Mastering Oracle+Python, Part 9: Jython and IronPython - JDBC and ODP.NET in Python
by Przemyslaw Piotrowski

This installment will cover the two most popular development environments for Python - Java and .NET - as well as Python's native implementations of these platforms.

Published December 2011

See series TOC

A successful programming language will always make it into top development platforms. That was certainly the case for Python and the world's top two programming environments: Java and Microsoft .NET.

Although Python is frequently dubbed "the glue language" for its ability to quickly assemble diverse software components, a need for native implementations emerged. In 1997, MIT grad student Jim Hugunin started The Jython Project, an implementation of Python in Java that allows the remarkable result of running a dynamic, high-level language on top of the efficient Java Virtual Machine (JVM). Since then, Jython has attracted numerous committers. Oracle WebLogic Server users may already be familiar with the tool, which is included in the WebLogic software bundle for scripting purposes. At the time of this writing, the current release of Jython is 2.5.2.

Hugunin didn't stop there. At one point, it was considered impossible to implement a dynamic language on top of the Common Language Runtime (CLR) at the core of .NET. However, in 2004, Hugunin - then at Microsoft - worked with a small team to create the IronPython project, which not only contradicted the conventional wisdom but also was shown to outperform the C implementation in some cases. Although the project is no longer led by Microsoft, the project is still active and IronPython 2.7.1 was released in October 2011.

In this article you'll learn about using these Python dialects with Oracle Database and its ubiquitous access drivers: JDBC and ODP.NET. Having gained basic knowledge on connectivity, querying practices, and driver specifics, you'll be able to plug Python into whatever environment you like, achieving extreme development speeds with the benefits of the JVM and CLR.

Note that you are not limited to using just the command line any more. There are multiple IDEs available on the market, including NetBeans IDE, Eclipse with the PyDev extension for Jython, and SharpDevelop for IronPython. Also, since this is plain-vanilla Python, you can surely take advantage of the dynamic interpreter, which lets you experiment with libraries and modules.

Jython and JDBC
Installing Jython is absolutely straightforward with a bundled JAR executable obtainable from www.jython.org. All you need to do after downloading jython_installer_2.5.2.jar is to execute the archive:

```
C:\java -jar jython_installer-2.5.2.jar
```

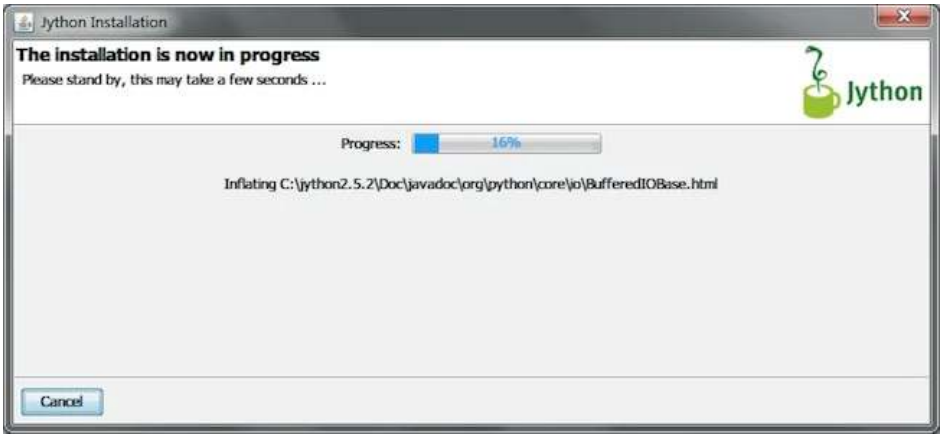


Figure 1 Jython installation screen

Under the destination Jython directory, you'll find the jython.jar file to launch the interpreter. In addition to CLASSPATH, another environment variable, JYTHONPATH, is respected during the run. One of the most useful options is the ability to inspect code after being run (-i switch) - allowing you to look at various variables, re-declare parts of code, and re-run.

```
usage: jython [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments:
-c cmd   : program passed in as string (terminates option list)
-Dprop=v : Set the property `prop' to value `v'
-C codec : Use a different codec when reading from the console.
-h       : print this help message and exit (also --help)
-i       : inspect interactively after running script
           and force prompts, even if stdin does not appear to be a terminal
-jar jar : program read from __run__.py in jar file
-m mod   : run library module as a script (terminates option list)
-Q arg   : division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew
-S       : don't imply 'import site' on initialization
-u       : unbuffered binary stdout and stderr
-v       : verbose (trace import statements)
-V       : print the Python version number and exit (also --version)
-W arg   : warning control (arg is action:message:category:module:lineno)
file     : program read from script file
-        : program read from stdin (default; interactive mode if a tty)
arg ...  : arguments passed to program in sys.argv[1:]
Other environment variables:
```

```
JYTHONPATH: ';'-separated list of directories prefixed to the default module
            search path.  The result is sys.path.
```

While the standard JDBC API is based on thejava.sql package, Oracle ships its own modules for extending basic functionality and providing mappings to Oracle data types: oracle.jdbc and oracle.sql. All of them will be covered here, but should you need any further information, consult the *Oracle Database JDBC Developer's Guide*.

For client connectivity, you are free to choose from the native Java implementation (JDBC Thin driver -fully crossplatform, wrapped in a single JAR archive) or the Oracle Call Interface (OCI) - the very same client you would use connecting from C. (Note that OCI has a dependency on the Oracle Client or Oracle Instant Client.)

There is no initial setup involved, a text file with a single line as below implements basic JDBC connectivity using Thin driver. All you need to provide is to make sure ojdbc6.jar is in the CLASSPATH or JYTHONPATH so that the connection driver could be resolved.

```
conn = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "hr", "hr")
```

Querying the database does not get much harder than that. Elementary Java knowledge is sufficient for comfortable work with Jython. Everything is just a lot less verbose.

```
from java.sql import DriverManager

conn = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "hr", "hr")
stmt = conn.createStatement()
rs = stmt.executeQuery("select * from employees order by last_name")

emp = {}
while rs.next():
  emp[rs.getString(4)] = {
       'name': "%s, %s" % (rs.getString(3), rs.getString(2)),
       'email': rs.getString(4),
       'salary': rs.getInt(8)
  }
  print "%s, %s (%s)" % (rs.getString(3), rs.getString(2), rs.getString(4))

rs.close()
stmt.close()
conn.close()
```

Not only did we get the advantage of having concise, quick implementation but what we can also run this script interactively, as you would do with Python. Figure 2. illustrates how we can break out from regular program execution to Jython shell when we can introspect existing objects and perform further actions on them - with either Python or native Java libraries.

Figure 2 Running scripts in standard vs. interactive mode

If you were to use OCI instead of Thin driver, it would require a change in the connection string provided to DriverManager like below. (Obvious advantages behind OCI are native features like connection pooling, Transparent Application Failover, and clientside result cache.)

```
conn = DriverManager.getConnection("jdbc:oracle:oci:@localhost:1521:XE", "hr", "hr")
```

OracleDataSource brings support for native data types, performance improvements, and better compatibility. Since with Jython you can use pretty much any library you would use with Java, oracle.jdbc.pool.OracleDataSource is no exception.
This example supplements regular JDBC with connection exception handling, catching java.sql.SQLException errors while establishing a new connection to the Oracle data source.

```
import sys
import java.sql.SQLException
from oracle.jdbc.pool import OracleDataSource

cs = "jdbc:oracle:thin:@localhost:1521:XE"

ods = OracleDataSource()
ods.setURL(cs)
ods.setUser("hr")
ods.setPassword("hr")
try:
```

```
  conn = ods.getConnection()
except java.sql.SQLException, e:
  print "Problem connecting to \"%s\"" % cs
  sys.exit()

stmt = conn.createStatement()
rs = stmt.executeQuery("select * from departments order by 2")

while rs.next():
  print rs.getString(2)

rs.close()
stmt.close()
conn.close()
```

Jython is a great choice for rapid prototyping of Java applications and gluing together distinct software components. But it is not just blazing development speed that sets Jython apart - dynamic introspection, concise syntax, and rich programming language are only some basic features. The real value comes from the wealth of libraries available for both platforms, no matter what language they originate from. You can use Jython with your JEE infrastructure and deploy Django on your Oracle WebLogic Server, mix Jython with Jasper Reports for an agile BI solution, or take advantage of one of the over 17,000 packages available at Python Package Index (PyPI) in your next Java project.

IronPython and ODP.NET
Python's philosophy of "batteries included" gets a whole new meaning with IronPython, which gives you access to the extensive .NET library. From *Accessibility* and various *Microsoft* containers up to vast *System* namespace, .NET-based IronPython gives you broad coding possibilities in your favorite programming language. Since our series is strictly database-oriented, this article focuses on Windows Presentation Foundation and Oracle Data Provider libraries.

Like with Jython, the most important part is IronPython executable. It comes in the form of four binaries:

ipy.exe
ipy64.exe
ipyw.exe
ipyw64.exe
where "w" suffix denotes "windowed" applications and "64" is meant for running on 64-bit architectures. Even though IronPython is targeted at Microsoft Windows platform, users of the Linux implementation of .NET - Mono - could also leverage some of its functionality.

```
Usage: ipy.exe Usage: ipy [options] [file.py|- [arguments]]

Options:
 -3                          Warn about Python 3.x incompatibilities
 -c cmd                      Program passed in as string (terminates option list)
 -D                          Enable application debugging
 -E                          Ignore environment variables
 -h                          Display usage
 -i                          Inspect interactively after running script
 -m module                   run library module as a script
 -O                          generate optimized code
 -OO                         remove doc strings and apply -O optimizations
 -Q arg                      Division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew
 -s                          Don't add user site directory to sys.path
 -S                          Don't imply 'import site' on initialization
 -t                          Issue warnings about inconsistent tab usage
 -tt                         Issue errors for inconsistent tab usage
 -u                          Unbuffered stdout & stderr
 -v                          Verbose (trace import statements) (also PYTHONVERBOSE=x)
 -V                          Print the version number and exit
 -W arg                      Warning control (arg is action:message:category:module:lineno)
 -x                          Skip first line of the source
 -X:AutoIndent               Enable auto-indenting in the REPL loop
 -X:ColorfulConsole          Enable ColorfulConsole
 -X:CompilationThreshold  The number of iterations before the interpreter starts compiling
 -X:Debug                    Enable application debugging (preferred over -D)
 -X:EnableProfiler           Enables profiling support in the compiler
 -X:ExceptionDetail          Enable ExceptionDetail mode
 -X:Frames                   Enable basic sys._getframe support
 -X:FullFrames               Enable sys._getframe with access to locals
 -X:GCStress                 Specifies the GC stress level (the generation to collect each statement)
 -X:LightweightScopes        Generate optimized scopes that can be garbage collected
 -X:MaxRecursion             Set the maximum recursion level
 -X:MTA                      Run in multithreaded apartment
 -X:NoAdaptiveCompilation Disable adaptive compilation
 -X:NoDebug <regex>          Provides a regular expression of files which should not be emitted in debug mode
 -X:PassExceptions           Do not catch exceptions that are unhandled by script code
 -X:PrivateBinding           Enable binding to private members
 -X:Python30                 Enable available Python 3.0 features
 -X:ShowClrExceptions        Display CLS Exception information
 -X:TabCompletion            Enable TabCompletion mode
 -X:Tracing                  Enable support for tracing all methods even before sys.settrace is called

Environment variables:
 IRONPYTHONPATH       Path to search for module
 IRONPYTHONSTARTUP Startup module
```

IronPython's interactive shell is much richer than Jython's. It supports tab auto-completion, syntax coloring, and tighter integration with the framework, and comes with advanced debugging and profiling mechanisms.

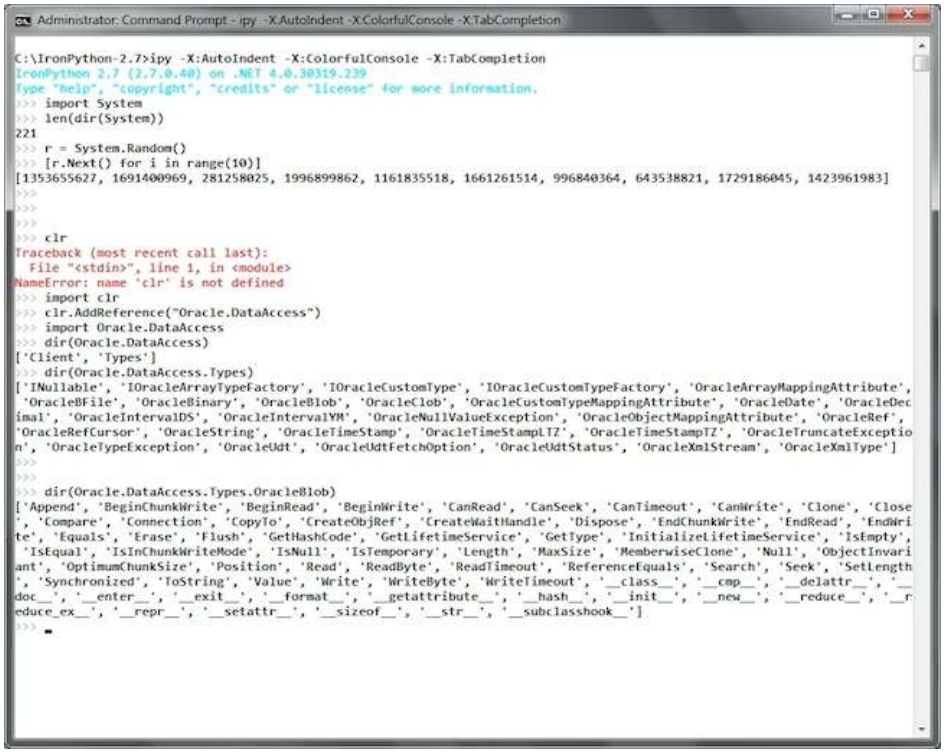


Figure 3 IronPython interactive shell with dynamic introspection, highlighting, and tab-completion while adding a reference to Oracle.DataAccess library

Loading .NET assemblies is possible in multiple ways; the most two common and recommended being:

clr.AddReference - Accepts direct object reference (e.g. “System.Random”)
clr.AddReferenceToFileAndPath - Uses direct path to resolve library reference (e.g. “D:/oraclexe/app/oracle/product/11.2.0/server/odp.net/bin/4/Oracle.DataAccess.dll”)
CLR, the virtual machine for .NET, is referenced from IronPython as clr module. All interactivity between IronPython and .NET assemblies is handled through this library.

Adding an external assembly from ipy interpreter is as easy as:

```
>>> import clr
>>> clr.AddReference(“Oracle.DataAccess”)
>>> import Oracle.DataAccess
```

To make sure the correct version is imported we can reference the file directly:

```
>>> import clr
>>>
clr.AddReferenceToFileAndPath(“”D:/oraclexe/app/oracle/product/11.2.0/server/odp.net/bin/4/Oracle.DataAccess.dll””
>>> import Oracle.DataAccess
```

The most basic example requires only one object to be imported from Oracle.DataAccess.Client namespace. The code itself resembles the one we used for Jython previously. The following example goes through all departments and prints them out.

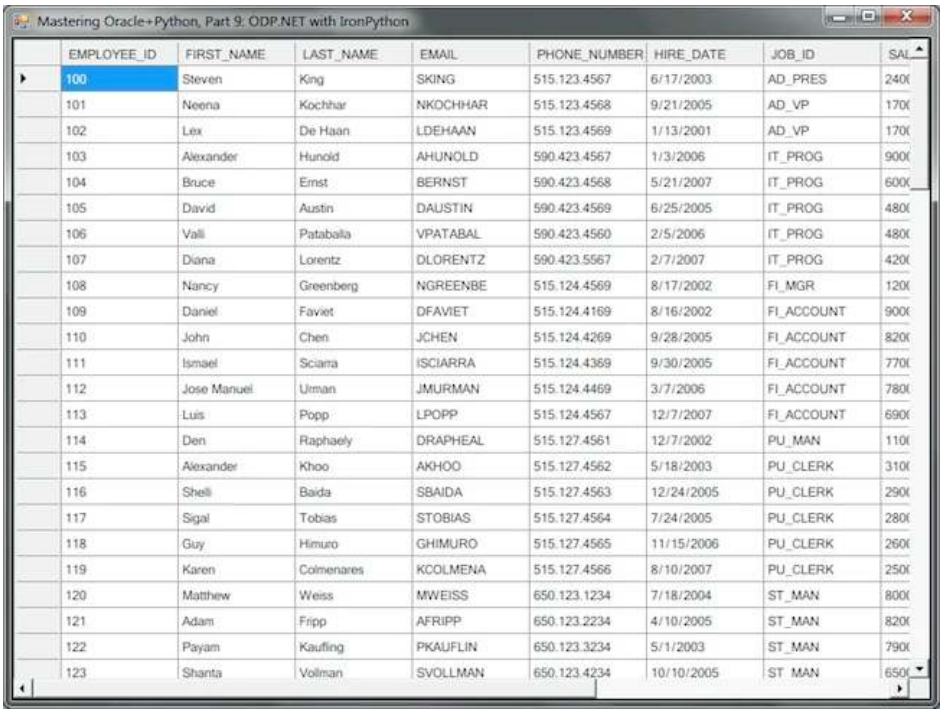
```
import clr
clr.AddReference("Oracle.DataAccess")
from Oracle.DataAccess.Client import OracleConnection
conn = OracleConnection("User Id=hr;Password=hr;Data Source=XE")
conn.Open()
cmd = conn.CreateCommand()
cmd.CommandText = "select * from departments order by 1"

reader = cmd.ExecuteReader()
if reader.HasRows:
  while reader.Read():
        dept = {
        'id': reader.GetDecimal(reader.GetOrdinal("DEPARTMENT_ID")),
        'name': reader.GetString(reader.GetOrdinal("DEPARTMENT_NAME"))
        }
        print "Department #%s: %s" % (dept['id'], dept['name'])

conn.Close()
```

Using .NET and not creating a windowed application doesn’t seem quite right, so let's try to harness some of the GUI controls and present the effects on a grid component. For this purpose, we’ll go through two important objects in .NET library when interfacing a database: *DataSet* and *OracleDataAdapter*. To render windows and lay out components we’ll make use of *System.Windows.Forms* and *System.Drawing* libraries.

```
import clr
clr.AddReference("Oracle.DataAccess")
clr.AddReference("System.Data")
clr.AddReference("System.Drawing")
clr.AddReference("System.Windows.Forms")

from System.Drawing import Size
from System.Data import DataSet
```

```
from Oracle.DataAccess.Client import OracleConnection, OracleDataAdapter
from System.Windows.Forms import Application, DataGridView, DockStyle, Form

form = Form(Size = Size(800, 600))
form.Text = "Mastering Oracle+Python, Part 9: ODP.NET with IronPython"

conn = OracleConnection("User Id=hr;Password=hr;Data Source=XE")
conn.Open()
oda = OracleDataAdapter()
ds = DataSet()

cmd = conn.CreateCommand()
cmd.CommandText = "select * from employees order by 1"
oda.SelectCommand = cmd
oda.Fill(ds)
conn.Close()

grid = DataGridView(DataSource = ds.Tables[0], AutoSize = True, Dock = DockStyle.Fill)
form.Controls.Add(grid)
Application.Run(form)
```

The resulting application would look similar to the one in Figure 4.



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SAL
100	Steven	King	SKING	515.123.4567	6/17/2003	AD_PRES	240(
101	Neena	Kochhar	NKOCHHAR	515.123.4568	9/21/2005	AD_VP	170(
102	Lex	De Haan	LDEHAAN	515.123.4569	1/13/2001	AD_VP	170(
103	Alexander	Hunold	AHUNOLD	590.423.4567	1/3/2006	IT_PROG	900(
104	Bruce	Ernst	BERNST	590.423.4568	5/21/2007	IT_PROG	600(
105	David	Austin	DAUSTIN	590.423.4569	6/25/2005	IT_PROG	480(
106	Valli	Pataballa	VPATABAL	590.423.4560	2/5/2006	IT_PROG	480(
107	Diana	Lorentz	DLORENTZ	590.423.5567	2/7/2007	IT_PROG	420(
108	Nancy	Greenberg	NGREENBE	515.124.4569	8/17/2002	FI_MGR	120(
109	Daniel	Faviet	DFAVIET	515.124.4169	8/16/2002	FI_ACCOUNT	900(
110	John	Chen	JCHEN	515.124.4269	9/28/2005	FI_ACCOUNT	820(
111	Ismael	Sciarra	ISCIARRA	515.124.4369	9/30/2005	FI_ACCOUNT	770(
112	Jose Manuel	Urman	JMURMAN	515.124.4469	3/7/2006	FI_ACCOUNT	780(
113	Luis	Popp	LPOPP	515.124.4567	12/7/2007	FI_ACCOUNT	690(
114	Den	Raphaely	DRAPHEAL	515.127.4561	12/7/2002	PU_MAN	110(
115	Alexander	Khoo	AKHOO	515.127.4562	5/18/2003	PU_CLERK	310(
116	Shelli	Baida	SBAIDA	515.127.4563	12/24/2005	PU_CLERK	290(
117	Sigal	Tobias	STOBIAS	515.127.4564	7/24/2005	PU_CLERK	280(
118	Guy	Himuro	GHIMURO	515.127.4565	11/15/2006	PU_CLERK	260(
119	Karen	Colmenares	KCOLMENA	515.127.4566	8/10/2007	PU_CLERK	250(
120	Matthew	Weiss	MWEISS	650.123.1234	7/18/2004	ST_MAN	800(
121	Adam	Fripp	AFRIPP	650.123.2234	4/10/2005	ST_MAN	820(
122	Payam	Kaufling	PKAUFLIN	650.123.3234	5/1/2003	ST_MAN	790(
123	Shanta	Vollman	SVOLLMAN	650.123.4234	10/10/2005	ST_MAN	650(

Figure 4 IronPython application interacting with Oracle Database XE through OracleDataAdapter and DataGridView.

As we have just demonstrated, interacting with .NET framework from IronPython is as smooth as working with a native CLR language like C# or Visual Basic.

Packaging IronPython programs into .exe binaries is very smooth with the bundled *ipy.exe* tool and pyc.py compilation script (located under the Tools\Scripts directory). To turn a script into executable binary you just provide the name and type of the output as below.

```
C:\IronPython-2.7>ipy Tools\Scripts\pyc.py /out:GridView /target:exe /main:gridview.py
Input Files:
Output:
       GridView
Target:
       ConsoleApplication
Platform:
       ILOnly
Machine:
       I386
Compiling...
Saved to GridView
```

By the time compilation is done, we should have GridView.exe and GridView.dll files located in the current directory. Now, who said you can't compile a Python program into an executable?

Tip of the Iceberg
Jython and IronPython are just two examples of using Python syntax in other programming environments. We saw how Python's dynamic nature fits well into statically typed JVM and CLR, despite other assumptions.

Another implementation of interest is PyPy, an implementation of Python written in statically typed dialect of Python called RPython. PyPy comes with a Just-In-Time compiler and tries to achieve most performance while staying compliant with native Python implementation.

ctypes provides compatibility with C language so that you can call shared libraries and DLLs directly from Python. No matter whether this is windll.kernel32 on Windows or libc.so.6 on Linux, ctypes gives you "pythonic" interface to all functions within. Pro*C from Python? Not a problem for ctypes.

In this installment, we covered interacting with Oracle Database using JDBC under Jython and ODP.NET under IronPython. We've seen how smooth is the transition from Java and C# to Python implementations for these platforms. Your growing investment in Python skills can pay off even more with the adoption of new platforms and extensive libraries that JVM and .NET have to offer.

Przemyslaw Piotrowski is an information technology specialist working with emerging technologies and dynamic, agile development environments. Having a strong IT background that includes administration, development and design, he finds many paths of software interoperability.