------
------------------------'

'The cbSizeInstance'

8

'-------------------------------------------------------------------------------
------------------------'

'The TypeKind'

4

'-------------------------------------------------------------------------------
------------------------'

'The Number of Functions'

467

'-------------------------------------------------------------------------------
------------------------'

'The Number of Variables/Data members'

0

'-------------------------------------------------------------------------------
------------------------'

'The Number of Implemented Interfaces'

1
```

```
'--------------------------------------------------------------------------
------------------------'

'The cbSizeVft'

56

'--------------------------------------------------------------------------
------------------------'

'The cbAlignment'

8

'--------------------------------------------------------------------------
------------------------'

'The wTypeFlags'

4160

'--------------------------------------------------------------------------
------------------------'

'The Major Version Number'

0

'--------------------------------------------------------------------------
------------------------'

'The Minor Version Number'

0

'--------------------------------------------------------------------------
------------------------'

'The tdescAliar'

None

'--------------------------------------------------------------------------
------------------------'

'The IDL Attributes'

(0, 0)
```

Out of all of these attributes, there is one that will be very helpful to explore as it will be used later down the road. The `cFuncs` attribute contains the number of functions in our COM object, and this is simply the number of methods and properties it has exposed to us.

In [14]:
```
# the information that we need lives in the TYPEATTR Object, so let's grab it.
xlTypeAttr = xlTypeInfo.GetTypeAttr()

# define the number of functions
numFuncs =  xlTypeAttr.cFuncs

# Let's grab all the Functions of a given object
for i in range(0, numFuncs):

    # first grab the func desc.
    xlFuncDesc = xlTypeInfo.GetFuncDesc(i)

    if xlFuncDesc.memid == 558:

        display('-'*100)

        # display the arguments of the function
        display(xlFuncDesc.args)

        # display the documenation string for that particular member id
        display(xlTypeInfo.GetDocumentation(xlFuncDesc.memid))

        # display the member ID
        display(xlFuncDesc.memid)

        # Get the name of the of function using the member ID
        display(xlTypeInfo.GetNames(xlFuncDesc.memid))

        # Indicates the type of function (virtual, static, or dispatch-only)
        display(xlFuncDesc.funckind)

        # The number of optional parameters.
        display(xlFuncDesc.cParamsOpt)

        # The function flags.
        display(xlFuncDesc.wFuncFlags)

        # The number of possible return values.
        display(xlFuncDesc.scodeArray)

        # For FUNC_VIRTUAL, specifies the offset in the VTBL.
        display(xlFuncDesc.oVft)

        # The invocation type. Indicates whether this is a property function,
 and if so, which type.
        display(xlFuncDesc.invkind)

        # The calling convention.
        display(xlFuncDesc.callconv)

        # The return type?
        display(xlFuncDesc.rettype)
```

```
'------------------------------------------------------------------------
------------------------'

()

('Visible',
 None,
 133229,
 'C:\\Program Files\\Microsoft Office\\Root\\Office16\\VBAXL10.CHM')

558

('Visible',)

4

0

0

()

2256

2

4

(11, 0, None)

'------------------------------------------------------------------------
------------------------'

((11, 1, None),)

('Visible',
 None,
 133229,
 'C:\\Program Files\\Microsoft Office\\Root\\Office16\\VBAXL10.CHM')

558

('Visible',)

4

0

0

()

2264

4

4

(24, 0, None)
```

I only took on of the items returned by FuncDesc method; in this case, I chose the `Visible` property. Let's dissect what was sent back to us.

Parameter Name: args

Return Value: ((11, 1, None),)

Description For this particular method/property, it takes a VT_BOOLEAN argument (11) the type description, takes PARAMFLAG_FIN (1) as an argument which is the idlFlag, and if PARAMFLAG_FHASDEFAULT are set, then this is the default value otherwise it's none.

Parameter Name: memid

Return Value: 558

Description: This is the DISPID for the specified property/method.

Parameter Name: funcKind

Return Value: 4

Description: Indicates the type of function (virtual, static, or dispatch-only).

```
FUNC_DISPATCH   4
pythoncom.FUNC_DISPATCH
The function can be accessed only through IDispatch.

FUNC_STATIC     3
pythoncom.FUNC_STATIC
The function is accessed by static address and does not take an implicit this point
er.

FUNC_VIRTUAL    0
pythoncom.FUNC_VIRTUAL
The function is accessed in the same way as FUNC_PUREVIRTUAL, except the function h
as an implementation.
```

Parameter Name: cParamsOpt

Return Value: 0

Description: The number of optional parameters

Parameter Name: wFuncFlags

Return Value: 0

Description: The function flags

Parameter Name: scodeArray

Return Value: ()

Description: The number of possible return values.

Parameter Name: ovft

Return Value: 2256

Description: The function flags

Parameter Name: invkind

Return Value: 2

Description: The invocation type. Indicates whether this is a property function, and if so, which type. In this case it's a INVOKE_PROPERTYGET (2)

```
INVOKE_PROPERTYGET      2
pythoncom.INVOKE_PROPERTYGET

INVOKE_FUNC             1
pythoncom.INVOKE_FUNC

INVOKE_PROPERTYPUT      4
pythoncom.INVOKE_PROPERTYPUT

INVOKE_PROPERTYPUTREF   8
pythoncom.INVOKE_PROPERTYPUTREF
```

Parameter Name: callconv

Return Value: 4

Description: The calling convention. In this case I believe this would be a STDCALL (4).

Parameter Name: rettype

Return Value: (11, 0, None)

Description: The return type which in this case is a VT_BOOLEAN (11), and has a IDLFlag of PARAMFLAG_NONE (0), finally does not return an object (None)

Okay, so we now know how to get all the information we need to invoke any method or property we need. This is very useful, and in fact, these different objects are leveraged inside the Win32COM modules directly when it comes to auto generating Python code that will be used for early binding in Win32COM.

Early Versus Late Binding

This leads to our next topic, early versus late binding inside of Win32COM. What is the difference between these two methods, well in simple terms when we are talking about early versus late binding all we are talking about is whether the programming language we are using has any upfront knowledge of the object. In other words, does the programming language know any of the methods and properties of the object at hand? If we do have upfront knowledge, we know all the methods and properties of the object; then the object was implemented using early-binding. If we don't have any upfront knowledge, we know none of the methods or properties of the object; then the object was implemented using late binding.

By default,the `win32com.client` package uses late-bound automation when using objects. Here is a simple way to tell if the object was implemented using late-binding.

In [33]:
```
import win32com.client

# Dispatch a new COM Object
PPTApp = win32com.client.Dispatch('PowerPoint.Application')
#3E65FF
# display it to the user
PPTApp
```

Out[33]:
```
<COMObject PowerPoint.Application>
```

As you can see, all Python knows is that the object is a COM Object and that it has a Program ID called `PowerPoint.Application`. However, this is all that Python knows about the object, not any of the methods or properties.

If we want, we can make Python have upfront knowledge of our COM object, but it will require us using early-bound automation. **Now, this is easy to do in Python because we call the `gencache` module and then the `EnsureDispatch` method inside that module.** However, a lot of the heavy lifting to make early-bound automation possible is done behind the scenes without us even knowing it. To give you a better understanding of this module, I think it pays dividends to explore what happens during this process.

Inside the PythonCOM module, there is a script called MakePy, which is a utility to support early-bound automation. This script leverages the COM Type Library object explored up above to generate Python source code that supports the COM object it was called on. In other words, this utility goes out and collects all the information on our COM object, converts it into Python source code, and stores it in individual scripts that we can call using the `win32com` library.

You might ask, why would I want to use early-bound automation?

1. It's faster for the Python Interface
2. Constants defined by the Type Library (xlDown) are made available to the Python program. This makes our code a lot more readable.

Now that we know the advantages of using early-bound automation let's see how to call the MakePy utility. Your first option is to use the PythonWin application, which is installed when you install the `win32` library for Python. Finding this application can be found in a few steps:

1. Type "PythonWin" into your Windows' search bar.
2. Open the application.
3. Once open, navigate the "Tool" button on the navigation bar. A new window will open up.
4. Select the Reference library you would like to call the MakePy utility on, and press Ok.
5. The library will be made; any errors encountered will be printed on the terminal below.

The second, and the more likely method you will use is to call the `gencache` module, and then the `EnsureDispatch()` function in that module. In whole, this how it would look:

```
xlApp = win32com.client.gencahce.EnsureDispatch('Excel.Application')
```

The `gencache` module runs the MakePy utility behind the scenes, so that early bound automation may be achieved. Pass through the CLSID or the ProgID. Regardless of which method you use, Python source code files will be created in each instance. To find these files, we can navigate to them. Just as a warning, I can't guarantee that the files will exist in the same location for every type of user this is simply the location that was leveraged on my system.

When the MakePy utility is run, a new folder called `gen_py` is created in your Local folder. Here is the path to it on my system:

```
C:\Users\{YOUR NAME}\AppData\Local\Temp\gen_py
```

Inside this folder is all the Python generated code made by the MakePy utility. You may find multiple folders, for example in mine there is a 3.5 & 3.7 folder, my best guess is that these are representing the different versions of Python on your system, but I can't say for sure. The generated python files exist in the 3.7 one for my system. I have seen some situations where people's numbering have been different.

When I run the MakePy utility on the PowerPoint application, it generates a folder with the following path.

`C:\Users\{YOUR NAME}\AppData\Local\Temp\gen_py\3.7\91493440-5A91-11CF-8700-00AA0060263Bx0x2x12`

The folder name may look confusing, but it's just the CLSID for the given object, in this case, the PowerPoint application. Inside the folder, depending on the object, you will see multiple python scrips which contain all the information about the given objects that exist in that particular object hierarchy. For example, in mine, there is one for the `Shapes` collection and the `Presentations` collection.

If you open up the `_Presentations.py` script, you'll see all the auto-generated code for that object. Here is just a small portion of that code.

In [ ]:
```
CLSID = IID('{91493440-5A91-11CF-8700-00AA0060263B}')
MajorVersion = 2
MinorVersion = 12
LibraryFlags = 8
LCID = 0x0

class _Presentation(DispatchBaseClass):

    CLSID = IID('{9149349D-5A91-11CF-8700-00AA0060263B}')
    coclass_clsid = IID('{91493444-5A91-11CF-8700-00AA0060263B}')

    def AcceptAll(self):
        return self._oleobj_.InvokeTypes(2115, LCID, 1, (24, 0), (),)

    def AddBaseline(self, FileName=''):
        return self._ApplyTypes_(2073, 1, (24, 32), ((8, 49),), 'AddBaseline',
None, FileName)
```

The first thing you should notice is that the MakePy utility makes a class object for each automation object in the Type library. The one you're currently looking at is the `_Presentation` object, inside this class object are all the methods and properties belonging to the said object.

For example, the `AcceptAll` method accepts all the changes in the presentation. Notice, how it is merely calling `InvokeTypes` method described up above? This is why we needed to cover the IDispatch interface! It's because it plays a crucial role in the development of the Python generated code. This way, when you look at the code, you won't be confused by all the abstract numbers.

The same goes for `_ApplyTypes_` , it's directly calling the `InvokeTypes` method behind the scenes to either set or retrieve the information at hand.

The other exciting part about the autogenerated code is the object properties, and if you scroll down further, you'll see, in some cases, a very comprehensive dictionary containing the all the `get_properties` and the `put_properties` . Here is how they look:

In [ ]:
```
# here is a few of the Get Properties
_prop_map_get_ = {
    # Method 'Application' returns object of type 'Application'
    "Application": (2001, 2, (13, 0), (), "Application", '{91493441-5A91-11CF-
8700-00AA0060263B}'),
    "AutoSaveOn": (2129, 2, (11, 0), (), "AutoSaveOn", None)
}

# here are a few of the Put properties
_prop_map_put_ = {
    "AutoSaveOn": ((2129, LCID, 4, 0),()),
    "ChartDataPointTrack": ((2125, LCID, 4, 0),())
}

# You can access these attributes a few different ways
AppProp1 = PPTApp.Presentations.Application
AppProp2 = PPTApp.Presentations.__getattr__('Application')
```

Let's take some time and walk through two of the items, one from each dictionary. If you notice each key represents the property of the object, and the value associated with that key is simply a tuple of information.

The first item, for the `_prop_map_get_` is the property id, the second number determines whether it is an INVOKE_PROPERTYGET (2) or INVOKE_PROPERTYPUT(4), the third is the return type description discussed up above, the fifth is the property name, and the final one specifies it doesn't return another object.

The first item, for the `_prop_map_put_` is the property id, the second number is the local context id discussed up above, the third represents that it is an INVOKE_PROPERTYPUT(4), and the fourth, from my best guess, represents its `wFlags` .

```
(2129, 2, (11, 0), (), "AutoSaveOn", None)
```

`2129` - The MemId for AutoSaveOn

`2` - pythoncom.INVOKE_PROPERTYGET

`(11, 0)` - 11 returns a boolean and 0 pythoncom.PARAMFLAG_NONE

`"AutoSaveOn"` - Property Name

`None` - This property does not return an object

```
"AutoSaveOn": ((2129, LCID, 4, 0),())
```

`2129` - The MemId for AutoSaveOn

`LCID` - Local Context ID, define at the beginning of the file

`4` - pythoncom.INVOKE_PROPERTYPUT

`0` - pythoncom.PARAMFLAG_NONE

At this point, you should feel somewhat comfortable exploring a MakePy generate Python file. You understand that the object is represented as a Python class object, the methods belonging to that the class object translates into the methods of the COM Object.The properties exist in a dictionary that can be accessed by the `__getattr__()` or calling the property directly on the class object, and finally the methods leverage the `InvokeTypes()` to call the methods and properties.

Regardless of whether you want to use late or early bound automation, you can force either one these by using `Dynamic` module, for late bound automation, and `gencahce` module, for early bound automation. The gencache runs the MakePy support that allows for early bound automation, and then the gencache calls the `EnsureDispatch` method to dispatch the object. Here is an example of forcing early and late bound automation.

If you choose to use the regular `Dispatch()` method without specifying either the `dynamic` or `gencache` module, then the `Win32COM` library will choose the best option. Most of the times, it will use early-bound automation if the `gencache` module has been called on the object before.

In [9]:
```
import win32com.client
# late bound automation
PPTApp_Late = win32com.client.dynamic.Dispatch('PowerPoint.Application')
display('LATE BOUND')
display(PPTApp_Late)

# early bound automation
PPTApp_Early = win32com.client.gencache.EnsureDispatch('PowerPoint.Applicatio
n')
display('EARLY BOUND')
display(PPTApp_Early)

# let win32COM choose
PPTApp_Choose = win32com.client.Dispatch('PowerPoint.Application')
display('CHOOSE FOR US')
display(PPTApp_Choose)
```

```
'LATE BOUND'

<COMObject PowerPoint.Application>

'EARLY BOUND'

<win32com.gen_py.Microsoft PowerPoint 16.0 Object Library._Application instan
ce at 0x1981722340544>

'CHOOSE FOR US'

<win32com.gen_py.Microsoft PowerPoint 16.0 Object Library._Application instan
ce at 0x1981719975808>
```