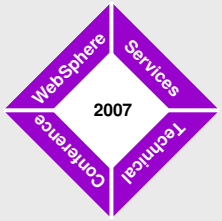
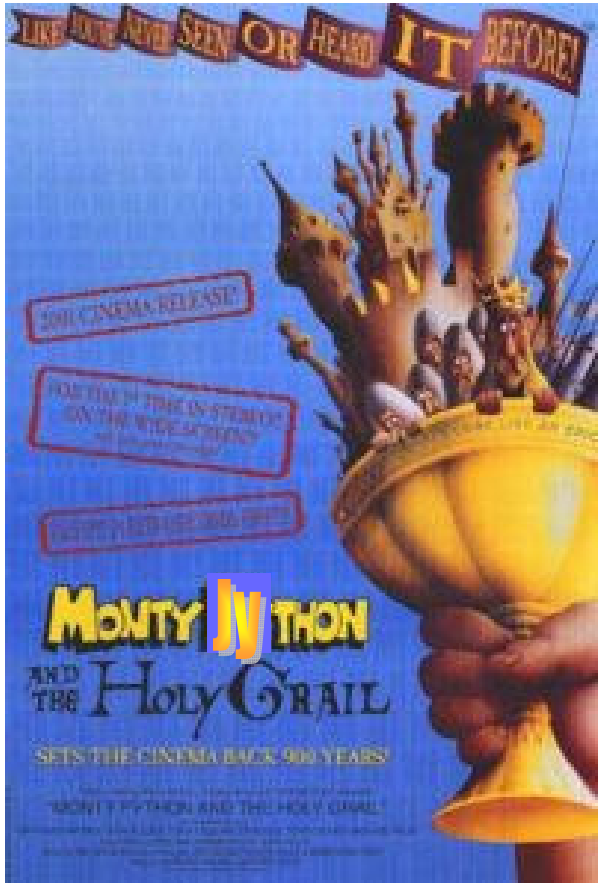


Monty Jython's Scripting Circus

Andrew Simms
(ISSW UKISA, Andrew Simms/UK/IBM, simmsa@uk.ibm.com)

Objectives

- Our Holy Grail: *To work out some recommended practices for using Jython in WAS scripting*
- How we're going to get there:
  - Quick look at what's in the WAS 6.1 Application Server Toolkit (AST)
  - Describe the essentials of the Jython language
  - Look at some more advanced Jython features
  - Pick out some key things as we proceed
  - Demonstrate bit and pieces of Jython
  - Suggest some guidelines for what scripts should look like



It's: A short history of Jython

- Python invented by Guido van Rossum in 1991
  - Operating system-independent
  - Object-oriented
  - Based on a language called ABC
  - Designed to be readable
  - Named after the TV programme
  - Small language core with extensive libraries
- Jython is a Java implementation of Python
  - WAS 6.1's Jython is version 2.1
    - Latest Jython is 2.2, Python 2.5

Jython programming in the AST

- Purpose:
  - To greatly ease **`wsadmin`** scripting
    - by simplifying the *development* of Jython scripts
      - using the Jython Editor
    - by simplifying *debugging* Jython scripts
      - using the Jython Debugger
    - by *generating* scriptlets
      - using the Admin Console Command Assist feature
    - by *converting* existing Jacl scripts to Jython
      - using the Jacl2Jython conversion tool

Jython editor in the AST

- Text editing (find, replace, indent, undo, etc)
- Syntax colouring
- **`wsadmin`** keyword & method assistance
  - Keyword & method syntax detection and colouring
  - Keyword & method code completion (including parameters)
  - Keyword & method context assistance and flyover help
- Outline view (classes & methods & loops)
- Provides integration with Jython Debugger
- Has “self-evident” usage (Eclipse consistent)
- NO compiler parse errors, NO parameter type checks

Jython debugger in the AST

- Uses local server runtime(s) for **`wsadmin`** execution
  - Can target (compatible) remote servers (using `-host -port`)
- Can run Jacl and Jython scripts
- But debugging is Jython only
  - Local v6.1
  - Breakpoints, step-over etc
  - Variables view (cannot change variable contents)
  - Stack frame view (variables reflect current level)

Command assistance in the AST

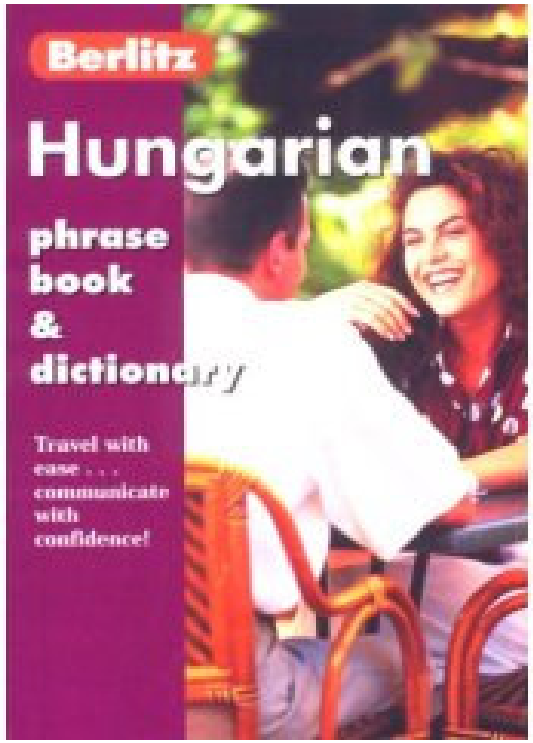
- Command Assist View in AST can receive configuration changes made via the Admin Console
  - Some (not all) actions result in Jython commands being generated
  - Limited in 6.1 – maybe 10%
  - Expect more in later releases
- Insert generated code into a script using the Jython Editor script
  - Will need further editing

Jacl2Jython utility from the AST

- Takes your Jacl scripts and converts them to Jython
  - Typically does 95-98% code conversion
  - Inserts problem warnings into the converted code
- The developer must then:
  - Manually verify all of the preliminary conversion
  - Modify some code to make it function as originally intended
  - Thoroughly test the resulting script
- Does it produce Jython code in a style you would use if writing from scratch?
  - Not if you want to use the OO features

Jython: Language essentials

- Indentation and comments
- Statements
- Trapping exceptions
- Data types
- Strings, tuples and lists
- Dictionaries
- Functions (classless methods)
- Built-in functions



Indentation and comments

- It's an indented language
  - No curly brackets
  - Colon and indentation instead

Beware mixing tabs and spaces

- Comments: use # and *anonymous string literals*:

```
# This is a comment
'This is a comment'
"""This is a comment that spreads across many lines
between triple-quotes.  So a good way to comment out
code is to use triple-quotes"""
```

In interactive wsadmin you can't cut and paste comments that spread over two or more lines. Not a problem for the AST.

Statements

- Much more like Java than Jacl (Tcl)
- Statement syntax is Java-like:

```
for x in (1, 2, 3):
  print "x=", x
else:
  print "Counter-intuitive that this gets executed"

for z in (range(45, 55)):
  if z == 50:
    break
  else:
    print z
else:
  print "Broke out so this won't get printed"
```

"else" here is a misnomer – always executed unless you use "break" to exit from the for loop

1

In interactive wsadmin make sure you type the indentation correctly, and you may need a blank line at the end of blocks. Not a problem for the AST.

Statements

- Multiple assignments (Fibonacci series):

```
a, b = 0, 1
while b < 1000000:
    print b
    a, b = b, a+b
```

This is actually a tuple assignment as we shall see later

2

- Some other things:

```
del a,b
x = 1
x += 1     # No x++
assert x == 2
pass       # no-op
```

try … except … else … finally …

- Try/except/else:

```
try:
    v = 1 / 0
except ArithmeticError:
    print "You can't do that!"
except:
    print "This is a catch-all"
else:
    print "value = ", v
```

"else" here: if you go to the exceptions then the else doesn't get executed; if you don't go to the exceptions then it does get executed

- Try/finally:

```
try:
    doStuff
finally:
    doCleanUpStuff
```

finally: always gets executed and any exception re-raised after it executes

- Use `raise` to raise exceptions

Jython data classes

- All data classes are dynamic:

```
s = "hi there" ; v = 42
type(v) # -> <jclass org.python.core.PyInteger at 658253628>
```

- Numeric types:
  - Integer, long, float, complex
  - Numeric objects are immutable:

```
id(v) # -> 791424812
v += 1
id(v) # -> 791818034
```

- Types are detected syntactically:

```
vi = 42       # -> Integer
vl = 42L      # -> Long
vf = 42.1234  # -> Float
vc = 42+43j   # -> Complex
```

Can use lower-case el – but don't (indistinguishable from the number one)

Probably not much use in WAS scripting!

Sequences: Strings, tuples and lists

Type	Contents	Mut-able?	Delimiter	Examples

Strings	Character data only	No	Quotes	`s = "one two three four"`
				`s[5]  # indexing -> 'w'`
				`s[5:9]  # slicing -> 'wo t'`

Tuples	Any kind of object	No	Round brackets	`t = ("one", "two", "three", "four")`
				`t[1]  # -> 'two'`
				`t[:2]  # -> ('one', 'two')`

Lists	Any kind of object	Yes	Square brackets	`l = ["one", "two", "th ee", "four"]`
				`l[-1]  # -> "four"`
				`l[1:-2] # -> ['two']`
				`l[0::2] # -> ['one', 'th ']`

Think of a tuple as a constant list, but you can still change any mutable element it may have

`len()` tells you the sequence length

Use empty paired delimiters to get an empty sequence, e.g.
`L=[]`

Slicing and indexing apply to all sequences

Sequence mutability

- Strings are immutable:

```
s = "one two three four"
s[8:-1] = "buckle my shoe" # fails
```

- Tuples are immutable:

```
t = ("one", "two", "three", "four")
t[2:] = "buckle my shoe" # fails
```

- But lists are mutable:

```
l = ["one", "two", "three", "four"]
l[-2:] = "buckle my shoe"
print l          # probably not what you wanted
```

3

```
l = ["one", "two", "three", "four"]
l[-2:] = ["buckle", "my", "shoe"]
print l          # better
```

4

Sequences: Strings

- Use single, double or triple quotes
- Reverse quotes (equivalent to the `repr()` function):

```
v = 42
s = `v / 6`
type(s)      # s is the string '7' not the integer 7
s = int(s)
type(s)      # now it is
```

- Some useful `PyString` methods:

```
capitalize(), endswith(), find(), isxxx(), join(), lower(),
rfind(), split(), splitlines(), startswith(), strip(), upper()
```

- `join()` converts a sequence to a string:

```
s1 = ":"
s2 = s1.join(["join", "with", "colons"])  # 'join:with:colons'
```

- `split()` converts a string to a list:

```
s2.split("i")       # -> ['jo', 'n:w', 'th:colons']
```

Sequences: Tuples

- Contains references to any object type
- Those objects can be mutable but the tuple itself is immutable
- No methods available for tuples
- Represented by round brackets but you don't have to specify them

```
t = "one", "two", "three", "four"
t = ("one", "two", "three", "four")
t = (("one", "two", "three", "four"))
```

But do so for clarity

- Objects can be of different types:

```
t = ("one", 2, 3L, 4.0+5.0j)
```

Beware:
`t=("one")` is a string
You need a trailing comma:
`t=("one",)`

Sequences: Lists

- Contains references to any object type
- The only sequence type that is mutable
- Represented by square brackets

```
l = ["one", 2, 3L, 4.0+5.0j]
```

- `PyList` methods:

```
append(), count(), extend(), index(), insert(), pop(),
remove(), reverse(), sort()
```

Convert a list to a tuple with `list(seq))` and vice versa with `tuple(seq)`

- Examples:

```
l.append(6.0E7)          # appends one object to the list
l.count(60000000)        # 1 (how often does the value occur)
l.extend([7, "eight"]) # appends a list to the list
l.index(4+5j)            # 3 (the index of this value)
l.insert(3, 2.5)         # inserts 2.5 in index 3
l.pop(1)                 # 2 (and removes it from the list)
l.remove(4+5j)           # removes this value from the list
l.reverse()              # reverses the list order
l.sort()                 # sorts the list (in some way)
```

List comprehension

- A syntax that allows you to create one list from another by applying a function or expression to each member:

```
[expr for var1 in seq1 if test1 for var2 in seq2 if test2 . . .]
```

- Exploit this to set heap sizes for all of your app servers in one line!

None of the Admin* functions return a true Jython list.

```
14 [AdminConfig.modify(x, [["initialHeapSize", 64],
5   ["maximumHeapSize", 128]]) for x in
14  AdminConfig.list("JavaVirtualMachine").splitlines() if
    x.find("nodeagent") == -1 and x.find("dmgr") == -1]
```

Dictionaries (mapping objects)

Dictionaries are very useful as we will see later

- Connects a set of immutable keys to a set of objects
- Enclose with curly brackets and colon- and comma-separated values:

```
chineseLanguages = ["Mandarin Chinese", "Cantonese"]
indianLanguages = ["Hindi", "Urdu", "Gujarati"]
china = ["Beijing", 1316E6, chineseLanguages]
india = ["New Delhi", 1110E6, indianLanguages]
cdict = {"China": china, "India": india}

# -> {'China': ['Beijing', 1.316E9, ['Mandarin Chinese',
'Cantonese']], 'India': ['New Delhi', 1.11E9, ['Hindi', 'Urdu',
'Gujarati']]}
```

6

Dictionaries (mapping objects)

- `PyDictionary` methods:

```
clear(), copy(), get(), has_key(), items(), keys(), popitem(),
setdefault(), update(), values()
```

- Examples:

```
cdict("England") = ["London", 4.8E7, ["Cockney", "Geordie",
"Sassenach"]]

cdict.update({"Scotland": ["Edinburgh", 1.0E7, ["English"]]})

cdict.get("India")          # returns value if present
cdict.has_key("Egypt")      # 0 (not present)
cdict.keys()                # ['India', 'China', 'England']
cdict.items()               # returns a list of tuples
cdict.popitem()             # pops an item as a tuple
cdict.setdefault("Egypt") # appends a key pair if not present
cdict.values()              # returns a list of values
copy=cdict.copy()           # performs a shallow copy
del cdict["Egypt"]          # deletes an entry
cdict.clear()               # empties the dictionary
```

Functions

- *Functions* are methods defined outside a class

```
def myFunction(p1, p2, p3):
    doSomeStuff
    return whatever
```

Function names:
• Don't use underscores as these have special meanings
• Don't use built-in function names

- Can return multiple values
  - A tuple is constructed

This is really useful – not restricted to returning a single value

- Functions can have attributes as well as variables:

```
def myFunction():
  myFunction.attr1 = "bonjour"
  attr2 = "hello"

myFunction()
myFunction.attr2 = "g'day"
```

attr2 is a local variable but attr1 is available externally

This is a new attribute assigned externally

Right room for an argument

- Positional, default values, variable args:

```
def myFunction(p1, p2="def", *p3, **p4):
    print vars()
```

Varargs: extra positional args are passed as a tuple

Varargs: extra key-value args are passed as a dictionary

```
myFunction("abc")
myFunction("abc", "ghi", "jkl", "mno")
myFunction(p2="xyz", p1="uvw")
myFunction("a", "b", "c", "d", id1="e", id2="f")
```

This is a great way of keeping a function's signature constant yet allowing arbitrary parameters to be passed to it

Doc strings

- Place an *anonymous string literal* after a function definition. Its content becomes that function's *doc string*.
- Print its documentation using `<name>.__doc__`

```
def someFunction():
    """someFunction does something or other"""
    pass



someFunction.__doc__    # -> someFunction does something or other
```

A Jython library becomes self-documenting. Can see this using the AST.

Built-in functions

- `type()` – type of an object:
- `id()` – identity of an object:
- Numeric functions:

```
hex(), oct(), abs(), ...
```

- Type conversions:

```
int(3.14159), tuple("abcd"), ...
```

- File handling:

```
open("/tmp/myFile", "r")
```

- Sequence generators:

```
range(3, 17, 2)
xrange(3, 1234567, 2)
```

- Class attributes:
  - dot notation
  - also: `hasattr(), delattr(), getattr(), setattr()`
- Many more

Jython: Classes and other advanced features

- Namespaces
- Functional programming
- Regular expressions
- Threads
- Modules and packages
- Classes
- Using Java in Jython

Namespaces: Bruces and new Bruces

- *Static* and *statically nested (lexical)* scoping
- Static scoping:
  - Two namespaces: *locals* and *globals*

```
bruce = 1
def changeBruce():
  # global bruce
  # bruce = 10
  bruce += 1
  print bruce

changeBruce()
print bruce
```

Without the global and without the assignment Jython treats this as a new bruce. Error: bruce isn't defined when incremented.

With this assignment but without the global, Jython sees this bruce as a new bruce

With the global we only have one bruce

Namespaces: Nested functions

- Statically nested scoping:
  - Names used in outer functions are not visible in the inner function without a special import

```
from __future__ import nested_scopes

def outer(x, y):
    def inner(z):
        if z > 0:
            print z, y
            inner(z-1)
    inner(x)

outer(3, "bruce")
```

Jython's way of introducing new features

Functional programming

- Create *anonymous functions* using *lambda forms* which have expressions but no statements:

```
isLeapYear = lambda year: not (year % 400 and (year % 4 or not
  year % 100))
print "2000 - ", isLeapYear(2000)
print "2007 - ", isLeapYear(2007)
print "2008 - ", isLeapYear(2008)
print "2100 - ", isLeapYear(2100)
```

9

Note that this returns a function

- `map` iterates over sequences calling a function on each member:

```
map(lambda x: x*x, range(10))

map(lambda x,y: x>y and x-y or y-x, [1, 5, 8], [3, 1, 7])
    # -> [2, 4, 1]
```

Functional programming

- `filter` iterates over a sequence returning a subset of its values where the called function returns true:

```
set1 = range(0, 200, 7)
set2 = range(0, 200, 3)
filter(lambda x: x in set1, set2)
  # -> [0, 21, 42, 63, 84, 105, 126, 147, 168, 189]
```

- `reduce` computes a single value by applying a two-arg function recursively:

```
reduce(lambda x, y: x+y, range(11))
```

Three ways of doing recursion

- Ordinary functions can be recursive:

```
def fact(x):
  x = long(x)
  if x == 0:
    return 1
  return x * fact(long(x-1))
```

Ultimately breaks with stack overflow, e.g. fact(1712)

- You can use an anonymous function:

```
fact = lambda num: num == 1 or num * fact(long(num-1))
```

Ditto

- You can use the `reduce()` function, which eats a sequence applying a recursive function to it:

```
fact = lambda num: num > 0 and
  reduce(lambda x, y: long(x)*long(y), range(1, num + 1)) or 0
```

Doesn't break

Functional programming examples using AdminConfig (1)

- Test whether a name is a valid configurable object:

```
isValidType = lambda x: x in AdminConfig.types().splitlines()
isValidType("JavaVirtualMachine") # -> 1
isValidType("Garbage")            # -> 0
```

- Set heap sizes for all of your app servers in one line (as earlier):

```
map(lambda x:
    AdminConfig.modify(x, [["initialHeapSize", 64],
                           ["maximumHeapSize", 128]]),
    filter(lambda x:
           x.find("nodeagent") == -1 and x.find("dmgr") == -1,
           AdminConfig.list("JavaVirtualMachine").splitlines()))
```

14 10 14

Functional programming examples using AdminConfig (2)

- Test whether some configurable type has a named attribute:

```
isAttribute = lambda x, type:
  isValidType(type) and x in
      map(lambda z: z.split()[0],
          AdminConfig.attributes(type).splitlines())
isAttribute("systemProperties", "JavaVirtualMachine") # -> 1
isAttribute("garbage", "JavaVirtualMachine")          # -> 0
```

Each entry is a string in the format attrName-space-attrType

- Store attributes of a configurable type in a Jython dictionary:

```
from __future__ import nested_scopes
attsToDict = lambda type, dict:
  map(lambda x:
    dict(x[0:x.index(" ")]) = x[x.index(" ") + 1:] ,
      AdminConfig.attributes(type).splitlines())

jvmatts = {}
attsToDict("JavaVirtualMachine", jvmatts)
jvmatts.has_key("systemProperties")
jvmatts.get("systemProperties")
```

The attrType may contain spaces

Builds dictionary of all atts including those whose values are references to other types

Functional programming examples using AdminConfig (3)

- Store just the simple attribute names and append the type name to each:

```
from __future__ import nested_scopes
attsToDict = lambda type, dict:
  map(lambda x:
      dict(x[0:x.index(" ")] + "_" + type) = x[x.index(" ") + 1:], \
      filter(lambda x: x.endswith("*") == 0 and x.endswith("@") == 0, \
        AdminConfig.attributes(type).splitlines()))
```

Contains entries such as:
`{initialHeapSize_JavaVirtualMachine int}`

- Build a Jython dictionary of all simple attribute names of all object types:

```
bigDict = {}
map(lambda x: attsToDict(x, bigDict), AdminConfig.types().splitlines())
```

11

Functional programming examples using AdminConfig (4)

- Use the dictionary to validate and set values:

One generic function serving most update needs

```
def setValues(baseType, simpleName, qualifier=None, **setThese):
  objid = AdminConfig.getid("/" + baseType + ":" + simpleName + "/")
  for attrUndType, value in setThese.items():
    undPos = attrUndType.find("_")
    if bigDict.has_key(attrUndType):
      attrName = attrUndType[:undPos] ; attrType = attrUndType[undPos+1:]
      attrTypeIdList = AdminConfig.list(attrType, objid).splitlines()
      if qualifier:
        for listItem in attrTypeIdList:
          if listItem.startswith(qualifier):
            attrTypeId = listItem
            break
      else:
        if len(attrTypeIdList) == 1:
          attrTypeId = attrTypeIdList[0]
      AdminConfig.modify(attrTypeId, [[attrName, value]])
```

15

Error checking removed to keep this example simple

Functional programming examples using AdminConfig (4)

- Use the dictionary to validate and set values:

16

```
setValues("Server", "engine1",
    initialHeapSize_JavaVirtualMachine = 1024,
    maxInMemorySessionCount_TuningParams = 200,
    parallelStartEnabled_Server = "false")


setValues("Server", "engine1",
    description_ThreadPool="some description",
    minimumSize_ThreadPool=2,
    maximumSize_ThreadPool = 17,
    qualifier="WebContainer")
```

setValues() works for simple changes. Doesn't create or delete objects. Doesn't add or delete attributes to existing objects (e.g. custom properties)

Regular expressions

- Similar to regexp in other languages
- Can get unreadable – use raw strings (introduced by "r")
- Produce a more readable list of application servers:

```
import re
for appserv in
 AdminConfig.list("ApplicationServer").splitlines():
(12)  print re.sub(r".*\(cells/.+/nodes/(.+)/servers/(.+)\|.+\)",
 r"\2 on \1", appserv)
```

Threads

- Run an object and arg tuple in a new thread:

```
import thread

mynode = "appServNode"

def startAServer(server):
  print "I'm: ", server
  AdminControl.startServer(server, mynode)
  print "I'm done: ", server

for server in "server1", "server2":
  thread.start_new_thread(startAServer, (server,))
```

Starts application servers in parallel threads. Note the tuple passed as an argument to the function

Modules and packages

- *Module*: a `.py` file containing Jython code
  - Can reload modules you're working on using `reload()`
- *Package*: Hierarchy of modules in a directory tree
  - Is a package if there's a file called `__init__.py` in the directory
- Use the `import` statement to load them
  - `import A.B.C` implies A and B are packages, C is a module or package
- Special variables: `__name__`, `__doc__`, `__file__`, `__all__`
- `dir(A.B.C)`, `dir(A.B.C.someFunction)` tell you what's available

Importing modules and packages

- Four types of import:
  - Import everything in a hierarchy:

```
import sys
```

  - Import a subset of a hierarchy:

```
from java import util
```

  - Import a hierarchy but give it a new name:

```
import os as myOS
```

  - Import a subset but give it a new name:

```
from sys import packageManager as pm
```

Can import WebSphere classes too

Using AdminConfig etc from packages

- Suppose `A.B.C.py` contains this:

```
def listServers():
  AdminConfig.list("Servers")
```

- and you invoke it from `D.py`:

```
import A.B.C as C
C.listServers() # -> NameError
```

Not even placing `global AdminConfig` in C.py works. Global in Jython is not the same as in Jacl

- Could change `D.py` to call `execfile("<path>/C.py")`, but this collapses everything to a single module – you might get name clashes. Would then call `listServers()` not `C.listServers()`.
- Instead you could change `C.py` and retain the hierarchy:

```
import com.ibm.ws.scripting.AdminConfigClient as Scripting
AdminConfig = Scripting.getInstance()

def listServers():
  AdminConfig.list("Servers")
```

Clearly this is WAS-version specific so isn't a great solution

Some useful Jython libraries

- Need to import these libraries to use them
- Useful things in `sys`:

```
argv, modules, path, platform, version, exc_info()
```

Java platform (e.g. 1.5)

Jython version (e.g. 2.1)

- Use `os` for platform-independence

```
os.linesep, os.pathsep, os.mkdir(), os.stat(), os.listdir(),
os.path.join(),os.path.isfile(),os.path.isdir(),
os.path.dirname(), ...
```

- Use `glob` for file pattern matching
- Use `re` for regular expressions
- Unit testing with `PyUnit`

For really robust admin scripts:

```
import unittest

<body of module>

if __name__ == '__main__':
    <test cases>
```

Classes

- No explicit private, protected, public tags
  - Implicit name prefixes:
    - One underscore => private
    - Two underscores => very private
    - But can always access via the full name
- Defining a class:

```
class class_name[(inheritance)]:
    <code>
class Myclass:
    """documentation"""
    <class-level attributes>
    <method 1>:
        <instance-level attributes>
```

Note: no class-level (static) methods

Can dynamically create attributes

Create class-level attributes within a method by prefixing with the class name

Instance methods and constructors

- Use the `__init__` method as a constructor
- Instance method definitions require an identifier as the first parameter (conventionally "self"):

```
class JVM:
  def __init__(self, server = "server1"):
    serverId = AdminConfig.getid("/Server:"+server)
    stringJvmIds = AdminConfig.list("JavaVirtualMachine", serverId)
    listJvmIds = stringJvmIds.split()
    if len(listJvmIds) != 1:
        raise "JVMIdException"
    self.jvmString = listJvmIds[0]

  def getHeapSizes(self):
    minHeap = AdminConfig.showAttribute(self.jvmString, "initialHeapSize")
    maxHeap = AdminConfig.showAttribute(self.jvmString, "maximumHeapSize")
    return (minHeap, maxHeap)

jvmid = JVM("controller")
minHeap, maxHeap = jvmid.getHeapSizes()
print minHeap, maxHeap
```

13

Class Inheritance

- Can inherit from multiple classes:

```
class OrderItem(Cust, Stock):
  def __init__(self, custref, stockref, qty):
    self.custref = custref
    self.stockref = stockref
    if stockref.qty - qty >= 0:
      self.qty = qty
      stockref.qty -= qty
    else:
      print "Not enough in stock"
  def showQty(self):
    print self.custref.id, ":", self.custref.name, ":",
 self.stockref.code, ":", self.qty

myitem = orderItem(alphaOrg, stock1, 14)
myitem.showQty()
```

Java from Jython

- A Jython class can only inherit from one Java class, but many Jython classes
- A Jython subclass cannot access a Java class's:
  - protected static methods and fields
  - protected instance fields
  - package-protected members
- Just import the Java classes and off you go:

```
from java.lang import System
from java.lang import String
x = String("Spamalot")
if x.startsWith("Spam"):
  System.out.println("Spam spam spam spam")
```

And now for something completely different

- Making **`wsadmin`** scripts more readable, robust, maintainable, extendable
- Making interactive administration a more friendly experience
- Suggested conventions

Scripts should be environment-independent

- Scripts must be independent of the target environment
  - Don't edit the scripts as you move from environment to environment
  - Externalise in properties files
  - Could split into "definitely unique per environment" and "common but seems sensible to externalise"
  - Choice:
    - Make the property values Jython sequences
    - Make them more human-readable
  - Choice:
    - Use `execfile()` to execute those properties files
    - Use the `-p` option on the `wsadmin` command line

Scripts should be modularised

- EITHER:
  - Develop a `common.py`
  - Develop individual `.py` files that wrap up a bunch of `AdminConfig`, `AdminApp` objects
    - e.g. `jdbc.py`, `appserver.py`, `cluster.py`
  - `execfile()` them all and have a single namespace for your entire scripts
- OR:
  - Use Jython packages, modules and classes to structure it in an OO fashion
  - `import` these and have separate namespaces

Good practices

- Strive for platform independence
- Never make N calls from a shell script to `wsadmin` each passing a separate `-c` option
  - Each involves connecting to WAS
  - Make those N calls from within a `wsadmin` script
- Script the entire configuration build
  - Tear down and rebuild
- Simplify the `wsadmin` complexity:
  - Hide the verbose naming convention
    - Work in WAS scopes
    - Display simple names, work out the verbose ones
  - Hide the navigation hierarchy

More good practices

- You probably don't need to dive into Java from Jython
  - Many (most?) administrators are not Java programmers
- Don't just provide create, modify and delete functions
  - List and show are also useful
  - Build a script library – a library of functions and/or classes & methods
- Move away from positional parameters on functions
  - Allow keywords or dictionaries to be passed in
- Make it possible to use your library easily interactively
  - This is hardly the case with out-of-the-box `wsadmin`
- Conventions:
  - Class names have upper case first char
  - Method and function names in camelCase
  - Spaces around operators and parameters
  - No space before colon in dictionaries
  - Indent consistently either 2 or 4 spaces

Summary

- Déjà vu:
  - We've seen there are excellent Jython productivity tools in the WAS 6.1 AST
  - We've looked at the basic and some advanced features of the Jython language
  - We've established some recommended practices for Jython scripting

References

- Jython Essentials, Samuele Pedroni & Noel Rappin (O'Reilly)
- Jython for Java Programmers, Robert Bill (New Riders)

Spanish Inquisition

- Nobody expects . . .