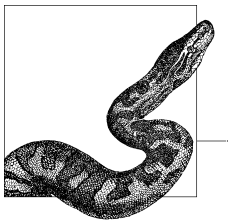

26

Extending and Embedding Jython

Jython implements the Python language on a Java Virtual Machine (JVM). Jython's built-in objects, such as numbers, sequences, dictionaries, and files, are coded in Java. To extend Classic Python with C, you code C modules using the Python C API (as covered in "Extending Python with Python's C API" on page 614). To extend Jython with Java, you do not have to code Java modules in special ways: every Java package on the Java `CLASSPATH` (or on Jython's `sys.path`) is automatically available to your Jython scripts and Jython interactive sessions for use with the `import` statement covered in "The import Statement" on page 140. This automatic availability applies to Java's standard libraries, third-party Java libraries you have installed, and Java classes you have coded yourself. You can extend Java with C using the Java Native Interface (JNI), and such extensions will be available to Jython code, just as if they were coded in pure Java rather than in JNI-compliant C.

For details on interoperation between Java and Jython, I recommend *Jython Essentials*, by Samuele Pedroni and Noel Rappin (O'Reilly). In this chapter, I offer a brief overview of the simplest interoperation scenarios, just enough for a large number of practical needs. In most cases, importing, using, extending, and implementing Java classes and interfaces in Jython *just works*. In some cases, however, you need to be aware of issues related to accessibility, type conversions, and overloading, as covered in this chapter. Embedding the Jython interpreter in Java-coded applications is similar to embedding the Python interpreter in C-coded applications (as covered in "Embedding Python" on page 647), but the Jython task is easier. Jython offers yet another possibility for interoperation with Java, using the *jythonc* compiler to turn your Python sources into classic, static JVM bytecode *.class* and *.jar* files. You can then use these bytecode files in Java applications and frameworks, exactly as if their source code had been in Java rather than in Python.

In this book, I do not cover the very similar task of extending and embedding IronPython with C# or other languages running on the CLR (Microsoft .NET, or the Mono open source implementation of CLR). However, most issues you will meet during this task are very similar to those covered in this chapter, considering the similarities between C# and Java, and the fact that the same programmer, Jim Hugunin, was responsible for initiating both Jython and IronPython. See *http://ironpython.com* for all details about IronPython, including ones related to extending and embedding tasks. At the time of this writing, the site *http://ironpython.com* was not being actively maintained, and the most up-to-date site about IronPython was instead *http://workspaces.gotdotnet.com/ironpython*. However, the IronPython team plans to revive *http://ironpython.com*, making it once more the reference site for IronPython, as soon as the team has released IronPython 1.0, which should be out by the time you read this book.

Importing Java Packages in Jython

Unlike Java, Jython does not implicitly and automatically import `java.lang`. Your Jython code can explicitly `import java.lang`, or even just `import java`, and then use classes such as `java.lang.System` and `java.lang.String` as if they were Python classes. Specifically, your Jython code can use imported Java classes as if they were Python classes with a `__slots__` class attribute (i.e., you cannot create arbitrary new instance attributes). You can subclass a Java class with your own Python class, and instances of your class do let you create new attributes just by binding them, as usual.

You may choose to import a top-level Java package (such as `java`) rather than specific subpackages (such as `java.lang`). Your Python code acquires the ability to access all subpackages when you import the top-level package. For example, after `import java`, your code can use classes `java.lang.String`, `java.util.Vector`, and so on.

The Jython runtime wraps every Java class you import in a transparent proxy, which manages communication between Python and Java code behind the scenes. This gives an extra reason to avoid the dubious idiom `from` *`somewhere`* `import *`, in addition to the reasons mentioned in "The from ... import * statement" on page 143. When you perform such a bulk import, the Jython runtime must build proxy wrappers for *all* the Java classes in package *`somewhere`*, spending substantial amounts of memory and time wrapping many classes your code will probably not use. Avoid `from ... import *`, except for occasional convenience in interactive exploratory sessions, and stick with the `import` statement. Alternatively, it's okay to use specific, explicit `from` statements for classes you know your Python code specifically wants to use (e.g., `from java.lang import System`).

The Jython Registry

Jython relies on a *registry* of Java properties as a cross-platform equivalent of the kind of settings that would normally use the Windows Registry, or environment variables on Unix-like systems. Jython's registry file is a standard Java properties file named *registry*, located in a directory known as the Jython root directory. The Jython root directory is normally the directory where *jython.jar* is located, but you can override this by setting Java properties `python.home` or `install.root`. For

special needs, you may tweak the Jython registry settings via an auxiliary Java properties file named *.jython* in your home directory, and/or via command-line options to the *jython* interpreter command. The registry option `python.path` is equivalent to Classic Python's `PYTHONPATH` environment variable. This is the option you may most often be interested in, as it can help you install extra Python packages outside of the Jython installation directories (e.g., sharing pure-Python packages that you have already installed for CPython use).

Accessibility

Normally, your Jython code can access only `public` features (methods, fields, inner classes) of Java classes. You may also choose to make `private` and `protected` features available by setting an option in the Jython registry before you run Jython:

```
python.security.respectJavaAccessibility=false
```

Such bending of normal Java rules is not necessary for normal operation. However, the ability to access private and protected features may be useful in Jython scripts meant to thoroughly test a Java package, which is why Jython gives you this option.

Type Conversions

The Jython runtime transparently converts data between Python and Java. However, when a Java method expects a `boolean` argument, you have to pass an `int` or an instance of `java.lang.Boolean` in order to call that method from Python. In Python, any object can be taken as true or false, but Jython does not perform the conversion to `boolean` implicitly on method calls to avoid confusion and the risk of errors. The new Version 2.2 of Jython, which is only out in alpha-stage at the time of this writing, also supports the more natural choice of Python type `bool` for this purpose.

Calling overloaded Java methods

A Java class can supply *overloaded* methods (i.e., several methods with the same name, distinguished by the number and types of their arguments). Jython resolves calls to overloaded methods at runtime, based on the number and types of arguments that Python code passes in each given call. If Jython's implicit overload resolution is not giving the results you expect, you can help it along by explicitly passing instances of Java's `java.lang` wrapper classes, such as `java.lang.Integer` where the Java method expects an `int` argument, `java.lang.Float` where the Java method expects a `float` argument, and so on. For example, if a Java class `C` supplies a method `M` in two overloaded versions, `M(long x)` and `M(int x)`, consider the following code:

```
import C, java.lang

c = C()
c.M(23)                     # calls M(long)
c.M(java.lang.Integer(23))  # calls M(int)
```

`c.M(23)` calls the `long` overloaded method due to the rules of Jython overload resolution. `c.M(java.lang.Integer(23))`, however, explicitly calls the `int` overloaded method.

The jarray module

When you pass Python sequences to Java methods that expect array arguments, Jython performs automatic conversion, copying each item of the Python sequence into an element of the Java array. When you call a Java method that accepts and modifies an array argument, the Python sequence that you pass cannot reflect any changes the Java method performs on its array argument. To let you effectively call methods that change their array arguments, Jython offers module `jarray`, which supplies two factory functions that let you build Java arrays directly.

array `array(`*`seq`*`,`*`typecode`*`)`

seq is any Python sequence. *typecode* is either a Java class or a single character (specifying a primitive Java type according to Table 26-1). `array` creates a Java array *a* with the same length as *seq* and elements of the class or type given by *typecode*. `array` initializes *a*'s elements from *seq*'s corresponding items.

Table 26-1. Typecodes for the jarray module

Typecode	Java type
`'b'`	`Byte`
`'c'`	`Char`
`'d'`	`Double`
`'f'`	`Float`
`'h'`	`Short`
`'i'`	`Int`
`'l'`	`Long`
`'z'`	`Boolean`

zeros `zeros(`*`length`*`,`*`typecode`*`)`

Creates a Java array *z* with length *length* and elements of the class or type given by *typecode*, which has the same meaning as in function `array`. `zeros` initializes each element of *z* to `0`, `null`, or `false`, as appropriate for the type or class. Of course, when you access such elements from Jython code, you see them as the equivalent Python `0` values (or `None` as the Jython equivalent of Java `null`), but when Java code accesses the elements, it sees them with the appropriate Java types and values.

You can use instances created by functions `array` and `zeros` as Python sequences of fixed length. When you pass such an instance to a Java method that accepts an array argument and modifies the argument, the changes are visible in the instance you passed so that your Python code can effectively call such methods.

The java.util collection classes

Jython performs no automatic conversion either way between Python containers and the collection classes of package `java.util`, such as `java.util.Vector`, `java.util.Dictionary`, and so on. However, Jython adds to the wrappers it builds for the Java collection classes a minimal amount of support to let you treat instances of collection classes as Python sequences, iterables, or mappings, as appropriate.

Subclassing a Java Class

A Python class may inherit from a Java class (equivalent to Java construct `extends`) and/or from Java interfaces (equivalent to Java construct `implements`), as well as from other Python classes. A Jython class cannot inherit, directly or indirectly, from more than one Java class. There is no limit on inheriting from interfaces. Your Jython code can access `protected` methods of the Java superclass, but not `protected` fields. You can override non-`final` superclass methods. In particular, you should always override the methods of interfaces you inherit from and abstract methods, if any, when your superclass is abstract. If a method is overloaded in the superclass, your overriding method must support all of the signatures of the overloads. To accomplish this, you can define your method to accept a variable number of arguments (by having its last formal argument use special form *`*args`*) and check at runtime for the number and types of arguments you receive on each call to know which overloaded variant was called.

JavaBeans

Jython offers special support for the JavaBeans idiom of naming accessor methods get*`SomeThing`*, is*`SomeThing`*, set*`SomeThing`*. When such methods exist in a Java class, Python code can access and set a property named *`someThing`* on instances of that Java class, using Python syntax for attribute access and binding: the Jython runtime transparently translates that syntax into calls to appropriate accessor methods.

Embedding Jython in Java

Your Java-coded application can embed the Jython interpreter in order to use Jython for scripting. *jython.jar* must be in your Java `CLASSPATH`. Your Java code must import `org.python.core.*` and `org.python.util.*` in order to access Jython's classes. To initialize Jython's state and instantiate an interpreter, use the Java statements:

```
PySystemState.initialize( );
PythonInterpreter interp = new PythonInterpreter( );
```

Jython also supplies several advanced overloads of this method and constructor in order to let you determine in detail how `PySystemState` is set up, and to control the system state and global scope for each interpreter instance. However, in typical, simple cases, the previous Java code is all your application needs.

The PythonInterpreter Class

Once you have an instance *interp* of class `PythonInterpreter`, you can call method *interp*`.eval` to have the interpreter evaluate a Python expression held in a Java string. You can also call any of several overloads of *interp*`.exec` and *interp*`.execfile` to have the interpreter execute Python statements held in a Java string, a precompiled Jython code object, a file, or a Java `InputStream`.

The Python code you execute can `import` your Java classes in order to access your application's functionality. Your Java code can set attributes in the interpreter namespace by calling overloads of *interp*`.set`, and get attributes from the interpreter namespace by calling overloads of *interp*`.get`. The methods' overloads give you a choice. You can work with native Java data and let Jython perform type conversions, or you can work directly with `PyObject`, the base class of all Python objects, covered in "The PyObject Class" on page 661. The most frequently used methods and overloads of a `PythonInterpreter` instance *interp* are the following.

eval

`PyObject` *interp*`.eval(String` *s*`)`

Evaluates, in *interp*'s namespace, the Python expression held in Java string *s*, and returns the `PyObject` that is the expression's result.

exec

`void` *interp*`.exec(String` *s*`) void` *interp*`.exec(PyObject` *code*`)`

Executes, in *interp*'s namespace, the Python statements held in Java string *s* or in compiled `PyObject` *code* (produced by function `__builtin__.compile` of package `org.python.core`, covered in "The `__builtin__` class" on page 661).

execfile

`void` *interp*`.execfile(String` *name*`) void` *interp*`.execfile(java.io.InputStream s) void` *interp*`.execfile(java.io.InputStream s,String` *name*`)`

Executes, in *interp*'s namespace, the Python statements read from the stream *s* or from the file *name*. When you pass both *s* and *name*, `execfile` reads the statements from *s* and uses *name* as the filename in error messages.

get	`PyObject` *interp*`.get(String` *name*`) Object` *interp*`.get(String` *name*`,Class` *javaclass*`)`
	
	Fetches the value of the attribute *name* from *interp*'s namespace. The overload with two arguments also converts the value to the specified *javaclass*, throwing a Java `PyException` exception that wraps a Python `TypeError` if the conversion is unfeasible. Either overload raises a `NullPointerException` if *name* is unbound. Typical use of the two-argument form might be a Java statement such as:
	
	`String s = (String)interp.get("attname", String.class);`

set	`void` *interp*`.set(String` *name*`,PyObject` *value*`) void` *interp*`.set(String` *name*`,Object` *value*`)` Binds the attribute named *name* in *interp*'s namespace to *value*. The second overload also converts the value to a `PyObject`.

The __builtin__ class

The `org.python.core` package supplies a class `__builtin__` whose static methods let your Java code access the functionality of Python built-in functions. The `compile` method, in particular, is quite similar to Python built-in function `compile`, covered in `compile` on page 160. Your Java code can call `compile` with three `String` arguments (a string of source code, a filename to use in error messages, and a *kind* that is normally `"exec"`), and `compile` returns a `PyObject` instance *p* that is a precompiled Python bytecode object. You can repeatedly call *interp*`.exec(`*p*`)` to execute the Python statements in *p* without the overhead of compiling the Python source for each execution. The advantages are the same as those covered in "Compile and Code Objects" on page 329.

The PyObject Class

Seen from Java, all Jython objects are instances of classes that extend `PyObject`. Class `PyObject` supplies methods named like Python objects' special methods, such as `__len__`, `__str__`, and so on. Concrete subclasses of `PyObject` override some special methods to supply meaningful implementations. For example, `__len__` makes sense for Python sequences and mappings, but not for numbers; `__add__` makes sense for numbers and sequences, but not for mappings. When your Java code calls a special method on a `PyObject` instance that does not in fact supply the method, the call raises a Java `PyException` exception that wraps a Python `AttributeError`.

`PyObject` methods that set, get, and delete attributes exist in two overloads, as the attribute name can be a `PyString` or a Java `String`. `PyObject` methods that set, get, and delete items exist in three overloads, as the key or index can be a `PyObject`, a

Java `String`, or an `int`. The Java `String` instances that you use as attribute names or item keys must be Java interned strings (i.e., either string literals or the result of calling *s*`.intern()` on any Java `String` instance *s*). In addition to the usual Python special methods `__getattr__` and `__getitem__`, class `PyObject` provides similar methods `__findattr__` and `__finditem__`, the difference being that, when the attribute or item is not found, the `__find...` methods return a Java `null`, while the `__get...` methods raise exceptions.

Every `PyObject` instance *p* has a method `__tojava__` that takes a single argument, a Java `Class` *c*, and returns an `Object` that is the value of *p* converted to *c* (or raises an exception if the conversion is unfeasible). Typical use might be a Java statement such as:

```
String s = (String)mypyobj.__tojava__(String.class);
```

Method `__call__` of `PyObject` has several convenience overloads, but the semantics of all the overloads boil down to `__call__`'s fundamental form:

```
PyObject p.__call__(PyObject args[], String keywords[]);
```

When array *keywords* has length *L*, array *args* must have length *N* >= *L*, and the last *L* items of *args* are taken as named arguments, the names being the corresponding items in *keywords*. When *args* has length *N* > *L*, *args*'s first *N*-*L* items are taken as positional actual arguments. The equivalent Python code is therefore similar to:

```
def docall(p, args, keywords):
    assert len(args) >= len(keywords)
    deltalen = len(args) - len(keywords)
    return p(*args[:deltalen], ** dict(zip(keywords, args[deltalen:])))
```

Jython supplies concrete subclasses of `PyObject` that represent all built-in Python types. You can instantiate such a concrete subclass in order to create a `PyObject` for further use. For example, class `PyList` extends `PyObject`, implements a Python list, and has constructors that take an array or a `java.util.Vector` of `PyObject` instances, as well as an empty constructor that builds the empty list `[]`.

The Py Class

The `Py` class supplies several utility class attributes and static methods. `Py.None` is Python's `None`. Method `Py.java2py` takes a Java `Object` argument and returns the corresponding `PyObject`. Methods `Py.py2`*type*, for all values of *type* that name a Java primitive type (`boolean`, `byte`, `long`, `short`, etc.), take a `PyObject` argument and return the corresponding value of the given primitive Java type.

Compiling Python into Java

Jython comes with the *jythonc* compiler. You can feed *jythonc* your *.py* source files, and *jythonc* compiles them into normal JVM bytecode and packages them into *.class* and *.jar* files. Since *jythonc* generates traditional static bytecode, it cannot quite cope with the whole range of dynamic possibilities that Python allows. For example, *jythonc* cannot successfully compile Python classes that

determine their base classes dynamically at runtime, as the normal Python interpreters allow. However, except for such extreme examples of dynamically changeable class structures, *jythonc* does support compilation of essentially the whole Python language into Java bytecode.

The jythonc Command

jythonc resides in the *Tools/jythonc* directory of your Jython installation. You invoke it from a shell (console) command line with the syntax:

```
jythonc options modules
```

options are zero or more option flags starting with --. *modules* are zero or more names of Python source files to compile, either as Python-style names of modules residing on Python's `sys.path`, or as relative or absolute paths to Python source files. Include the *.py* extension in each path to a source file, but not in a module name.

More often than not, you will specify the *jythonc* option *--jar jarfile* to build a *.jar* file of compiled bytecode rather than separate *.class* files. Most other options deal with what to put in the *.jar* file. You can choose to make the file self-sufficient (for browsers and other Java runtime environments that do not support the use of multiple *.jar* files) at the expense of making the file larger. Option *--all* ensures all Jython core classes are copied into the *.jar* file, while *--core* tries to be more conservative, copying as few core classes as feasible. Option *--addpackages packages* lets you list (in *packages*, a comma-separated list) those external Java packages whose classes are copied into the *.jar* file if any of the Python classes *jythonc* is compiling depends on them. An important alternative to *--jar* is *--bean jarfile*, which also includes a bean manifest in the *.jar* file as needed for Python-coded JavaBeans components.

Another useful *jythonc* option is *--package package*, which instructs Jython to place all the new Java classes it's creating in the given *package* (and any subpackages of *package* needed to reflect the Python-side package structure).

Adding Java-Visible Methods

The Java classes that *jythonc* creates normally extend existing classes from Java libraries and/or implement existing interfaces. Other Java-coded applications and frameworks instantiate the *jythonc*-created classes via constructor overloads, which have the same signatures as the constructors of their Java superclasses. The Python-side `__init__` executes after the superclass is initialized, and with the same arguments (therefore, don't `__init__` a Java superclass in the `__init__` of a Python class meant to be compiled by *jythonc*). Afterward, Java code can access the functionality of instances of Python-coded classes by calling instance methods defined in known interfaces or superclasses and overridden by Python code.

Python code can never supply Java-visible `static` methods or attributes, only instance methods. By default, each Python class supplies only the instance methods it inherits from the Java class it extends or the Java interfaces it implements. However, Python code can also supply other Java-visible instance methods via the `@sig` directive.

To expose a method of your Python class to Java when *jythonc* compiles the class, code the method's docstring as `@sig` followed by a Java method signature. For example:

```
class APythonClass(java.lang.object):
    def __init_  _(self, greeting="Hello, %s!"):
        "@sig public APythonClass(String greeting)"
        self.greeting = greeting
    def hello(self, name):
        "@sig public String hello(String name)"
        return self.greeting % name
```

To expose a constructor, use the `@sig` signature for the class, as shown in the previous example's `__init__` method docstring. All names of classes in `@sig` signatures must be fully qualified, except for names that come from `java.lang` and names supplied by the Python-coded module being compiled. When a Python method with a `@sig` has optional arguments, *jythonc* generates Java-visible overloads of the method with each legal signature and deals with supplying the default argument values where needed. An `__init__` constructor with a `@sig`, for a Python class that extends a Java class, implicitly initializes the superclass using the superclass's empty constructor.

Since a Python class cannot expose data attributes directly to Java, you may need to code accessors with the usual JavaBeans convention and expose them via the `@sig` mechanism. For example, instances of `APythonClass` in the above example do not allow Java code to directly access or change the `greeting` attribute. When such functionality is needed, you can supply it in a subclass as follows:

```
class APythonBean(APythonClass):
    def getGreeting(self):
        "@sig public String getGreeting( )"
        return self.greeting
    def setGreeting(self, greeting):
        "@sig public void setGreeting(String greeting)"
        self.greeting = greeting
```

Python Applets and Servlets

Two simple examples of using Jython within existing Java frameworks are applets and servlets. Applets are typical examples of *jythonc* use (with specific caveats), while servlets are specifically supported by a Jython-supplied utility.

Python applets

A Jython applet class must import `java.applet.Applet` and extend it, typically overriding method `paint` and others. You compile the applet into a *.jar* file by calling *jythonc* with options *--jar somejar.jar* and either *--core* or *--all*. Normally, Jython is installed in a modern Java 2 environment, which is okay for most uses. It is fine for applets, as long as the applets run only in browsers that support Java 2, typically with a Sun-supplied browser plug-in. However, if you need to support browsers that are limited to Java 1.1, you must ensure that the JDK you use is

Release 1.1, and that you compile your applet with Jython under a JDK 1.1 environment. It's possible to share a single Jython installation between different JDKs, such as 1.1 and 1.4. However, I suggest you perform separate installations of Jython, one under each JDK you need to support, in separate directories, in order to minimize the risk of confusion and accidents.

Python servlets

You can use *jythonc* to build and deploy servlets. However, Jython also supports an alternative that lets you deploy Python-coded servlets as source *.py* files. Use the servlet class `org.python.util.PyServlet`, supplied with Jython, and a servlet mapping of all **.py* URLs to `PyServlet`. Each servlet *.py* file must reside in the *web-app* top-level directory, and must expose an object callable without arguments (normally a class) with the same name as the file. `PyServlet` uses that callable as a factory for instances of the servlet, and calls methods on the instance according to the Java Servlet API. Your servlet instance, in turn, accesses Servlet API objects such as the *`request`* and *`response`* objects, passed as method arguments, and these objects' attributes and methods, such as *`response`*`.outputStream` and *`request`*`.getSession`. `PyServlet` provides an excellent, fast-turnaround way to experiment with servlets and rapidly deploy them.