Oracle® Data Integrator

Jython Quick Reference

10g Release 3 (10.1.3)

September 2008

Table Of Contents

This manual provides a reference of the **Jython** scripting language. It is intended for developers who want to use Jython scripting in their integration scenarios.

Organization of This Manual

This manual contains the following:

- **Chapter 1 - The Basics** explains the basics of the Jython syntax.
- **Chapter 2 - Using Jython in Oracle Data Integrator** details the use of Jython in Oracle Data Integrator.
- **Chapter 3 - Examples** provide samples scripts for Jython.

The comprehensive Jython documentation is available at http://www.jython.org

The Basics

Oracle Data Integrator and Jython

Jython (Java version of Python) is an object-oriented scripting language. Jython scripts run on any platform that has a java virtual machine.

Oracle Data Integrator includes the Jython Interpreter within the execution agent. The agent is able to run script written in Jython.

Oracle Data Integrator users may write procedures or knowledge modules using Jython, and may mix Jython code with SQL, PL/SQL, OS Calls, etc.

Thanks to Jython, the programming capabilities of Oracle Data Integrator are dramatically increased. It is possible to perform complex processing with strings, lists, "dictionaries", call FTP modules, manage files, integrate external Java classes, etc.

> **Note**: To use Jython code in KM procedure commands or procedures commands, you must systematically set the technology to `Jython`.

Points to Remember

The basic rules to write a Jython program are:

Code execution

Statements are executed in sequence up to a control structure: `if`, `for`, `while`, `raise` , or a function call.

Block

A block is defined by lines with the same indentation level (spaces or tabulations).

Statements

A statement stops at the end of a line, and can be continued on several lines if they end with a `\`, or if they are enclosed in `()`, `[]`, `{}` or `'''`. Several instructions can be on the same line if they are separated with a `;`.

Comments

A comment starts with a hash character `#` and ends at the end of the physical line.

String Documentation

If a function, a module or a class starts with a string constant, this string is stored in the `__doc__` attribute of the object.

Examples

Simple program that displays "Hello World"

```
# Assign a value to a string
s = 'Hello World'
# Display the value
print s
```

Program that displays "Hello World" 4 times

```
s = 'Hello World %d'
for i in range(4):
   j = i * 2
   print s % j
```

Keywords

The following identifiers are reserved words, or keywords, in Jython.

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass
def       finally   in        print
```

Please also note the following points.

- A statement must be written on one single line. To split a statement over several lines, you must use the `\` (backslash) character.
- Expressions in parentheses (), square brackets [] or curly braces {} can be split over more than one physical line without using backslashes.
- Several statements can be written on the same line if they are separated by semicolons (;).
- A comment begins with a hash character (`#`) that is not part of a string literal, and continues until the end of the line.

Operators

Operators, by precedence order:

Operator	Description
`lambda` `args: expr`	Anonymous function constructor
`x` **`or`** `y`	Logical OR

`x and y`	Logical AND
`not x`	Logical NOT

`x<y  x<=y  x>y  x>=y  x==y  x!=y  x<>y`
`x is y   x is not y`
`x in s   x not in s`
	Comparison operators (equal, not equal, is same object, belongs to a sequence...)

`x|y`	Bitwise OR
`x^y`	Exclusive OR
`x&y`	Bitwise AND
`x<<y   x>>y`	Left shift, right shift
`x+y  x-y`	Addition/concatenation, subtraction
`x*y  x/y  x%y`	Multiplication/repeat, division, modulo
`x**y`	Power
`+x, -x, ~x`	Identity, unary NOT, bitwise complement
`s[i]  s[i:j]  s.attr  f(...)`	Index, range, attribute qualification, function call
`(...) [...] {...} '...'`	Tuple, list, dictionary, conversion to string

Data Types

Numbers

- **Decimal integer**: `1234, 1234567890546378940L` (or `l`)
- **Octal integer**: `0177, 0177777777777777777L` (starts with a 0)
- **Hex integer**: `0xFF, 0XFFFFffffFFFFFFFFFFL` (starts with a 0x or 0X)
- **Long integer** (unlimited precision): 1234567890123456L (ends with L or l)
- **Float** (double precision): `3.14e-10, .001, 10., 1E3`
- **Complex numbers**: `1J, 2+3J, 4+5j` (ends with J or j. To create a complex number with a nonzero real part, add a floating point number to it with a '+')

Strings

The following sequences are defined as strings:

- `'Strings are enclosed in single quotes'`
- `"Another string in quotes"`

- `'String enclosed by single quotes with a " (double quote) in it'`
- `"String enclosed in double quotes with a ' in it"`
- `'''A string that contains carriage returns and ' can be enclosed in tree single quotes'''`
- `""" Triple quotes can also be used"""`
- `r' non interpreted strings (the \ are kept). Useful for the windows paths!'`
- `R" non interpreted string"`

Use \ at the end of the line to continue a string on several lines

2 adjacent strings are concatenated (ex : `'Oracle Data Integrator and' ' Python'` is equivalent to `'Data Integrator and Python'`.

Escape sequences

`\newline` : Ignored (Escape newline)

`\\` : Backslash (\)

`\e`: : Escape (ESC)

`\v` : Vertical Tabulation (VT)

`\'` : Single Quote (')

`\f` : Form Feed (FF)

`\OOO` : Character with Octal value OOO

`\"` : Double quote (")

`\n` :Line Feed (LF)

`\a` : Beep (BEL)

`\r` : Carriage Return (CR)

`\xHH` : Hexadecimal Character HH

`\b` : BackSpace (BS)

`\t` : Horizontal Tabulation (TAB)

`\uHHHH`: Hexadecial Unicode Character HHHH

`\AllCharacter:` left as such

String formatting

String formatting can be very useful. it is very close to the C function sprintf() :

Examples :

`"My tailor is %s..." % "rich"` returns `"My tailor is rich..."`

`"Tea %d %d %s" % (4, 2, "etc.")` returns `"Tea 4 2 etc."`

`"%(itemNumber)d %(itemColor)s" % {"itemNumber":123, "itemColor":"blue"}` returns `"123 blue"`

% codes to format strings:

Code	Description
`%s`	String or any object
`%r`	Equivalent to %s but uses repr()
`%c`	Character
`%d`	Decimal integer
`%i`	Integer
`%u`	Unsigned integer
`%o`	Octal integer
`%x, %X`	Hexadecimal integer
`%e, %E`	Float exponent
`%f, %F`	Float
`%g, %G`	%e or %f float
`%%`	'%' literal

Most common methods for strings

The following table summarizes the most common methods for strings. For instance, if `s` is a string, `s.lower()` returns s converted to lower cases. All operations on sequences are authorized.

Code	Description
`s.capitalize()`	Returns a copy of s in upper cases
`s.center(width)`	Returns a copy of s centered on a string of `width` characters
`s.count(sub[,start[,end]])`	Returns the number of occurrences of `sub` in `s`
`s.encode([encoding[,errors]])`	Returns the encoded version of s
`s.endswith(suffix[,start[,end]])`	Returns TRUE if s ends with a suffix
`s.expandtabs([tabsize])`	Returns a copy of s where all tabulations are replaced with tabsize spaces
`s.find(sub[,start[,end]])`	Returns the first index of s where sub was found
`s.index(sub[,start[,end]])`	Same as 'find' but returns an error sub is not found
`s.isalnum()`	Returns TRUE if all characters of s are alpha

	numeric
`s.isalpha()`	Returns TRUE if all characters of s are alpha
`s.isdigit()`	Returns TRUE if all characters of s are numeric
`s.islower()`	Returns TRUE if s is in lower case.
`s.isspace()`	Returns TRUE if s only contains spaces
`s.istitle()`	Returns TRUE if each word in s starts with an upper case
`s.isupper()`	Returns TRUE if all characters in s are in upper case
`s.join(seq)`	Returns the concatenation of strings of the sequence seq separated by s
`s.ljust(width)`	Returns a left justified copy of s with a maximum length of width characters
`s.lower()`	Returns a lower case copy of s
`s.lstrip()`	Returns a copy of s, trimming all spaces on the left.
`s.replace(old, new[, maxsplit])`	Replaces old with new in s
`s.rfind(sub[,start[,end]])`	Returns the last index of s where sub was found
`s.rindex(sub[,start[,end]])`	Same as rfind but returns an error if not found
`s.rjust(width)`	Returns a right-justified copy of s with a maximum length of width characters
`s.rstrip()`	Returns a copy of s, trimming all spaces on the right
`s.split([sep[,maxsplit]])`	Returns a list of words from s, using sep as a separator
`s.splitlines([keepends])`	Returns the list of lines from s
`s.startswith(prefix[,start[,end]])`	Returns TRUE if s starts with prefix
`s.strip()`	Returns a copy of s trimming all spaces on the left and right
`s.swapcase()`	Returns a copy of s with uppercases converted to lowercases and vice versa
`s.title()`	Returns a copy of s where all words start with an uppercase.
`s.translate(table[,deletechars])`	Translates s according to table
`s.upper()`	Returns an uppercase copy of s

Lists

Lists are arrays of modifiable references to objects, accessed with an index.

A list is a series of values separated by commas and enclosed in brackets.

- `[]` is an empty list
- `[0, 1, 2, 3, 4, 5]` is a list of 6 elements indexed from 0 to 5
- `mylist = ['john', 1, ['albert', 'collin']]` is a list of 3 elements where the 2 index (third element) is also a list

`mylist[2]` returns `['albert', 'collin']`

`mylist[2][1]` returns `'collin'`

Some list functions

Method	Description
`mylist.append(x)`	Adds an element at the end of the list
`mylist.sort([function])`	Sorts the list with the optional [function] comparison function
`mylist.reverse()`	Reverses the list (from last to first)
`mylist.index(x)`	Seeks the index x
`mylist.insert(i, x)`	Inserts x at index i
`mylist.count(x)`	Returns the number of occurrences of x in the list
`mylist.remove(x)`	Deletes the first occurrence of x in the list
`mylist.pop([i])`	Deletes and return the last element in the list or the element at index i

Dictionaries

Dictionaries are arrays of objects indexed on a key (string value) and not by an index.

A dictionary is accessed with a tuple **key:value** separated by commas and in brackets.

- `{}` is an empty dictionary
- `{'P1':'Watch', 'P2': 'Birds', 'P3':'Horses'}` is a dictionary with 3 elements with the following keys: P1, P2 and P3
- `adict = {'FR_US':{'Bonjour':'Hello', 'Au revoir':'Goodbye'}, 'US_FR':{'Hello': 'Bonjour','Goodbye':'Au Revoir'}}` is a dictionary that contains other dictionaries. To translate 'Hello' in French:: `adict['US_FR']['Hello']`

Some methods to handle dictionaries

Method	Description
`adict.has_key(k)`	Returns TRUE (1) if the key `k` exists in the dictionary
`adict.keys()`	Returns the list of dictionary keys
`adict.values()`	Returns a list of dictionary values
`adict.items()`	Returns a list of tuples (key, value) for each element of the dictionary
`adict.clear()`	Deletes all elements of `adict`
`adict.copy()`	Returns a copy of `adict`
`dic1.update(dic2)`	Update the dictionary `dic1` with the values of `dic2` based on the values of the keys
`adict.get(k[,default])`	Equivalent to `adict[k]` but returns `default` if `k` cannot be found.
`adict.popitem()`	Retrieves an element and deletes it from the dictionary

Tuples

Tuples are **non modifiable** object arrays parsed with an index.

A tuple is handled as a series of values separated by commas and within brackets.

- `()` is an empty tuple
- `(0,1,2,3)` is a 4 elements tuple, indexed from 0 to 3
- `tuple = (0, (1, 2), (4,5,6))` is a tuple that contains other tuples. `tuple[1][0]` returns `1`

Operations on sequences are available for the tuples.

Sequences

Sequences can be strings, lists, tuples or dictionaries.

The most common operations are described below:

All sequences

Operation	Description
`X in S, X not in S`	Belonging
`for X in S:`	Iteration
`S+S`	Concatenation
`S*N, N*S`	Repeating

`S[i]`	Indexing
`S[i:j]`	Array indexing
`len(S)`	Length (Size)
`iter(S)`	Iterating object
`min(S)`	Smallest element
`max(S)`	Largest element

Modifiable lists

Operation	Description
`S[i]=X`	Assignment/modification with index i
`S[i:j]=S2`	Assign S2 to an array
`del S[i]`	Delete the element i
`del S[i:j]`	Delete the array from i to j

dictionaries

Opération	Description
`D[k]`	Key indexing
`D[k] = X`	Assignment / modification with the key
`del D[k]`	Delete the element at key k
`len(D)`	Number of keys in D

Examples

```
>>> s='ABCDEFGH'
>>>s[0]
'A'
>>>s[0:2]
'AB'
>>>s[:2]
'AB'
>>>s[-3:-1]
'FG'
```

```
>>>s[-1:]
'H'
```

Files

File objects are handled with built-in functions. The `open` method is used to open a file for reading or writing operations.

The following table shows the most common methods used for files

F Operation	Description
`f = open(filename [, mode='r'])`	Opens a file in the proper mode: mode : `'r'` : Read `'w'` : Write. Create the file if it does not exist `'a'` : Append. `'+'` : (added to the previous modes - example 'a+') opens the file for updates `'b'` : (added to the previous modes - example 'rb') open the file in binary mode
`f.close()`	Closes the f file
`f.fileno()`	Returns descriptor of the f file
`f.flush()`	Empties the internal buffer of f
`f.isatty()`	Returns true if f is a TTY
`f.read([size])`	Reads a maximum of size bytes from the file and returns a string
`f.readline()`	Reads a line from f
`f.readlines()`	Reads until the end of file (EOF) and returns a list of lines
`f.xreadlines()`	Returns a sequence without reading until the end of file (preferred to readlines()
`f.seek(offset[, whence=0])`	Sets the file's current position to 'offset' bytes from 'whence': 0: from the beginning of the file 1: from the current location 2: from the end of the file
`f.tell()`	Returns the current location in the file
`f.write(str)`	Writes str into the file

`f.writelines(list)`	Writes a list of strings in the file

Syntax

Identifiers

Identifiers can be named as follows:

```
    (letter | "_")  (letter | number | "_")*
```

Note: Identifiers, keywords and attributes are case-sensitive.

Special Forms:

`_ident` , `__ident__` and `__ident` have a particular significance. Please refer to the Jython documentation.

The Assignments

All the following assignment forms are valid:

```
x = v
x1 = x2 = v
x1, x2 = v1, v2
x1, x2, ..., xn = v1, v2, ..., vn
(x1, x2, ..., xn) = (v1, v2, ..., vn)
[x1, x2, ..., xn] = [v1, v2, ..., vn]
```

The following special assignment forms are also valid:

`x += y` is equivalent to `x = x + y`

`x *= y` is equivalent to `x = x * y`

`x /= y` is equivalent to `x = x / y`

`x -= y` is equivalent to `x = x - y`

`x %= y` is equivalent to `x = x % y`

`x &= y` is equivalent to `x = x & y`

`x ^= y` is equivalent to `x = x ^ y`

`x **= y` is equivalent to `x = x ** y`

`x |= y` is equivalent to `x = x | y`

Expressions

```
expression
function([value1, arg_name=value2, ...])
object.method([value1, arg_name=value2, ...])
```

A function call always assigns values to all parameters mentioned in the parameter list, either from position arguments, from keyword arguments, or from default values.

Instructions

The break Statement

```
break
```

Terminates the nearest enclosing `while` or `for` loop, skipping the optional `else` clause if the loop has one.

The class Statement

```
class class_name [(super_class1 [,super_class2]*)]:
 instructions
```

Creates a new `class_name` class, that can then be instantiated to create objects.

Examples

```
class c:
  def __init__(self, name, pos):
    self.name = name
    self.pos = pos
  def showcol(self):
    print "Name : %s; Position :%d" % (self.name, self.pos)

col2 = c("CUSTNAME", "2")
col2.showcol()
```

Returns:

```
Name : CUSTNAME, Position :2
```

The continue Statement

```
continue
```

Continues with the next cycle of the nearest enclosing `while` or `for` loop.

The def Statement

```
def func_name ([arg, arg=value, ... *arg, **arg]):
  instructions
```

Defines a func_name function.

The parameters are passed by value and can be defined as follows:

Parameter	Description
`arg`	Parameter passed by value
`arg=value`	Parameter with a default value (if arg is not passed during the call)
`*arg`	Free parameters. arg takes the value of a tuple of all parameters.
`**arg`	Free parameters. arg takes the dictionary value for each parameter.

Examples

Function with default parameters:

```
def my_function(x, y=3):
        print x+y

my_function(3,5)
```

Displays `8`

```
my_function(3)
```

Displays `6`

Function with free parameters:

```
def my_print(*x):
     for s in x:
             print s,

my_print('a','b','c','d')
```

Displays `a b c d`

The del Statement

```
del x
del x[i]
del x[i:j]
del x.attribute
```

Deletes names, references, slicings, attributes

The exec Statement

```
exec x [in globals [,locals]]
```

Executes `x` in the indicated namespace. By default, `x` is executed in the current namespace.

`x` can be a string, a file object or a function object.

The for Statement

```
for x in sequence:
  instructions
[else:
  instructions]
```

Used to iterate over the elements of a sequence. Each item of *sequence* is assigned to *x* using the standard rules for assignments, and then the code is executed. When the items are exhausted (which is immediately when the sequence is empty), the code in the else clause, if present, is executed, and the loop terminates.

Examples

Loop from 0 to 3:

```
for i in range(4):
  print i
```

Loop from 2 to 5:

```
for i in range(2, 6):
  print i
```

Loop from 2 to 10 by 2:

```
for i in range(2, 11, 2):
  print i
```

Loop on all elements of a list:

```
l = [ 'a', 'b', 'c', 'd']
for x in l:
  print x
```

The from Statement

```
from module import name1 [as othername1] [, name2]*
from module import *
```

Imports the names of a module into the current namespace.

Examples

Display the directories of c:\

```
from os import listdir as directorylist
dir_list = directorylist('c:/')
print dir_list
```

The global Statement

```
global name1 [, name2]
```

The *`global`* statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as global identifiers. It would be impossible to assign to a global variable without global, although free variables may refer to global identifiers without being declared global. *`name1`* is a reference to the global variable `name1`.

The if Statement

```
if condition:
  instructions
[elif condition:
  instructions]*
[else:
  instructions]
```

Used for conditional execution: **If , else if ... , else**

Examples

```
x = 2
y = 4
if x == y :
  print "Waooo"
elif x*2 == y:
  print "Ok"
else:
  print "???"
```

The import Statement

```
import module1 [as name1] [, module2]*
```

Imports a module or package so that it becomes accessible. The module contains names that are accessible through their `module_name.name` qualifier.

Examples

Display the directories in c:\

```
import os
dir_list = os.listdir('c:/')
print dir_list
```

Using the JDBC classes: run a SQL query and display the results

```
import java.sql as jsql
import java.lang as lang
driver, url, user, passwd = (
     "oracle.jdbc.driver.OracleDriver",
     "jdbc:oracle:thin:@pluton:1521:pluton",
     "user",
     "pass")
lang.Class.forName(driver)
c = jsql.DriverManager.getConnection(url,user,passwd)
s = c.createStatement()
sql_stmt = "select * from user_tables"
print "executing " , sql_stmt
rs = s.executeQuery(sql_stmt)
while (rs.next()):
        print rs.getString("TABLE_NAME"), rs.getString("OWNER")
c.close()
```

The pass Statement

```
pass
```

This is a null operation -- when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically but no code needs to be executed.

The print Statement

```
print [value [, value]* [,]]
print >> file_object [, value [, value]* [,]]
```

Evaluates each expression in turn and writes the resulting object to standard output (stdout) or to the `file_object` file. A "\n" character is written at the end, unless the print statement ends with a comma.

The raise Statement

```
raise [exception [, value]]:
```

Raises the `exception` exception with the optional `value` value.

If the `exception` object is a class, it becomes the type of the exception. `Value` is used to determine the exception value: If it is an instance of the class, the instance becomes the exception value. If `value` is a tuple, it is used as the argument list for the class constructor; if it is None, an empty argument list is used, and any other object is treated as a single argument to the constructor. The instance so created by calling the constructor is used as the exception value.

If no expressions are present, `raise` re-raises the last expression that was active in the current scope. If no exception has been active in the current scope, an exception is raised that indicates that this is the error.

The return Instruction

```
return [expression]
```

Leaves the current function call with *`expression`*

(or None) as the return value.

The try Statement

```
try :
 suite1
[except [exception [, value]]:
 suite2]*
[else :
 suite3]

try :
 suite1
finally :
 suite2
```

Specifies exception handlers and/or cleanup code for a group of statements. There are two forms of try statement:

1st form: When no exception occurs in `suite1`, no exception handler is executed. When an exception occurs in `suite1`, a search for an `exception` is started. If it is found, `suite2` is executed. Otherwise, `suite3` is executed. If the exception has a value, it is assigned to the triggered instance.

2nd form: When no exception occurs in `suite1`, no exception handler is executed. When an exception occurs in `suite1`, `suite2` is executed in all cases and the exception is raised.

Examples

Open a file and close it in all cases:

```
f = open('c:/my_file.txt', 'w')
try:
  f.write('Hello world')
  ...
finally:
  f.close()
```

Open a non existing file and trap the exception:

```
try:
  f = open('inexisting_file.txt', 'r')
  f.close()
except IOError, v:
  print 'IO Error detected: ', v
else:
  print 'Other Error'
```

The while Statement

```
while condition:
  instructions
[else:
  instructions]
```

Used for repeated execution as long as an expression is true. A `break` statement executed in the first suite terminates the loop without executing the else clause's suite and before *`condition`* is false.

Examples

Display i from 0 to 8

```
i = 0
while i < 9:
  print i
  i+=1
```

Modules

Internal modules must be imported with the import statement. There are many modules.

An exhaustive list of modules and their reference documentation is available at http://www.jython.org

The following list gives a brief overview of the most commonly used modules:

Modules	Description	Some methods
None	Standard Functions	`apply, callable, chr, cmp, coerce,`

		compile, complex, delattr, eval, execfile, filter, getattr, globals, hasattr, hash, hex, id, input, intern, isinstance, issubclass, list locals, ord, pow, range, reduce, reload, repr, round, setattr, slice, str, tuple, type, vars, xrange
sys	Contains functions relative to the interpreter. They give access to the environment components (command line, standard I/O)	argv, builtin_module_names, check_interval, exc_info, exit, exitfunc, getrecursionlimit, getrefcount, maxint, modules, path, platform, setcheckinterval, setdefaultencoding, setprofile, setrecursionlimit, settrace, stderr, stdin, stdout, version
os	Interface with the operating system, independently of the platform.	_exit, altsep, chdir, chmod, curdir, environ, error, execv, fork, getcwd, getpid, kill, linesep, listdir, makedirs, name, pardir, path, pathsep, pipe, removedirs, renames, sep, system, times, wait, waitpid
math	Math Functions	acos, asin, atan, atan2, ceil, cos, cosh, e, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pi, pow, sin, sinh, sqrt, tan, tanh
time	Date and time functions	altzone, daylight, gmtime, localtime, sleep, strftime, time
re	Regular expression functions	compile, I, L, M, S, U, X, match, search, split, sub, subn, findall, escape
socket	TCP/IP Socket support	See reference documentation
cgi	CGI script support	See reference documentation
urllib, urllib2	Web page search. Supports the following URLs: http, ftp, gopher and file	See reference documentation
ftplib	FTP protocol support	See reference documentation
httplib, nntplib	Http and nntp support	See reference documentation
poplib, imaplib, smtplib	POP, IMAP and SMTP protocol support	See reference documentation
telnetlib, gopherlib	Telnet, gopher support	See reference documentation

Using Jython in Oracle Data Integrator

Using the Jython interpreter

Jython programs can be interpreted for test purposes outside of Data Integrator, using the standard Jython interpreter.

Start the Jython interpreter

1. Start an OS prompt (console)
2. Go to the `/bin` directory.
3. Key in: `jython`
4. The interpreter is launched

Exiting the Jython interpreter

1. Hit Ctrl+Z (^Z), then Enter
2. You exit the interpreter

Running Jython scripts

1. Go to the `/bin` directory.
2. Type in: `jython <script_path.py>`
3. The script is executed

Using Jython in the procedures

All Jython programs can be called from a **procedure** or a **Knowledge Module**.

Create a procedure that calls Jython

1. In **Designer**, select a **Folder** in your **Project** and insert a new **Procedure**
2. Type the **Name** of the procedure
3. Add a **command line** in the **Detail** tab
4. In the **command** window, type the **Name** for this command
5. In the **Command on Target** tab, choose the `Jython` **Technology** from the list
6. In the **Command** text, type the Jython program to be executed, or use the expression editor
7. Click **OK** to apply the changes
8. Click **Apply** to apply the changes in the **procedure** window

9. In the **Execution** tab, click the **Execute** button and follow the execution results in the execution log.

The procedure that was created with this process can be added to a **Package** like any other procedure.

Jython variable persistence in a session

All the Jython variables used are persistent in an execution session.

If a **Procedure** `TRT1` has 3 **command lines** defined as:

Line 1 : `Set value for x`

```
x = 'My Taylor is Rich'
```

Line 2 : `Set value for y`

```
y = 'I wish I could say :'
```

Line 3 : `Write output file`

```
f = open('test.txt', 'w')
f.write('Result : %s %s'  % (y, x))
f.close()
```

After running `TRT1`, the content of the resulting `test.txt` file will be `Result : I wish I could say : My Taylor is Rich`

The Jython variables `x` and `y` have kept their values within the same procedure across several command lines.

Likewise, a process `TRT2` that would be executed after `TRT1` in a package could use the variables `x` and `y` within the same execution session.

Add a Specific Module to the Standard Distribution

It is possible to extend the basic functions of Jython by adding new modules to the default ones.

You can write your own Jython module (please refer to the documentation available at http://www.jython.org) and put this module in the `/lib/scripting/Lib` sub-directory of your Oracle Data Integrator installation directory.

Additional modules in Oracle Data Integrator

For an easier use of Jython within Oracle Data Integrator, the following modules have been added:

snpsftp Module

This module simplifies the use of FTP (File Transfer Protocol) with Jython

It implements the class SnpsFTP

SnpsFTP Class

Constructor / Methode	Description
`SnpsFTP([host [,user [,passwd[, acct [, port]]]]])`	Constructor: creates an ftp object and connects to the `host` FTP server on port number `port` using `user`, `passwd` and `acct` for the authentication.
`connect(host [,port])`	Connects to the FTP host server on port number `port`
`login([user [,passwd [,acct]]])`	Performs authentication against the FTP server.
`setmode(mode)`	Sets the mode to ASCII or BINARY. Possible values are: `'ASCII'` or `'BINARY'`. The default value for transfers is ASCII.
`setpassive(0 | 1)`	Sets the FTP connection in passive (1) or active (0) mode.
`get(src[, dest [, mode]])`	Downloads the file described with its full path `src` (on the FTP server) into the file or directory `dest`. The `mode` can be forced to 'ASCII' or 'BINARY'.
`mget(srcdir, pattern [, destdir [, mode]])`	Downloads a set of files from the directory `srcdir` that matches the filter `pattern` in the directory `destdir` using the `mode` mode.
`put(src [, dest [, mode='' [, blocksize=8192]]])`	Puts the local file `src` in the server file `dest` using the `mode` mode. Uses a bloc transfer size of `blocksize` bytes.
`mput(srcdir, pattern [, destdir [, mode [, blocksize=8192]]])`	Puts several local files from the directory `srcdir` that match the filter `pattern` into the server directory `destdir` using the `mode` mode. Uses a transfer bloc size of `blocksize` octets.
`quit()`	Sends a QUIT command, then closes the connection with the FTP server.
`close()`	Closes the connection with the FTP server.

Examples

Retrieve the `*.txt` files from `/home/odi` on the server `ftp.myserver.com` in the local directory `c:\temp`

```
import snpsftp
ftp = snpsftp.SnpsFTP('ftp.myserver.com', 'mylogin', 'mypasswd')
ftp.setmode('ASCII')
ftp.mget('/home/odi', '*.txt', 'c:/temp')
ftp.close()
```

Put the `*.zip` files from `C:\odi\lib` onto the server `ftp.myserver.com` in the remote directory `/home/odi/lib`

```
import snpsftp
ftp = snpsftp.SnpsFTP('ftp.myserver.com', 'mylogin', 'mypasswd')
ftp.setmode('BINARY')
ftp.mput('C:/odi/lib', '*.zip', '/home/odi/lib')
ftp.close()
```

Read From and Write to a File

The `SRC_AGE_GROUP.txt` file contains records where the columns are separated by `;`. The following example transforms the `SRC_AGE_GROUP.txt` file into a new file `SRC_AGE_GROUP_NEW.txt` using tabulations as separators.

This example uses the `split()` string methods to determine the list of fields separated by `;` and `join()` to rebuild a new string separated by tabulations (`'\t'`).

```
fsrc = open('c:/odi/demo/file/SRC_AGE_GROUP.txt', 'r')
ftrg = open('c:/odi/demo/file/SRC_AGE_GROUP_NEW.txt', 'w')
try:
  for lsrc in fsrc.readlines():
    # get the list of values separated by ;
    valueList = lsrc.split(';')
    # transform this list of values to a string separated by a tab
('\t')
    ltrg = '\t'.join(valueList)
    # write the new string to the target file
    ftrg.write(ltrg)
finally:
  fsrc.close()
  ftrg.close()
```

The method `readlines()` in the above example loads the entire file into memory. It should only be used for small files. For larger files, use the `readline()` method as in the following example. `readline()` will read the lines one by one.:

```
fsrc = open('c:/odi/demo/file/SRC_AGE_GROUP.txt', 'r')
ftrg = open('c:/odi/demo/file/SRC_AGE_GROUP_NEW.txt', 'w')
try:
  lsrc=fsrc.readline()
  while (lsrc):
    valueList = lsrc.split(';')
    ltrg = '\t'.join(valueList)
    ftrg.write(ltrg)
    lsrc=fsrc.readline()
finally:
  fsrc.close()
  ftrg.close()
```

List the contents of a directory

The following example lists the contents of the directory `c:/odi` and writes this list into `c:/temp/listdir.txt`. For each element in the list, the method `os.path.isdir()` checks whether it is a file or a directory

```
import os
ftrg = open('c:/temp/listdir.txt', 'w')
try:
    mydir = 'c:/odi'
    mylist = os.listdir(mydir)
    mylist.sort()
    for dirOrFile in mylist:
        if os.path.isdir(mydir + os.sep + dirOrFile):
            print >> ftrg, 'DIRECTORY: %s' % dirOrFile
        else:
            print >> ftrg, 'FILE: %s' % dirOrFile
finally:
    ftrg.close()
```

Using the Operating System Environment Variables

It can be usefull to retrieve the Operating System environment variables. The following examples show how to retrieve this list:

```
import os
ftrg = open('c:/temp/listenv.txt', 'w')
try:
  envDict = os.environ
  osCurrentDirectory = os.getcwd()
  print >> ftrg, 'Current Directory: %s'  % osCurrentDirectory
  print >> ftrg, '=============================='
  print >> ftrg, 'List of environment variables:'
  print >> ftrg, '=============================='
  for aKey in envDict.keys():
    print >> ftrg, '%s\t= %s' % (aKey, envDict[aKey])
  print >> ftrg, '=============================='
  print >> ftrg, 'Oracle Data Integrator specific environment
variables:'
  print >> ftrg, '=============================='
  for aKey in envDict.keys():
    if aKey.startswith('SNP_'):
      print >> ftrg, '%s\t= %s' % (aKey, envDict[aKey])
```

```
finally:
  ftrg.close()
```

To retrieve the value of the USERNAME environment variable, just write:

```
import os
currentUser = os.environ['USERNAME']
```

Using JDBC

It can be convenient to use JDBC (Java DataBase Connectivity) to connect to a database from Jython. All Java classes in the CLASSPATH can be directly used in Jython. The following example shows how to use the JDBC API to connect to a database, to run a SQL query and write the result into a file.

The reference documentation for Java is available at http://java.sun.com

```
import java.sql as sql
import java.lang as lang
def main():
  driver, url, user, passwd = (
    'oracle.jdbc.driver.OracleDriver',
    'jdbc:oracle:thin:@myserver:1521:mysid',
    'myuser',
    'mypasswd')
  ##### Register Driver
  lang.Class.forName(driver)

  ##### Create a Connection Object
  myCon = sql.DriverManager.getConnection(url, user, passwd)
  f = open('c:/temp/jdbc_res.txt', 'w')
  try:
    ##### Create a Statement
    myStmt = myCon.createStatement()
    ##### Run a Select Query and get a Result Set
    myRs = myStmt.executeQuery("select TABLE_NAME, OWNER from ALL_TABLES
where TABLE_NAME like 'SNP%'")

    ##### Loop over the Result Set and print the result in a file
    while (myRs.next()):
      print >> f , "%s\t%s" %(myRs.getString("TABLE_NAME"),
myRs.getString("OWNER") )
  finally:
    myCon.close()
    f.close()

### Entry Point of the program
```

```
if __name__ == '__main__':
  main()
```

It is possible to combine Jython with odiRef API in the Oracle Data Integrator Procedures, for even more flexibility. Instead of hard-coding the parameters to connect to a database in the program, the getInfo method can be used:

```
import java.sql as sql
import java.lang as lang
def main():
  driver, url, user, passwd = (
    '<%=odiRef.getInfo("DEST_JAVA_DRIVER")%>',
    '<%=odiRef.getInfo("DEST_JAVA_URL")%>',
    '<%=odiRef.getInfo("DEST_USER_NAME")%>',
    '<%=odiRef.getInfo("DEST_PASS")%>')
  ##### Register Driver
  lang.Class.forName(driver)
[...]
```

Using FTP

In some environments, it can be useful to use FTP (File Transfer Protocol) to transfer files between heterogeneous systems. Oracle Data Integrator provides an additional Jython module to further integrate FTP.

The following examples show how to use this module:

Pull the `*.txt` files from `/home/odi` of the server `ftp.myserver.com` into the local directory `c:\temp`

```
import snpsftp
ftp = snpsftp.SnpsFTP('ftp.myserver.com', 'mylogin', 'mypasswd')
try:
  ftp.setmode('ASCII')
  ftp.mget('/home/odi', '*.txt', 'c:/temp')
finally:
  ftp.close()
```

Push the files `*.zip` from `C:\odi\lib` onto `ftp.myserver.com` in the remote directory `/home/odi/lib`

```
import snpsftp
ftp = snpsftp.SnpsFTP('ftp.myserver.com', 'mylogin', 'mypasswd')
try:
  ftp.setmode('BINARY')
  ftp.mput('C:/odi/lib', '*.zip', '/home/odi/lib')
finally:
  ftp.close()
```

Using IP sockets

IP sockets are used to initiate an IP communication between two processes on the network. Jython greatly simplifies the creation of IP servers (waiting for IP packets) or IP clients (sending IP packets).

The following example shows the implementation of a very basic IP server. It waits for data coming from client software, and writes each received packet into the file `c:/temp/socketserver.log`. If a server receives the packet `STOPSERVER`, the server stops:

Server

```
import socket
import time
HOST = ''
PORT = 9191 # Arbitrary port (not recommended)
LOG_FILE = 'c:/temp/sockserver.log'
mySock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
mySock.bind((HOST, PORT))
logfile = open(LOG_FILE, 'w')
try:
  print >> logfile, '*** Server started : %s' % time.strftime('%Y-%m-%d
%H:%M:%S')
  while 1:
    data, addr = mySock.recvfrom(1024)
    print >> logfile, '%s (%s): %s' % (time.strftime('%Y-%m-%d
%H:%M:%S'), addr, data)
    if data == 'STOPSERVER':
      print >> logfile, '*** Server shutdown at %s by %s' %
(time.strftime('%Y-%m-%d %H:%M:%S'), addr)
      break
finally:
  logfile.close()
```

Client

The following example can be used ot test the above server. It sends two packets before asking the server to stop.

```
import socket
import sys
PORT = 9191 # Same port as the server
HOST = 'SERVER_IP_ADDRESS'
mySock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
mySock.sendto('Hello World !', (HOST, PORT))
mySock.sendto('Do U hear me?', (HOST, PORT))
mySock.sendto('STOPSERVER', (HOST, PORT))
```