# Using the Jython Scripting Language with WSADMIN

**A tutorial on the use of the Jython Scripting Language focusing on its use with the WSADMIN commands**

Mike Loos
Consulting IT Specialist
WebSphere on z/OS
mikeloos@us.ibm.com

Thanks to the following folks:
Rohith Ashok who provided much help with Jython and wsadmin scripting.
Don Bagwell for help with format, content, and publishing.
Bill Keller for his technical editing.

Others who contributed, too many to name.

# **Jython Scripting Language**
used with the WSADMIN tool
**WP100963**

## ***Introduction to the Jython Scripting Language.***

The intent of this document is to introduce the reader to the use of Jython as a scripting language to be used with the wsadmin function of WebSphere Application Server.  WebSphere supports both the JACL language and the Jython language for scripting use, but since JACL has been deprecated with the V6.1 release of WebSphere, and the recommended direction is to use Jython, it seems appropriate to provide this short tutorial on the language.

The methodology used here will be to introduce in a simple and abbreviated manner the components of the Jython language most likely to be used in wsadmin scripts. That introduction will then be illustrated with a few examples demonstrating the capabilities and functioning of Jython.  Next, a few the wsadmin commands commonly used in scripts will be introduced, and finally the two (Jython and wsadmin) will be tied together with some example scripts performing simple, common, and hopefully useful functions.

## *A Simple Description of the Jython Language Components.*

There are several possible methods to describe a language. The one that we'll follow here will be to first describe the syntax and data types, then to move immediately into some short examples. If (when) a more complete understanding of the language is desired, a couple of good places to start are with the following links (also easily found by searching on **JYTHON** on the IBM developerworks site (http://www.ibm.com/developerworks/).

A quick overview entitled "Charming Jython":

http://www.ibm.com/developerworks/java/library/j-jython.html

An "Intro to Jython Part 1":

https://www.software.ibm.com/developerworks/education/j-jython1/

An "Intro to Jython Part 2":

https://www.software.ibm.com/developerworks/education/j-jython2/

### Jython Data Types

If you are a procedural language person, don't be intimidated by the fact that Jython sees everything as an object. Even though everything is seen as an object, the way it is used in Jython is very "procedurally" oriented. People experienced primarily in procedural languages will find themselves relatively comfortable in the way that Jython deals with data. All data and code is an object and can be manipulated as an object. Some types of data, such as numbers and strings, are easier to think of as values rather than objects. Jython supports this as well. Jython supports exactly one null value, which has a reserved name of "None".

### Operations (Operators)

Basic assignment is done via a single equals sign (=). That is, to assign the value of "3" to a variable called "x", a statement such as:

```
x = 3
```

would be used. This works as well for string type data.

```
x = “a string value”
```

would effectively assign the value “a string value” to the variable “x”.

## Operations (Comparison)

| | |
|---|---|
| `x < y` | `Is x less than y?` |
| `x > y` | `Is x greater than y?` |
| `x <= y` | `Is x less than or equal to y?` |
| `x >= y` | `Is x greater than or equal to y?` |
| `x == y` | `Is x equal to y?` |
| `x != y` | `Is x not equal to y?` |
| `x <> y` | `Is x not equal to y?` |

## Operations (Numeric)

| | |
|---|---|
| `x + y` | `Add y to x` |
| `x - y` | `Subtract y from x` |
| `x * y` | `Multiply x by y` |
| `x / y` | `Divide x by y` |
| `x ** y` | `Raise x to the y power` |

This is not an exhaustive list of the available operations in the language, but serves to illustrate those most likely to be used in a wsadmin script.

## Lists

Lists are sequences of elements of any type. They can be thought of as arrays if that is more convenient. Lists are particularly important to us since wsadmin makes such extensive use of them. The number of elements in a list can increase or decrease as elements are added, removed or replaced. A list can contain any number of elements, and the elements of the list can be any type of object. Some examples of lists are included below.

| | |
|---|---|
| `[]` | `An empty list` |

| | |
|---|---|
| `[1]` | `A list with a single element, an integer` |
| `[‘Mike’, 10, “Don”, 20]` | `A list with four elements, two strings and two integers` |
| `[[], [7], [8, 9]]` | `A list of lists;  Each sub list is either an empty list or a list of integers` |
| `x = 7; y = 2; z = 3;`<br>`[1, x, y, x + y]` | `This is a list of integers, demonstrating the use of variables and expressions, both of which are supported.` |

## Remarks

Remarks are preceded by the pound sign (or sharp) character (#).  All text following the # character on the same line is considered a remark.  A remark can start in any column.  An example would be:

```
import sys
#  The HelloWorld application is one of the the most simple.
print ‘Hello World’     #  print the Hello World line.
```

## Statement Syntax

Jython has a very simple statement syntax.  In general, each line contains a single statement.  With the exception of expression or assignment statements, each statement begins with a keyword, such as “if” or “for”.  Blank lines or remark lines can be inserted anywhere.  There is no need to end each line with a semicolon (;) but if you wish to include more than one statement per line, each statement must end with a semicolon.

Statements can be continued on subsequent lines by ending the continued statement with a backslash character (\).  If you are in the middle of a structure enclosed in parenthesis or brackets (“()” or “[]”) you may continue the line after any comma in the structure without the use of the backslash.

## Identifiers

Jython Identifiers are used to name variables, functions, classes, and keywords.  Identifiers can be any length, but must start with an alphabetic character (upper or lower case), or the underscore (_).  Other than the first character, identifiers can be composed of any combination of alphabetic characters, decimal numeric digits (0 – 9), and the underscore character.

## Code Blocks

Blocks of code are used where multiple statements are used when a single statement is expected. All statements that can handle a block of code as a target introduce the block with a colon (:) character. The statements: if, elif, else, for, while, try, except, def, and class can accept a block as their target.

Example:

```
if x == 1:
    y = 2
    z = 3
elif:
    y = 4
    z = 5
```

It is important to keep in mind that indentation has meaning in Jython. It is encouraged that you indent using the space character rather than the tab character, since different implementations on various platforms can interpret the tab character differently. It is important that code blocks be indented, and indented uniformly, since a change in the indentation will signal the end of the code block.

Before we start with some simple examples, it is worth noting that a Jython interpreter is available for the Windows and Linux environments. Simply go to the Jython Project website http://www.jython.org/Project/index.html and download and install the correct implementation following the instructions provided. This will provide you the ability to test some of these examples (that don't use wsadmin commands) in a "lighter weight" environment than running under the wsadmin.sh command. Of course you cannot include any of the wsadmin specific commands.

## *Some Simple Examples of the Language.*

The most common simple example is the HelloWorld application. In Jython this is simply:

```
print “Hello World”
```

Yields:

```
Hello World
```

The print keyword is one of the most commonly used Jython tools. It will print the argument(s) immediately following it. If you add a comma at the end of the statement, it will NOT include a newline character.

```
print “This demonstrates the use of the”,
print “ comma at the end of a print stmt.”
```

Yields:

```
This demonstrates the use of the comma at the end of a print stmt.
```

The “for” statement will commonly be used to iterate through a block of code. The following example assigns values to a list, and then prints the values in the list one per line.

```
mylist1 = [“one”, “two”, “three”]
for lv in mylist1:
    print lv
    continue
```

Yields:

```
one
two
three
```

Let’s add a simple conditional to the above snippet to show the use of the “if” statement.

```
mylist1 = [“one”, “two”, “three”]
for lv in mylist1:
    if lv == “two”:
        print “the value of lv is “, lv
    else:
        print “the value of lv is not two but, “, lv
    continue
```

Yields:

```
the value of lv is not two but, one
the value of lv is two
the value of lv is not two but, three
```

In the above two examples you also may have noticed the use of an “iterator” value. In the examples, “lv” was used. This is an arbitrary selection and any identifier can be used. Some languages force the use of a single character iterator identifier, but Jython allows the use of any valid identifier, of any length. The iterator” takes on the value of each element in the list of values as the for loop goes through the block of code for each element. If you wished to access the specific elements directly, you could also code for that. The elements are numbered starting with 0. From the above example list:

```
mylist1[0]              would be one
mylist1[1]              would be two
mylist1[2]              would be three
```

You will probably use this concept often when working with the wsadmin commands as many of the objects with which you will work in wsadmin will be in the form of lists or lists of lists.

## Passing arguments to a script

It is common to want to pass argument(s) to a script. It allows the script to be used more than once without modification. The arguments passed on the command line are passed as values in the list sys.argv. The number of values passed can be obtained by examining len(sys.argv). You'll note the inclusion of the "import" command. In this case we are simply including the entire sys class so that we can use the methods, such as argv that exist for the class. A short example follows:

```
import sys
print "test1"
print sys.argv[0]
print sys.argv[1]
print len(sys.argv)
```

If the invocation of the script were:

```
/u/mjloos/test1 mike don
```

The result would be:

```
test1
mike
don
2
```

This is one area where the use of Jython in a "standalone" environment will be different from when it is used as a scripting tool with wsadmin. The value of sys.arg[0] under wsadmin is the first parameter passed as an argument to the script. The value of sys.arg[0] in a standalone environment will be the pathname of the script itself. So, in a standalone environment the same test would result in:

```
/u/mjloos/test1 mike don
test1
mike
don
3
```

Also, you will notice that the first line of the script is an import statement for the sys package. This is necessary in a standalone environment, but will have already been accomplished in the wsadmin environment. Including it will cause no problems.

The language is much richer than the above examples demonstrate, but with those few constructs, it is possible to develop quite a few useful wsadmin scripts.

## *An Abbreviated Lesson on the WSADMIN Commands.*

The wsadmin commands can be divided up into four basic groups (actually objects): AdminApp, AdminConfig, AdminControl, AdminTask, and Help.

AdminApp is the set of commands involved with the installation, removal and editing of applications. The best way to find information on the available commands is to search on "AdminApp" in the InfoCenter and select the result named "Commands for the AdminApp object".

AdminConfig is the set of commands used to alter specific items within the configuration. There are commands within the AdminConfig object that allow you to create, remove, and modify attributes of objects within the configuration. The best way to find information on the available commands is to search on "AdminConfig" in the InfoCenter and select the result named "Commands for the AdminConfig object".

AdminControl is the set of commands used to control objects within the configuration. These are the commands used to start and stop servers, start and stop applications, test connections, start node synchronization, etc. The best way to find information on the available commands is to search on "AdminControl" in the InfoCenter and select the result named "Commands for the AdminControl object".

AdminTask is the set of commands used to perform specific tasks using the wsadmin interface. It would be used for such things as creating or deleting a server, managing the port configuration, running reports on the configuration, etc. Most everything that you would do with the AdminTask commands could also be done using combinations of the AdminApp, AdminConfig, and AdminControl commands, but someone in wsadmin development has packaged them up into useful commands and delivered them under the AdminTask object. The best way to find information on the available commands is to search on "AdminTask" in the InfoCenter and select the result named "Commands for the AdminTask object". At the bottom of that entry you would find links to the commands associated with the various groups of the AdminTask object (ClusterConfigCommands, PortManagement, AdminstrationReport, etc.).

There is help available at a "base" level:

```
print Help.help();
```

will give a list of the objects and statements for which help is available.

There is help available on each of the other objects mentioned, for instance the AdminTask object:

```
print AdminTask.help()
```

will provide a list of the available help options for the AdminTask object.  Drilling down a further level:

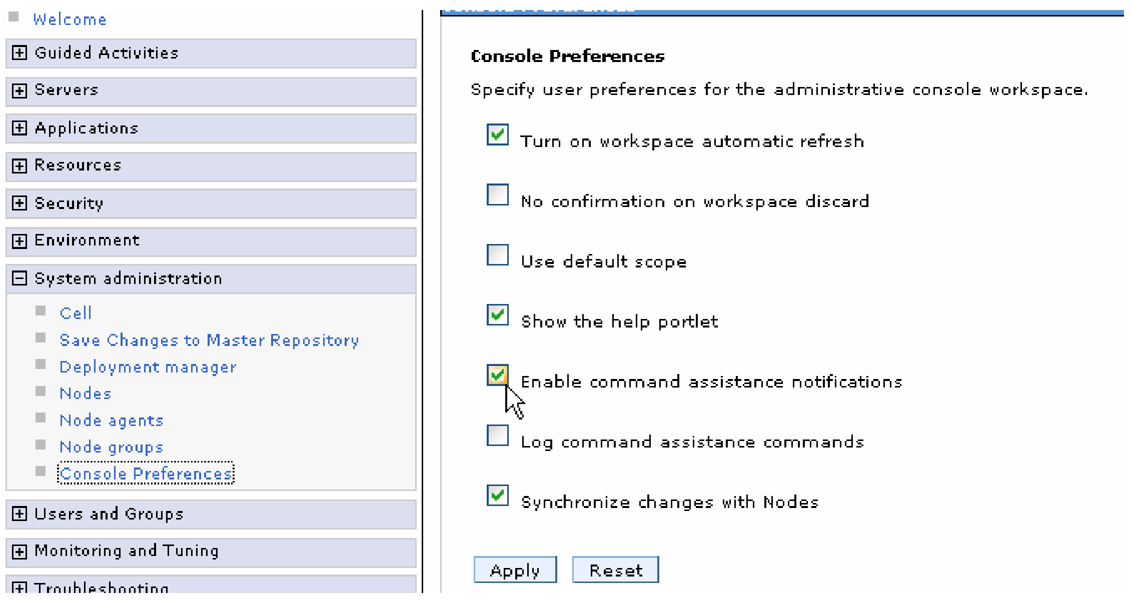
```
print AdminTask.help(“-commands”)
```

will print a list of all of the commands available under the AdminTask object.  Continuing to an even finer level:

```
print AdminTask.help("listServerPorts")
```

will provide help on the specific command, listServerPorts, of the AdminTask object.

Naturally, the most difficult part of writing a script is to know which of the hundreds of commands is the correct one for each task.  There is some help available in the WebSphere V6.1 adminconsole, using the command assistance notifications.  To enable this option, you set it in the adminconsole >> System administration >> Console preferences section.

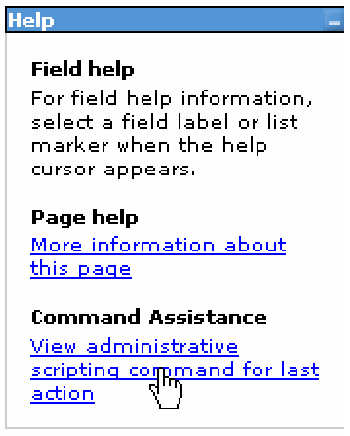


**Console Preferences**

Specify user preferences for the administrative console workspace.

- ✓ Turn on workspace automatic refresh
- ☐ No confirmation on workspace discard
- ☐ Use default scope
- ✓ Show the help portlet
- ✓ Enable command assistance notifications
- ☐ Log command assistance commands
- ✓ Synchronize changes with Nodes

Once it is set, after you have done something in the adminconsole that you would like to duplicate in a script, you would click on the command assistance link in the help box on the right hand side of the console.

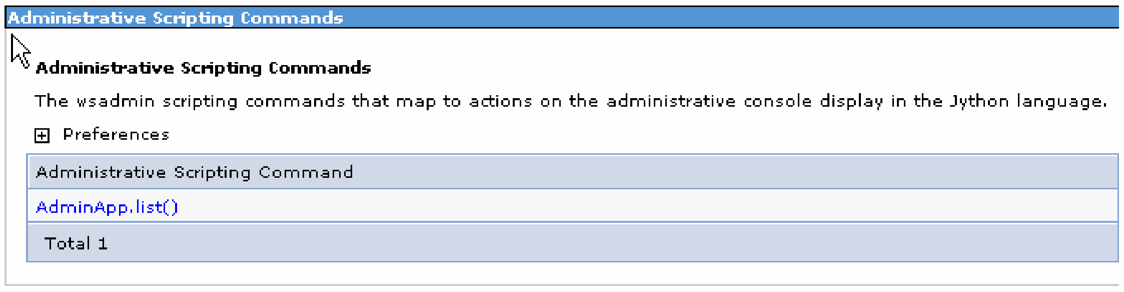

Help

**Field help**
For field help information, select a field label or list marker when the help cursor appears.

**Page help**
More information about this page

**Command Assistance**
View administrative scripting command for last action

That will cause a popup box to be displayed that will have the command in it.


Administrative Scripting Commands

**Administrative Scripting Commands**

The wsadmin scripting commands that map to actions on the administrative console display in the Jython language.

⊞ Preferences

| Administrative Scripting Command |
|---|
| AdminApp.list() |
| Total 1 |

You also certainly have the option of reading about each command in the InfoCenter. I actually would suggest that you print a copy of each of the InfoCenter articles mentioned at the beginning of this section, entitled “Commands for the AdminApp object” or whichever object you are interested in at the time.

The following link points to a page in the InfoCenter that contains not only a very good description of wsadmin scripting, but also has a list of links to more detail on many of the subjects that we will be illustrating in the next section.

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/welcadminscript.html

Also, the techdocs website, at: http://www.ibm.com/support/techdocs when searched using an argument of "wsadmin", yields several documents that can serve as a primer on the wsadmin scripting tool. In particular WP100421 - WebSphere Version 5 for z/OS - WSADMIN Primer is a very good introduction. It uses JACL, but the translation to Jython is not particularly difficult.

## *Examples of Jython Scripts used with WSADMIN Commands.*

Let's take a look at a few examples that might prove useful.

How about simply getting a list of all of the servers in a node? All you really need to know is the name of the node for which you wish a list. We can pass that in as an argument.

```
import sys
#
#
n1 = sys.argv[0]
#
#
plist = "[-serverType APPLICATION_SERVER -nodeName " + n1 + "]"
print plist
#
slist = AdminTask.listServers(plist).split("\n")
#
for s in slist:
  print s
  continue
```

The first non-remark line, "`import sys`" is not really necessary, but allows us to do some minor testing in a standalone (non wsadmin) environment if we wish, and won't cause any trouble in a wsadmin environment. Its purpose is to bring in the methods necessary to process the arguments to the command.

The command to invoke wsadmin to run this script might look like this (all on a single line):

```
/wasv61config/s1cell/s1dmnode/DeploymentManager/profiles/default/bin/wsadmin.sh
–lang jython –javaoptions –Dscript.encoding=IBM-1047 –f
/u/mjloos/scripts/listservs.py s1nodec
```

The second line (`n1 = sys.argv[0]`) assigns the first command argument, hopefully the value for node, to the variable n1.

Next we set up the parameter list for the `AdminTask.listServers` command. The parameter list is of the form

```
[-serverType APPLICATION_SERVER –nodeNAME nodename]
```
We are really concatenating three things together (a constant, a variable, and a constant) with the "+" operator to form the variable named plist.

We then print the parameter list with the statement:
```
print plist
```
Result: `[-serverType APPLICATION_SERVER -nodeName s1nodec]`

Printing the parameter list is not necessary, but it can be very valuable when initially debugging the script, and to help determine what might have gone wrong at some time in the future.

The next line: `slist = AdminTask.listServers(plist).split("\n")` assigns the list of servers (splitting each element in the list on the newline character with the split("\n") function to the variable slist. The split("\n") function splits the returned value, so that we are presented with a new element in the list every time a newline character is encountered. Most of the values returned by .list type commands include a newline character to delineate the elements in the list.

Next we use a loop to print each element in the list.

```
for s in slist:
  print s
  continue
```

Result:
```
s1sr09t(cells/s1cell/nodes/s1nodec/servers/s1sr09t|server.xml)
s1sr01c(cells/s1cell/nodes/s1nodec/servers/s1sr01c|server.xml)
```

And that completes the list of servers example.

The results look a little unwieldy, but that is what WebSphere uses to keep track of the "parts" of its configuration. Sometimes the values returned are referred to as "complete object names" and sometimes the "object id" of the element.

How about a list of all applications in a cell? That one is surprisingly easy.

```
applist = AdminApp.list().split("\n")
for a in applist:
    print a
    continue
```

The first line applist = AdminApp.list() stores a list of all applications in the cell into a list variable, applist.

The next few lines simply print each element in the list in turn.

Let’s try something a little more useful and common. After almost any change it will be necessary to synchronize with the nodes. The following fragment will cause a synchronization to happen with each node in the cell and can be included as part of many other scripts. Naturally, this would be done after the configuration has been “saved”, using the AdminConfig.save() command, and that will be illustrated in a later example.

```
nodelist = AdminConfig.list('Node').split("\n")
for n in nodelist:
    nodename = AdminConfig.showAttribute(n, 'name')
    objname = "type=NodeSync,node=" + nodename + ",*"
    Syncl = AdminControl.completeObjectName(objname)
    if Syncl != "":
        AdminControl.invoke(Syncl, 'sync')
        print "Done with node " + nodename
    else:
        print "Skipping node " + nodename
    continue
```

The first line stores a list of all nodes in a cell into the variable nodelist. The for loop then starts by storing the name attribute of each element in the list in a variable called nodename. It next creates the parameter list by concatenating a constant string with the variable nodename and a final constant string and stores it in a variable called objname. Next the AdminControl.completeObjectName method is used with the parameter list stored in objname to store the complete object name in variable Syncl. If this is done for a deployment manager node, the complete object name has no value, so we check to see if it is an empty value and if it is not we invoke the synchronization service for the node, otherwise we print a message that we are skipping that node. Then we simply loop through all elements (nodes) in the list.

Okay, now that we know we can list things, and we know how to synchronize the dmgr with the nodes, let’s try something vaguely useful. One reasonably common thing to change in a server is the JVM heap sizes. The following script will accomplish exactly that for a particular server. Note that the formatting imposed by the word processor has caused some of the lines in the script to split to a second line. Since this script is a little longer than those previously shown, I’ll include the description inline in a different *font and color.*

```
import sys
```

```
#
# ./wsadmin.sh -lang jython -javaoption -Dscript.encoding=IBM-1047\
# -user userid -password password -f /scriptingpath/heapadjust.py
# servername initheap maxheap
#
```
*The following few lines demonstrate the use of a Jython function. This function, printHelp, is used when the number of arguments passed to the script isn't sufficient to allow the script to run properly, and simply prints some documentation on proper usage.*
```
def printHelp():
    print "This script sets the jvm init and max heap sizes for a
server"
    print "Format is heapadjust.py servername initheapsize
maxheapsize"
#
```
*Check to see that at least three arguments have been passed, and set the proper variables from the arguments, s1 for server name, ih for initial heapsize, and mh for maximum heapsize. If the number of variables is less than three then invoke the printHelp function and the sys.exit method to terminate the script.*
```
if(len(sys.argv) > 2):
    # get server name, initialheapsize, and maxheapsize from the
command line
    s1 = sys.argv[0]
    ih = sys.argv[1]
    mh = sys.argv[2]
else:
    printHelp()
    sys.exit()
```
*Set the ihc and mhc variables to the strings necessary for the AdminConfig.modify command to be used later.*
```
ihc = 'initialHeapSize'
mhc = 'maximumHeapSize'
#
#   get the id of the server
#
```

*Set up the string necessary to find the server with the getid method. The form is /Server:servername/.*
```
srv = "/Server:" + s1 + "/"
```
*The following lines first get the id of the server, and then find all of the process definitions for the server. It then loops through the list of process definitions until it finds the process definition for the servant. This is done by comparing the attribute called processType, which is one of the attributes of a process definition, for a value of Servant. Once we have a true comparison, we use that process definition to get the JavaVirtualMachine object for that process definition.*

*Note: The ProcessDef object only exists in the z/OS environment where the controller / servant server structure exists. Other environments could go directly to the JavaVirtualMachine.*

```
srvid = AdminConfig.getid(srv)
s1pdefs = AdminConfig.list('ProcessDef', srvid).split("\n")
for pd in s1pdefs:
    if AdminConfig.showAttribute(pd, 'processType') == 'Servant':
        jvms = AdminConfig.list('JavaVirtualMachine',
pd).split("\n")
    continue

#  we now have the correct jvm
```

*We can now modify the values for initial and maximum heapsize in the servant jvm.*

```
for jvm in jvms:
    AdminConfig.modify(jvm, [[ihc, ih]])
    AdminConfig.modify(jvm, [[mhc, mh]])
    continue

#
#   Save the Config
#
```

*This command saves the configuration change. We then use the code we previously explained to synchronize the changes with the nodes.*

```
AdminConfig.save()
#
#
#   Synchronize with the nodes...
#
nl = AdminConfig.list('Node').split("\n")
#
#
print "Beginning node synchronization"
for n in nl:
    nn = AdminConfig.showAttribute(n, 'name')
    objn = "type=NodeSync,node=" + nn + ",*"
    Syncl = AdminControl.completeObjectName(objn)
    if Syncl != "":
        AdminControl.invoke(Syncl, 'sync')
        print "Done with node " + nn
    else:
        print "Skipping node " + nn
    continue
#  All Done.
```

In the previous script, we used a method on the AdminConfig object, showAttribute that we really hadn't discussed much. The following snippets of code might help to demonstrate how to use showAttribute, and how to get a list of attributes with the show method. Let's assume that we have already gotten the server, and retrieved the list of process definitions in a variable s1pdefs.

```
for sx in s1pdefs:
    print sx
    continue
```

```
(cells/s1cell/nodes/s1nodec/servers/s1sr01c|server.xml#JavaProcessDe
f_1154632346642)
(cells/s1cell/nodes/s1nodec/servers/s1sr01c|server.xml#JavaProcessDe
f_1154632346643)
(cells/s1cell/nodes/s1nodec/servers/s1sr01c|server.xml#JavaProcessDe
f_1154632346644)
```

So now we've printed the object ids for the three process definitions that exist for server s1sr01c. Again, we will loop through the process definitions until we find the one for the servant. Then we set the variable s1jvm to the JavaVirtualMachine of the servant process definition. The next line prints the object id of the jvm. Then a blank line; Then a list of all of the attributes on the jvm; another blank line; And finally the value of a particular attribute, the verboseModeGarbageCollection attribute.

```
for sx in s1pdefs:
    if AdminConfig.showAttribute(sx, 'processType') == 'Servant':
        s1jvm = AdminConfig.list('JavaVirtualMachine', sx)
        print s1jvm
        print " "
        print AdminConfig.show(s1jvm)
        print " "
        print AdminConfig.showAttribute(s1jvm,
'verboseModeGarbageCollection')
        continue
```

*The following is the result of the print s1jvm line.*

```
(cells/s1cell/nodes/s1nodec/servers/s1sr01c|server.xml#JavaVirtualMa
chine_1154632346643)
```

*The following is the result of the print AdminConfig.show(s1jvm) line.*

```
[bootClasspath []]
[classpath []]
[debugArgs "-Djava.compiler=NONE -Xdebug -Xnoagent -
Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777"]
[debugMode false]
[disableJIT false]
```

```
[genericJvmArguments []]
[hprofArguments []]
[initialHeapSize 256]
[internalClassAccessMode ALLOW]
[maximumHeapSize 512]
[runHProf false]
[systemProperties []]
[verboseModeClass false]
[verboseModeGarbageCollection false]
[verboseModeJNI false]
```

*And finally, the following line is the result of the print AdminConfig.showAttribute(s1jvm, ‘verboseModeGarbageCollection’) line.*

```
False
```

Some values are stored as a string containing a list of values. One example would be the list of generic JVM arguments that are passed to the JVM at startup time. The following example will demonstrate how to add an argument to the list of arguments. It also serves to demonstrate making the same modification to all elements in a cell, in this case every jvm in the cell. In particular this script adds the jvm argument –Xifa:force to the generic jvm arguments string for every jvm in the cell. This might be done so as to allow the collection of information in the RMF type 72 record on the use of a zAAP engine, prior to the actual installation of a zAAP engine.

```
import sys
# ./wsadmin.sh -lang jython -javaoption -Dscript.encoding=IBM-1047\
# -user userid -password password -f /scriptingpath/xifaforce.py
#
#   get a list of all of the jvms in the cell
#
```

*The following method will get a list of all JVM objects in the cell.*

```
jvmlist = AdminConfig.list('JavaVirtualMachine').split("\n")
#
ifaconst = '-Xifa:force'
for jvm in jvmlist:
    print
    print "Modifying JVM ",
    print jvm
    print
```

*The following few lines will obtain the genericJvmArgument attribute for the specific jvm, check to see if it is null and if it is, set it to a “valid” null value, print it as is, and add a space and the value of ifaconst (set earlier to ‘-Xifa:force’) to the attribute. Next we get rid of extraneous spaces with the strip function, print the new value, and modify the value of the attribute in the jvm object.*

```
    arg1 = 'genericJvmArguments'
    arg2 = AdminConfig.showAttribute(jvm, arg1)
    if not arg2:
        arg2 = ""
    print 'Old generic JVM args = ',
    print arg2
    arg2 = arg2 + " " + ifaconst
    arg2 = arg2.strip()
    print "New generic JVM args = ",
    print arg2
    print
    rc = AdminConfig.modify(jvm, [[arg1, arg2]])
    continue
#
#
#   Save the Config
#
```

*Now we simply save and synch as we have shown previously.*

```
AdminConfig.save()
#
#
#   Synchronize with the nodes...
print "Synchronizing Nodes"
print
#
nl = AdminConfig.list('Node').split("\n")
#
#
for n in nl:
    nn = AdminConfig.showAttribute(n, 'name')
    objn = "type=NodeSync,node=" + nn + ",*"
    Syncl = AdminControl.completeObjectName(objn)
    if Syncl != "":
        AdminControl.invoke(Syncl, 'sync')
        print "Done with node " + nn
    else:
        print "Skipping node " + nn
    continue
#  All Done.
```

The next script simply undoes what the previous one did.

```
import sys
#
# ./wsadmin.sh -lang jython -javaoption -Dscript.encoding=IBM-1047\
# -user userid -password password -f /scriptingpath/remxifaforce.py
#
#   get a list of all of the jvms in the cell
#
```

```
jvmlist = AdminConfig.list('JavaVirtualMachine').split("\n")
#
ifaconst = '-Xifa:force'
for jvm in jvmlist:
    print
    print "Modifying JVM ",
    print jvm
    print
    argm = ""
    arg1 = 'genericJvmArguments'
```

*In the case where we want to remove an argument, we need to have the arguments in a list. We know from examination that a space is used as a delimiter for the arguments, so we set the arg2 variable to a list of the arguments, delimited by space.*

```
    arg2 = AdminConfig.showAttribute(jvm, arg1).split(" ")
    print 'Old generic JVM args = ',
    print arg2
```

*Now we can loop through the particular arguments searching for the one in which we are interested, and cause the elimination of the particular argument from the new argument string. Everything else is the same as the previous script: modify, save, synch…*

```
    for a in arg2:
        if a != ifaconst:
          argm = argm + " " + a
        continue
    argm = argm.strip()
    print 'New generic JVM args = ',
    print argm
    rc = AdminConfig.modify(jvm, [[arg1, argm]])
    continue
#
#
#   Save the Config
#
AdminConfig.save()
#
#
#   Synchronize with the nodes...
print "Synchronizing Nodes"
print
#
nl = AdminConfig.list('Node').split("\n")
#
#
for n in nl:
    nn = AdminConfig.showAttribute(n, 'name')
    objn = "type=NodeSync,node=" + nn + ",*"
    Syncl = AdminControl.completeObjectName(objn)
    if Syncl != "":
        AdminControl.invoke(Syncl, 'sync')
        print "Done with node " + nn
    else:
        print "Skipping node " + nn
    continue
#  All Done.
```

We haven't done anything with the AdminApp object, other than our simple list example at the beginning of the section, nor have we used the AdminControl object. This next script uses them both to stop, uninstall, install, and start the PolicyIVP application with which many are already familiar.

We start with some documentation on how to invoke the script. For simplicity the application name is a literal in this program. You would probably normally use a script argument to set the application name, but on the other hand, most applications tend to install quite differently from each other. If your intent is to write a "generic" script to handle the installation of many different applications, you will quickly need a great number of arguments.

```
#
#  Sample invocation:
#
#  ./wsadmin.sh -lang jython -javaoption -Dscript.encoding=IBM-1047 -f
/u/mjloos/tb/pivpinst.py
#
#
#  Uninstall the app if it is there
#
```

*Here we get the list of installed applications and loop through it searching for the one in which we are interested. Once we have found it, we print a message, use the queryNames command of the AdminControl object to get the id of the Mbean which represents the running application in the proper server. We can then use that to use the invoke command to stop the application. Next we use the uninstall command of the AdminApp object to uninstall the application.*

```
applist = AdminApp.list().split("\n")
#
for ap in applist:
    if ap == "PolicyIVPV5":
        print "Found and uninstalling ",
        print ap
        am =
AdminControl.queryNames('cell=s1cell,node=s1nodec,type=ApplicationManager,p
rocess=s1sr01c,*')
        AdminControl.invoke(am, 'stopApplication', 'PolicyIVPV5')
        AdminApp.uninstall(ap)
    continue
```

*Next we install the application. The two arguments to the install command of the AdminApp object are the path to the ear file, and the options for the application install. Until you become very experienced in this area, there are two ways to find the proper options to use. One is to use the technique described earlier in the previous section that uses the command assistance popup in the adminconsole. The other is to do the application install interactively by invoking the wsadmin.sh command without the –f parameter for a script. Example:*

```
/wasv61config/s1cell/s1dmnode/DeploymentManager/profiles/default/bin/wsadmin.sh
–lang jython –javaoptions –Dscript.encoding=IBM-1047
```

*You can then run the command: AdminApp.installInteractive('/tmp/PolicyIVPV5.ear') and respond to the "prompts". It will be the command line equivalent of using the adminconsole to install the application. After the installation has been completed, you can look at the WASX7278I message in the output log for wsadmin (the default is called wsadmin.traceout). It will have all of the options specified for the install command and you can simply cut and paste them into your script. The InfoCenter has info on this at the link:*

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/zseries/ae/rxml_taskinfo.html

*One you have the location of the ear file, and all of the options specified, you simply issue the install command on the AdminApp object.*

```
#
#  Installing app
#
print "Installing application PolicyIVPV5"
#
#
earfile = "/tmp/PolicyIVPV5.ear"
appopts = "[-appname "
appopts = appopts + "PolicyIVPV5 "
appopts = appopts + "-reloadEnabled -reloadInterval 0 -
createMBeansForResources "
appopts = appopts + "-MapModulesToServers [[ PolicyIVPV5CMP
PolicyIVPV5CMP.jar,META-INF/ejb-jar.xml "
appopts = appopts + " WebSphere:cell=s1cell,node=s1nodec,server=s1sr01c ] "
appopts = appopts + " [ PolicyIVPV5BMP PolicyIVPV5BMP.jar,META-INF/ejb-
jar.xml "
appopts = appopts + " WebSphere:cell=s1cell,node=s1nodec,server=s1sr01c ] "
appopts = appopts + " [ PolicyIVPV5Session  PolicyIVPV5Session.jar,META-
INF/ejb-jar.xml "
appopts = appopts + " WebSphere:cell=s1cell,node=s1nodec,server=s1sr01c ] "
appopts = appopts + " [ PolicyIVPV5Web PolicyIVPV5Web.war,WEB-INF/web.xml "
appopts = appopts + " WebSphere:cell=s1cell,node=s1nodec,server=s1sr01c ]]
"
appopts = appopts + " -MapResRefToEJB [[ PolicyIVPV5BMP PolicyBMPV5 "
appopts = appopts + " PolicyIVPV5BMP.jar,META-INF/ejb-jar.xml jdbc/policy
javax.sql.DataSource "
appopts = appopts + " jdbc/IVPDS ]] "
appopts = appopts + "  -DataSourceFor20CMPBeans [[ PolicyIVPV5CMP
PolicyCMPV5 "
appopts = appopts + " PolicyIVPV5CMP.jar,META-INF/ejb-jar.xml jdbc/IVPDS
cmpBinding.perConnectionFactory ]] "
appopts = appopts + "  -DataSourceFor20EJBModules [[ PolicyIVPV5CMP
PolicyIVPV5CMP.jar,META-INF/ejb-jar.xml "
appopts = appopts + "  jdbc/IVPDS cmpBinding.perConnectionFactory ]]]"
#
#print "appopts = " + appopts
#
AdminApp.install(earfile, appopts)
#
```

*Once the install is done, we do the normal save and synchronize operations.*

```
#
#  Save the config
#
AdminConfig.save()
#
#  Sync the node.
#
nl = AdminConfig.list('Node').split("\n")
for n in nl:
    nn = AdminConfig.showAttribute(n, 'name')
    objn = "type=NodeSync,node=" + nn + ",*"
    Syncl = AdminControl.completeObjectName(objn)
    if Syncl != "":
        AdminControl.invoke(Syncl, 'sync')
        print "Done with node " + nn
    else:
        print "Skipping node " + nn
    continue
#
```

*Now we use queryNames to find the correct Mbean again, and use the invoke command of the AdminControl object to start the application.*

```
#  Start the application
#
am =
AdminControl.queryNames('cell=s1cell,node=s1nodec,type=ApplicationManager,p
rocess=s1sr01c,*')
AdminControl.invoke(am, 'startApplication', 'PolicyIVPV5')
#  All done!
```

As a final example, let's look at a script that is used to delete and add some variables immediately after initial setup of a cell. This script is used to remove the ras_log_logstreamName variable from all of the scopes in which it exists, and to add to the cell scope, the variables necessary to redirect all of the WTO messages to sysout ddnames.

This example allows us to demonstrate another use of the split function. When we get a list of all of the variable maps in the cell, it is in the following form:

```
(cells/s1cell/nodes/s1dmnode/servers/dmgr|variables.xml#VariableMap_1)
(cells/s1cell/nodes/s1dmnode|variables.xml#VariableMap_1)
(cells/s1cell/nodes/s1nodec/servers/s1sr01c|variables.xml#VariableMap_1)
(cells/s1cell/nodes/s1nodec/servers/s1sr09t|variables.xml#VariableMap_1)
(cells/s1cell/nodes/s1nodec|variables.xml#VariableMap_1)
(cells/s1cell|variables.xml#VariableMap_1)
```

As you can see, the scope of the variables can be determined by examining the part of the object id up to the vertical bar | symbol. In this example, we want to add variables to the cell scope, which can be identified by the string "(cells/s1cell" as the entire portion of the string prior to the | character. We use the split function to isolate this.

```
import sys
#
#  Sample invocation:
#
#  ./wsadmin.sh -lang jython -javaoption -Dscript.encoding=IBM-1047 -f
scriptpath/initsetvars.py cellname
#
#
#  Set up some variables after initial setup of a cell.  Specifically, delete
all occurances of the
#  ras_log_logstreamName variable in all scopes so that the annoying RACF
message the log failure message
#  will stop appearing if you haven't actually setup logstreams and add all of
the variables necessary
#  to cause the many WTO messages to be re-routed to the DEFALTDD and HRDCPYDD
ddnames.  The ddnames
#  need to be added to all of the procs that end in 'Z'.
#
def printHelp():
    print "This script sets some initial variables after initial setup of a
cell"
    print "Format is initsetvars.py cellname"
#
if(len(sys.argv) > 0):
    # get cell name from the command line
    cellname = sys.argv[0]
    print "Cell name is ",
    print cellname
else:
    printHelp()
    sys.exit()
#
#   set up the cellname comparison variable
#
```
*This next line concatenates the constant "(cells/" with the value for cellname passed in as an argument. The value in cellvar will be (cells/s1cell in the case where the cellname passed as an argument is s1cell.*
```
cellvar = "(cells/" + cellname
print 'Cell variable is ',
print cellvar
```

*This section of code gets a list of all of the variable maps in the cell. It then iterates through the list and gets a list of variable entries for each variable map. It next examines each variable entry, comparing the symbolic name attribute for the value of the variable we are interested in, ras_logstream_logstreamName. When it gets an equal comparison, it removes that variable entry from the map. This effectively deletes the variable.*
```
vlist = AdminConfig.list('VariableMap').split('\n')
for v in vlist:
    vselist = AdminConfig.list('VariableSubstitutionEntry', v).split('\n')
    for vs in vselist:
        if AdminConfig.showAttribute(vs, 'symbolicName') ==
'ras_log_logstreamName':
            rc = AdminConfig.remove(vs)
        continue
    continue
```
*Here we are setting up the attribute pairs, and the list of attribute pairs that will be used as arguments for the create method.*
```
sname1 = ['symbolicName', 'DAEMON_ras_default_msg_dd']
value1 = ['value', 'DEFALTDD']
arglist1 = [sname1, value1]
sname2 = ['symbolicName', 'DAEMON_ras_hardcopy_msg_dd']
value2 = ['value', 'HRDCPYDD']
arglist2 = [sname2, value2]
sname3 = ['symbolicName', 'ras_default_msg_dd']
arglist3 = [sname3, value1]
sname4 = ['symbolicName', 'ras_hardcopy_msg_dd']
arglist4 = [sname4, value2]
```
*We are looking for the variable map for the cell scope, as we discussed earlier. When we find it, we simply drive the create method once for each variable entry we want to add to the map.*
```
vlist = AdminConfig.list('VariableMap').split('\n')
for v in vlist:
    if v.split('|')[0] == cellvar:
        rc = AdminConfig.create('VariableSubstitutionEntry', v, arglist1)
        rc = AdminConfig.create('VariableSubstitutionEntry', v, arglist2)
        rc = AdminConfig.create('VariableSubstitutionEntry', v, arglist3)
        rc = AdminConfig.create('VariableSubstitutionEntry', v, arglist4)
    continue
```
*Then all that is necessary is the familiar save and synch to complete the script.*
```
#  Save the config....
```

```
rc = AdminConfig.save()
#  Synch with nodes...
nl = AdminConfig.list('Node').split("\n")
for n in nl:
    nn = AdminConfig.showAttribute(n, 'name')
    objn = "type=NodeSync,node=" + nn + ",*"
    Syncl = AdminControl.completeObjectName(objn)
    if Syncl != "":
        AdminControl.invoke(Syncl, 'sync')
        print "Done with node " + nn
    else:
        print "Skipping node " + nn
    continue
# All done!
```

All of the scripts used in this section have been tested on a V6.1 WebSphere on z/OS cell and worked properly. It would probably be a good idea to include more error checking. These examples demonstrate some of what is possible, and show how a few common things can be accomplished via the Jython scripting interface.

# **Jython Scripting Language**
used with the WSADMIN tool
**WP100963**

## ***Document Change History***

Check the date in the footer of the document for the version of the document.

| | |
|---|---|
| January 18, 2007 | Original Document |
| April 28, 2008 | Updated the accompanying zip file with a fresher copy of the initsetvars.py script.<br>Added copyright notice. |

**End of Document WP100963**