wmi Tutorial

Introduction

From the Wikipedia entry for WMI:

> Windows Management Instrumentation(WMI) is a set of extensions to the Windows Driver Model that provides an operating system interface through which instrumented components can provide information and notification. WMI is Microsoft's implementation of the Web-Based Enterprise Management(WBEM) Standard from the Distributed Management Task Force(DMTF). WMI allows scripting languages like VBScript to manage Microsoft Windows personal computers and servers, both locally and remotely. WMI is preinstalled in Windows Vista, Windows Server 2003, Windows XP, Windows Me, and Windows 2000.

This tutorial covers accessing WMI specifically using the Python programming language and assumes you have downloaded and installed Python itself, the pywin32 extensions and the WMI Python module. Python is able to use WMI by means of COM-enabling packages such as Mark Hammond's pywin32 extensions or the comtypes spinoff from Thomas Heller's ctypes. The WMI module assumes that the pywin32 extensions are present, and exposes a slightly more Python-friendly interface to the sometimes messy WMI scripting API. NB it does nothing but wrap pywin32 functionality: if WMI under pywin32 is too slow, this module won't be any faster.

The Basics

We're not really going to be looking at what you can do with WMI in general, except by way of demonstrating some functionality within the module. There are no few examples around the web of what you can do with the technology (and you can do most things with it if you try hard enough). Some links are at the bottom of the document.

Connecting

Most of the time, you'll simply connect to the local machine with the defaults:

```
import wmi
c = wmi.WMI()
```

If you need to connect to a different machine, specify its name as the first parameter:

```
import wmi
c = wmi.WMI("other_machine")
```

Querying

The most common thing you'll be doing with WMI is finding out about various parts of your system. That involves determining which WMI class to interrogate and then treating that as an attribute of the Python WMI object:

```
import wmi
c = wmi.WMI()
for os in c.Win32_OperatingSystem():
  print os.Caption
```

The "which WMI class to interrogate" part of that earlier sentence is not always so easy as it sounds. However, with a good search engine at your disposal, you can be pretty much assured that someone somewhere has done the same thing as you albeit with a different language. There are some helpful links at the bottom of this document, but I often simply stick "WMI thing-to-do" into my search engine of choice and scan the results for convincing answers.

Note that, although there is only, in this case, one Operating System, a WMI query always returns a list, possibly of one item. The items in the list are wrapped by the Python module for attribute access. In this case, the Win32_OperatingSystem class has several attributes, one of which is .Caption, which refers to the name of the installed OS.

Monitoring

WMI has the concept of events. There are two types, intrinsic and extrinsic, which are discussed below. The Python module makes the difference as transparent as it can. Say you wanted to track new processes starting up:

```
import wmi
c = wmi.WMI()
process_watcher = c.Win32_Process.watch_for("creation")
while True:
  new_process = process_watcher()
  print new_process.Caption
```

Note that you must pass one of "creation", "deletion", "modification" or "operation" to the .watch_for method. If not, slightly odd errors will result.

New in version 1.3.1: If you don't specify a notification type to an intrinsic event then "operation" will be assumed, which triggers on any change to an object of that class.

Updating

Some, but by no means all, WMI classes allow for the possibility of writing information back. This you do by setting attributes as usual in Python. Note that the class may let you do this without complaint even though nothing useful has happened. To change the display name of a service, for example, you need to call the service's .Change method. You can update the displayName attribute directly without error, but it will have no effect.

The most typical place in which you'll set an attribute directly is when you're creating a whole new object from a class's .new method. In that case you can either set all the parameters as keyword arguments to the .new call or you can specify them one by one afterwards.

Methods

Some WMI classes have methods to operate on them. You can call them as though they were normal class or instance methods in Python. From version 1.3.1, parameters can be positioned; before that, they must be named. If you wanted to stop the (running) RunAs service, whose short name is "seclogon":

```
import wmi
c = wmi.WMI()
for service in c.Win32_Service(Name="seclogon"):
  result, = service.StopService()
  if result == 0:
    print "Service", service.Name, "stopped"
  else:
    print "Some problem"
  break
else:
  print "Service not found"
```

Advanced Stuff

The basics of what can be done with the WMI module is covered above and this is probably as far as many people need to go. However, there are many slight

subtleties to WMI and you may find yourself studying a VBS-oriented example somewhere on the web and thinking “How do I do this in Python?”.

Advanced Connecting

The .connect function(aliased as .WMI) has quite a few parameters, most of which are optional and can safely be ignored. For the majority of them, I would refer you to the MS documentation on WMI monikers into which they slot fairly straightforwardly. We will introduce here a few of the more common requirements.

Connecting to a remote machine

This is the most common and the most straightforward extra parameter. It is the first positional parameter or the one named “computer”. You can connect to your own computer this way by specifying nothing, a blank string, a dot or any of the computer’s DNS names, including localhost. But usually you just don’t need to pass the parameter at all. To connect to the WMI subsystem on a computer named “MachineB”:

```
import wmi
c = wmi.WMI("MachineB")
```

Connecting to a remote machine as a named user

This is the second most common need and is fairly straightforward, but with a few caveats. The first is that, no matter how hard you try to obfuscate, you can’t connect to your local computer this way. The second is this technique doesn’t always play well with the many layers of WMI security. More on that below in troubleshooting. To connect to a machine called “MachineB” with username “fred” and password “secret”:

```
import wmi
c = wmi.WMI("MachineB", user=r"MachineB\fred", password="secret")
```

Connecting to a particular namespace

WMI classes are organised into a namespace hierarchy. The majority of the useful ones are under the cimv2 namespace, which is the default. But add-on providers may supply extra namespaces, for example MicrosoftIISv2 or DEFAULT/StdRegProv. To use a different namespace from the default(which is, incidentally, not the one named default!) specify it via the namespace parameter. All namespaces are assumed to start from root so it need not be specified, although if you want to specify the root namespace itself, you can do:

```
import wmi
c = wmi.WMI(namespace="WMI")
```

Specifying the full moniker

In some cases you want to be able to pass the full moniker along, either because the moniker itself is so complex, or because you want to be able to cut-and-paste from elsewhere. In that case, pass the moniker as a string via the "moniker" parameter:

```
import wmi
c = wmi.WMI(
  moniker="winmgmts:{impersonationLevel=impersonate,(LockMemory,
!IncreaseQuota)}"
)
```

Connecting to a specific class or object

A special case of the full moniker is that it can be used to connect directly to a WMI class or even a specific object. The Python module will notice that the moniker refers to a class or object and will return the wrapped object directly rather than a namespace. Any WMI object's path can be used as a moniker to recreate it, so to attach directly to the Win32_LogicalDisk class, for example:

```
import wmi
logical_disk = wmi.WMI(moniker="//./root/cimv2:Win32_LogicalDisk")
```

This is equivalent to getting hold of the class through the normal mechanism although it's mostly of use internally to the module and when translating examples which use the technique. Access to a specific object is similar and slightly more useful:

```
import wmi
c_drive = wmi.WMI(moniker='//./root/cimv2:Win32_LogicalDisk.DeviceID="C:"')
```

This object is the same as you'd have received by querying against the Win32_LogicalDisk in the cimv2 namespace with a parameter of DeviceID="C:" so from the point of view of the Python module is not so very useful. However it is a fairly common usage in VBS examples on the web and eases translation a little.

Advanced Querying

Filtering the returned list

We've already seen this in action above; I just didn't comment on it at the time. When you "call" a WMI class, you can pass along simple equal-to parameters to narrow down the list. This filtering is happening at the WMI level; you can still do whatever post-hoc filtering you want in Python once you've got the values back. Note that, even if the resulting list is only one element long, it is still a list. To find all fixed disks:

```
import wmi
c = wmi.WMI()
for disk in c.Win32_LogicalDisk(DriveType=3):
  print disk
```

Selecting only certain fields

By default, all fields in the class will be returned. For reasons of performance or simply manageability, you may want to specify that only certain fields be returned by the query. This is done by setting the first positional parameter to a list of field names. Note that the key field (typically an id or a unique name or even a combination) will always be returned:

```
import wmi
c = wmi.WMI()
for disk in c.Win32_LogicalDisk(["Caption", "Description"], DriveType=3):
  print disk
```

Performing arbitrary WQL queries

If you want to carry out arbitrary WMI queries, using its pseudo-SQL language WQL, you can use the .query method of the namespace. To list all non-fixed disks, for example:

```
import wmi
c = wmi.WMI()
wql = "SELECT Caption, Description FROM Win32_LogicalDisk WHERE DriveType <>
3"
for disk in c.query(wql):
  print disk
```

Advanced Monitoring

Intrinsic events

Intrinsic events occur when you hook into a general event mechanism offered by the WMI system to poll other classes on your behalf. You can track the creation, modification or deletion of any WMI class. You have to specify the type of event (creation, deletion, modification or simply operation to catch any type) and give a polling frequency in whole seconds. After those parameters, you can pass along keyword parameters in the normal way to narrow down the events returned. Note

that, since this is polling behind the scenes, you do not want to use this to, say, monitor an entire directory structure.

To watch an event log for errors, say:

```
import wmi
c = wmi.WMI(privileges=["Security"])
watcher = c.Win32_NTLogEvent.watch_for("creation", 2, Type="error")
while 1:
  error = watcher()
  print "Error in %s log: %s" % (error.Logfile, error.Message)
  # send mail to sysadmin etc.
```

A caveat here: this is polling, and at the frequency you've specified. It is possible to miss events this way.

The return from a watcher is in fact a special **`_wmi_event object`**, subclass of a conventional **`_wmi_object`**, and which includes, for intrinsic events, the event type, timestamp and previous value for a modification as attributes: **`_wmi_event.event_type`**, **`_wmi_event.timestamp`** and **`_wmi_event.previous`** respectively.

Extrinsic events

Note that, while "Win32_NTLogEvent" ends in "Event", it is not in fact an extrinsic event. You can tell which classes are extrinsic events by examining their derivation and looking for `__ExtrinsicEvent`:

```
import wmi
c = wmi.WMI()
print c.Win32_PowerManagementEvent.derivation()
```

Alternatively, you can go top down and look for subclasses of __ExtrinsicEvent:

```
import wmi
c = wmi.WMI()
for i in c.subclasses_of("__ExtrinsicEvent"):
  print i
```

You use extrinsic events in much the same way as intrinsic ones. The difference is that any event type and delay are ignored since WMI isn't polling on your behalf, but waiting on the underlying subsystem. The return from the watcher is still a _wmi_event object (1.3.1) but without the extra information, which isn't supplied by WMI. Suppose you wanted to do something whenever your computer came out of standby, eg to notify an IM group of your presence:

```
import wmi
import datetime
c = wmi.WMI()
watcher = c.Win32_PowerManagementEvent.watch_for(EventType=7)
while True:
  event = watcher()
  print "resumed"
  #
  # Number of 100-ns intervals since 1st Jan 1601!
  # TIME_CREATED doesn't seem to be provided on Win2K
  #
  if hasattr(event, "TIME_CREATED"):
    ns100 = int(event.TIME_CREATED)
    offset = datetime.timedelta(microseconds=ns100 / 10)
    base = datetime.datetime(1601, 1, 1)
    print "Resumed at", base  + offset
```

For an intrinsic modification event, you could compare the before and after values of the trigger instance:

```
import wmi
c = wmi.WMI()
watcher = c.Win32_Process.watch_for("modification")
event = watcher()
print "Modification occurred at", event.timestamp

print event.path()
prev = event.previous
curr = event
for p in prev.properties:
  pprev = getattr(prev, p)
  pcurr = getattr(curr, p)
```

```
  if pprev != pcurr:
    print p
    print "  Previous:", pprev
    print "   Current:", pcurr
```

Watchers with timeouts

But there's more! Although you can use these watchers inside threads (of which more below) it might be easier in some cases to poll them with a timeout. If, for example, you wanted to monitor event log entries on two boxes without getting into threading and queues:

```
import wmi

def wmi_connection(server, username, password):
  print "attempting connection with", server
  if username:
    return wmi.WMI(server, user=username, password=password)
  else:
    return wmi.WMI(server)

servers = [
  (".", "", ""),
  ("goyle", "wmiuser", "secret")
]
watchers = {}
for server, username, password in servers:
  connection = wmi_connection(server, username, password)
  watchers[server] = connection.Win32_PrintJob.watch_for("creation")

while True:
  for server, watcher in watchers.items():
    try:
      event = watcher(timeout_ms=10)
    except wmi.x_wmi_timed_out:
      pass
    else:
      print "print job added on", server
      print event
```

More About Methods

Determing available methods

If you examine the keys of the .methods dictionary which every wrapped WMI class uses to cache its wrapped methods, you will see what methods are exposed:

```
import wmi
c = wmi.WMI()
c.Win32_ComputerSystem.methods.keys()
```

Showing method signatures

Each wrapped method produces its function signature as its repr or str. If a function such as .Shutdown requires additional privileges, this is also indicated:

```
import wmi
c = wmi.WMI()
os = c.Win32_OperatingSystem
for method_name in os.methods:
  method = getattr(os, method_name)
  print method
```

Note that if a parameter is expected to be a list it will be suffixed with "[]". Note also that the return values are always a tuple, albeit of length one.

Finding a method's Win32 API equivalent

I was a bit surprised to come across this myself, but WMI tells you which Win32 API call is going on under the covers when you call a WMI method (not, unfortunately, for a property). This is exposed as a function wrapper's .provenance attribute:

```
import wmi
c = wmi.WMI()
print c.Win32_Process.Create.provenance
```

More Advanced Topics: Bits & Pieces

Creating WMI Objects

WMI exposes a SpawnInstance_ method which is wrapped as the `_wmi_object.new()` method of the Python WMI classes. But you'll use this method far less often than you think. If you want to create a new disk share, for example, rather than using `Win32_Share.new`, you'll actually call the `Win32_Share` class's `Create` method. In fact, most of the classes which allow instance creation via WMI offer a `Create` method (`Win32_Process`, `Win32_Share` etc.):

```
import wmi
c = wmi.WMI()
result, = c.Win32_Share.Create(Path="c:\\temp", Name="temp", Type=0)
if result == 0:
  print "Share created successfully"
else:
  raise RuntimeError, "Problem creating share: %d" % result
```

The times you will need to spawn a new instance are when you need to feed one WMI object with another created on the fly. Typical examples are passing security descriptors to new objects or process startup information to a new process. This example from MSDN can be translated into Python as follows:

```
import wmi

SW_SHOWNORMAL = 1

c = wmi.WMI()
process_startup = c.Win32_ProcessStartup.new()
process_startup.ShowWindow = SW_SHOWNORMAL
#
# could also be done:
# process_startup =
c.Win32_ProcessStartup.new(ShowWindow=win32con.SW_SHOWNORMAL)

process_id, result = c.Win32_Process.Create(
  CommandLine="notepad.exe",
  ProcessStartupInformation=process_startup
)
if result == 0:
  print "Process started successfully: %d" % process_id
else:
  raise RuntimeError, "Problem creating process: %d" % result
```

WMI Classes/Objects

Class/Object details

Each class and object will return a readable version of its structure when rendered as a string:

```
import wmi
c = wmi.WMI()
print c.Win32_OperatingSystem
```

```
for os in c.Win32_OperatingSystem():
  print os
```

The object hierarchy

WMI objects occur within a hierarchy of classes. Each object knows its own ancestors:

```
import wmi
c = wmi.WMI()
print c.Win32_Process.derivation()
```

You can also look down the tree by finding all the subclasses of a named class, optionally filtering via a regex. To find all extrinsic event classes other than the builtin ones (indicated by a leading underscore):

```
import wmi
c = wmi.WMI()
for extrinsic_event in c.subclasses_of("__ExtrinsicEvent", "[^_].*"):
  print extrinsic_event
  print "  ", " < ".join(getattr(c, extrinsic_event).derivation())
```

Comparing two WMI objects for equality

The `_wmi_object.__eq__()` operator is overridden in wrapped WMI classes and calls the underlying .CompareTo method, so comparing two WMI objects for equality should do The Right Thing.

Associators

Associators are classes which link together other classes. If, for example, you want to know what groups are on your system, and which users are in each group:

```
import wmi
```

```
c = wmi.WMI()

for group in c.Win32_Group():
  print group.Caption
  for user in group.associators(wmi_result_class="Win32_UserAccount"):
    print "  ", user.Caption
```

which can also be written in terms of the associator classes:

```
import wmi
c = wmi.WMI()

for group in c.Win32_Group():
  print group.Caption
  for user in group.associators("Win32_GroupUser"):
    print "  ", user.Caption
```

New in version 1.3.1: The **`_wmi_object.associators()`** method will convert its results to a _wmi_object.

Caveats, Troubleshooting and Performance

Speeding things up

Thanks to a useful collaboration last summer with Paul Tiemann, the module was able to speed things up considerably if needed with a combination of caching and lightweight calls where needed. Not all of that is covered here, but the most straightforward improvements combine removing runtime introspection and caching so that wrappers are generated only on demand and can be pre-cached.

Turning off introspection

The focus of the module originally, and still a large part of its use today, is in the interpreter. For that reason, when you instantiate a WMI namespace it looks for all the classes available in that namespace. But this takes quite a while on the larger

namespaces and is unnecessary even on the smaller ones once you know what you're after. In production code, therefore, you can turn this off:

```
import wmi
c = wmi.WMI(find_classes=False)
```

If you need to determine which classes are available, you can still use the subclasses_of functionality described above to search, for example, for the performances classes available on a given machine at runtime:

```
import wmi
c = wmi.WMI(find_classes=False)
perf_classes = c.subclasses_of("Win32_PerfRawData")
```

Note

From v1.4 onwards, the `find_classes` parameter is False by default: it has to be turned on specifically. But... the `classes` attribute now does a lazy lookup, so if you do call it directly or indirectly, eg by using IPython which invokes its attribute lookup magic method `_wmi_object._getAttributes()` it will return the full list of classes in the naeepsace.

Pre-cache class and method wrappers

To avoid an initial lookup hit when a class is first queried or its method first called, it's possible to push the class into the cache beforehand simply by referring to it. So, extending the code above:

```
import wmi
c = wmi.WMI(find_classes=False)
for perf_class in c.subclasses_of("Win32_PerfRawData"):
  # do nothing, just get it into the cache
  getattr(c, perf_class)
```

Specifying fields in the query

By default a WMI query will return all the fields of a class in each instance. By specifying the fields you're interested in up-front as the first parameter of the query, you'll avoid any expensive lookups. Although many fields represent static or cheap data, a few are calculated on the fly. This is especially true for performance or other realtime data in classes such as `Win32_Process`:

```
import wmi
c = wmi.WMI(find_classes=False)
for i in c.Win32_Process(["Caption", "ProcessID"]):
  print i
```

Security

This is going to be a small section at the moment, more of a heads-up until I have a few more firm facts at my disposal. In short, the simplest way by far to access WMI functionality is to run as a Domain Admin user on an NT/AD domain. Other techniques are certainly possible, but if they stall at any point, you're left ploughing through at least three layers of security, prodding hopefully at each one until you get a result or give up in disgust.

NT Security

The user in question has to have some kind of access to the machine whose WMI functionality is being invoked. This might either be by virtue of being included in the local Admin group or by specific access granted to a named user.

DCOM Security

WMI is a DCOM-based technology and so whatever rules apply to DCOM connections apply to WMI. If there's a problem authenticating at the DCOM level then, in theory, you ought to have the same problem doing a DispatchEx on Word.Application. The program you want to look at is dcomcnfg.exe and that's all I'll say for now.

WMI Security

WMI namespaces are system objects with their own ACLs. If you go to the WMI MMC snap-in (accessed via the Manage Computer interface) and access the properties for a namespace, there will be a security tab. The account using WMI functionality on the machine needs to have sufficient access via this security.

WMI and Threads

WMI is a COM/DCOM-based mechanism so the rules which apply to COM-threading apply to WMI as well. This is true whether or not your program explicitly invokes Python threading: if you're running in a service, for example, you're probably threading whether you like it or not, since the service control manager seems to run the service control code in a different thread from the main service.

CoInitialize & CoUninitialize

Any COM code which wants to use threading must specify a threading model. There is much said out there on the subject but unless you have specific requirements you can normally get away with initializing COM threading before you instantiate a WMI object within a thread and then uninitializing afterwards:

```
import pythoncom
import wmi
import threading
import time

class Info(threading.Thread):
  def __init__(self):
    threading.Thread.__init__(self)
  def run(self):
    print 'In Another Thread...'
    pythoncom.CoInitialize()
    try:
      c = wmi.WMI()
      for i in range(5):
        for process in c.Win32_Process():
          print process.ProcessId, process.Name
        time.sleep(2)
    finally:
      pythoncom.CoUninitialize()

if __name__ == '__main__':
  print 'In Main Thread'
  c = wmi.WMI()
  for process in c.Win32_Process():
    print process.ProcessId, process.Name
  Info().start()
```

New in version 1.4.1: From v1.4 onwards, the Moniker Syntax Error which usually results from failing to initialise threaded WMI access will be caught by the underlying code and a **`x_wmi_uninitialised_thread`** exception will be raised instead.