JPype Documentation

Release 0.6.2

Steve Menard, Luis Nell and others

Mar 27, 2017

Contents

JPype is an effort to allow python programs full access to java class libraries. This is achieved not through re-implementing Python, as Jython/JPython has done, but rather through interfacing at the native level in both virtual machines. Eventually, it should be possible to replace Java with python in many, though not all, situations. JSP, Servlets, RMI servers and IDE plugins are good candidates.

CHAPTER 1

Parts of the documentation

Installation

Get JPype from the github or from PyPi. If you are using Anaconda Python stack, you can install pre-compiled binaries from conda-forge for Linux, OSX and Windows.

Binary Install

1. Ensure you have installed Anaconda/Miniconda. Instructions can be found here.
2. Install from the conda-forge software channel:

```
conda install -c conda-forge jpype1
```

From source - Requirements

Either the Sun/Oracle JDK/JRE Variant or OpenJDK. Python 2.6+ (including Python 3+).

Debian/Ubuntu

Debian/Ubuntu users will have to install `g++` and `python-dev` first:

```
sudo apt-get install g++ python-dev
```

Windows

Windows users need a Python installation and C++ compilers:

1. Install some version of Python (2.7 or higher), e.g., Anaconda is a good choice for users not yet familiar with the language

2. Install a Windows C++ Compiler

Install

Should be easy as

```
python setup.py install
```

If it fails...

This happens mostly due to the setup not being able to find your JAVA_HOME. In case this happens, please do two things:

1. You can continue the installation by finding the JAVA_HOME on your own (the place where the headers etc. are) and explicitly setting it for the installation:

   JAVA_HOME=/usr/lib/java/jdk1.6.0/ python setup.py install

2. Please create an Issue on github and post all the information you have.

Tested on

- OS X 10.7.4 with Sun/Oracle JDK 1.6.0
- OSX 10.8.1-10.8.4 with Sun/Oracle JDK 1.6.0
- OSX 10.9 DP5
- Debian 6.0.4/6.0.5 with Sun/Oracle JDK 1.6.0
- Debian 7.1 with OpenJDK 1.6.0
- Ubuntu 12.04 with Sun/Oracle JDK 1.6.0

Known Bugs/Limitations

- Java classes outside of a package (in the <default>) cannot be imported.
- unable to access a field or method if it conflicts with a python keyword.
- Because of lack of JVM support, you cannot shutdown the JVM and then restart it.
- Some methods rely on the "current" class/caller. Since calls coming directly from python code do not have a current class, these methods do not work. The User Manual lists all the known methods like that.
- Mixing 64 bit Python with 32 bit Java and vice versa crashes on import jpype.

User Guide

Overview

JPype is an effort to allow Python programs full access to Java class libraries. This is achieved not through re-implementing Python, as Jython/JPython has done, but rather through interfacing at the native level in both virtual machines.

Eventually, it should be possible to replace Java with Python in many, though not all, situations. JSP, Servlets, RMI servers and IDE plugins are all good candidates.

Once this integration is achieved, a second phase will be started to separate the Java logic from the Python logic, eventually allowing the bridging technology to be used in other environments, i.e. Ruby, Perl, COM, etc ...

Why such a project?

As much as I enjoy programming in Python, there is no denying that Java has the bulk of the mindshare. Just look on Sourceforge, at the time of creation of this project, there were 3267 Python-related projects, and 12126 Java- related projects. And that's not counting commercial interests.

Server-side Python is also pretty weak. Zope may be a great application server, but I have never been able to figure it out. Java, on the other hand, shines on the server.

So in order to both enjoy the language, and have access to the most popular libraries, I have started this project.

What about Jython?

Jython (formerly known as JPython) is a great idea. However, it suffers from a large number of drawbacks, i.e. it always lags behind CPython, it is slow and it does not allow access to most Python extensions.

My idea allows using both kinds of libraries in tandem, so the developer is free to pick and choose.

Using JPype

Here is a sample program to demonstrate how to use JPype:

```
from jpype import *
startJVM(getDefaultJVMPath(), "-ea")
java.lang.System.out.println("hello world")
shutdownJVM()
```

This is of course a simple **hello world** type of application. Yet it shows the 2 most important calls: **startupJVM** and **shutdownJVM**.

The rest will be explained in more detail in the next sections.

Core Ideas

Threading

Any non-trivial application will have need of threading. Be it implicitly by using a GUI, or because you're writing a multi-user server. Or explicitly for performance reason.

The only real problem here is making sure Java threads and Python threads cooperate correctly. Thankfully, this is pretty easy to do.

Python Threads

For the most part, Python threads based on OS level threads (i.e. posix threads) will work without problem. The only thing to remember is to call **jpype.attachThreadToJVM()** in the thread body to make the JVM usable from that thread. For threads that you do not start yourself, you can call **isThreadAttachedToJVM()** to check.

Java Threads

At the moment, it is not possible to use threads created from Java, since there is no **callback** available.

Other Threads

Some Python libraries offer other kinds of thread, (i.e. microthreads). How they interact with Java depends on their nature. As stated earlier, any OS- level threads will work without problem. Emulated threads, like microthreads, will appear as a single thread to Java, so special care will have to be taken for synchronization.

Synchronization

Java synchronization support can be split into 2 categories. The first is the **synchronized** keyword, both as prefix on a method and as a block inside a method. The second are the different methods available on the Object class (**notify, notifyAll, wait**).

To support the **synchronized** functionality, JPype defines a method called synchronized(O). O has to be a Java object or Java class, or a Java wrapper that corresponds to an object (JString and JObject). The return value is a monitor object that will keep the synchronization on as long as the object is kept alive. The lock will be broken as soon as the monitor is GCd. So make sure to hang on to it as long as you need it.

The other methods are available as-is on any _JavaObject.

For synchronization that does not have to be shared with Java code, I suggest using Python's support instead of Java's, as it'll be more natural and easier.

Performance

JPype uses JNI, which is well known in the Java world as not being the most efficient of interfaces. Further, JPype bridges two very different runtime environments, performing conversion back and forth as needed. Both of these can impose rather large performance bottlenecks.

JNI is the standard native interface for most, if not all, JVMs, so there is no getting around it. Down the road, it is possible that interfacing with CNI (GCC's java native interface) may be used. The only way to minimize the JNI cost is to move some code over to Java.

Follow the regular Python philosophy : **Write it all in Python, then write only those parts that need it in C.** Except this time, it's write the parts that need it in Java.

For the conversion costs, again, nothing much can be done. In cases where a given object (be it a string, an object, an array, etc ...) is passed often into Java, you can pre-convert it once using the wrappers, and then pass in the wrappers. For most situations, this should solve the problem.

As a final note, while a JPype program will likely be slower than its pure Java counterpart, it has a good chance of being faster than the pure Python version of it. The JVM is a memory hog, but does a good job of optimizing code execution speeds.

Inner Classes

For the most part, inner classes can be used like normal classes, with the following differences:

- Inner classes in Java natively use $ to separate the outer class from the inner class. For example, inner class Foo defined inside class Bar is called Bar.Foo in Java, but its real native name is Bar$Foo.

- Because of this name mangling, you cannot use the standard package access method to get them. Use the method __getclass__ in JPackage to load them.
- Non-static inner classes cannot be instantiated from Python code. Instances received from Java code that can be used without problem.

Arrays

JPype has full support for receiving Java arrays and passing them to Java methods. Java arrays, wrapped in the JArray wrapper class, behave like Python lists, except that their size is fixed, and that the contents are of a specific type.

Multi-dimensional arrays (array of arrays) also work without problem.

As of version 0.5.5.3 we use NumPy arrays to interchange data with Java. This is much faster than using lists, since we do not need to handle every single array element but can process all data at once.

If you do not want this optional feature, because eg. it depends on NumPy, you can opt it out in the installation process by passing *"–disable-numpy"* to *setup.py*. To opt out with pip you need to append the additional argument *"–install-option='–disable-numpy'*. This possibility exists since version 0.5.6.

Creating Java arrays from Python

The JArray wrapper is used to create Arrays from Python code. The code to create an array is like this:

```
JArray(type, num_dims)(sz or sequence)
```

Type is either a Java Class (as a String or a JavaClass object) or a Wrapper type. num_dims is the number of dimensions to build the array and defaults to 1.

sz is the actual number of elements in the arrays, and sequence is a sequence to initialize the array with.

The logic behind this is that JArray(type, ndims) returns an Array Class, which can then be called like any other class to create an instance.

Type conversion

One of the most complex parts of a bridge system like JPype is finding a way to seamlessly translate between Python types and Java types. The following table will show what implicit conversions occur, both Python to Java and Java to Python. Explicit conversion, which happens when a Python object is wrapped, is converted in each wrapper.

Conversion from Python to Java

This type of conversion happens when a Python object is used either as a parameter to a Java method or to set the value of a Java field.

Type Matching

JPype defines different levels of "match" between Python objects and Java types. These levels are:

- **none**, There is no way to convert.
- **explicit (E)**, JPype can convert the desired type, but only explicitly via the wrapper classes. This means the proper wrapper class will access this type as argument.
- **implicit (I)**, JPype will convert as needed.

- **exact> (X)**, Like implicit, but when deciding with method overload to use, one where all the parameters match "exact" will take precedence over "implicit" matches.

Python\Java	byte	short	int	long	float	double	boolean	char	String	Array	Object	Class
int	I[1]	I[1]	X	I			X[10]					
long	I[1]	I[1]	I[1]	X								
float					I[1]	X						
sequence												
dictionary												
string								I[2]	X			
unicode								I[2]	X			
JByte	X											
JShort		X										
JInt			X									
JLong				X								
JFloat					X							
JDouble						X						
JBoolean							X					
JString									X		I[3]	
JChar								X				
JArray										I/X[4]	I[5]	
JObject										I/X[6]	I/X[7]	
JavaObject											I[8]	
JavaClass											I[9]	X

Converting from Java to Python

The rules here are much simpler.

Java **byte, short and int** are converted to Python **int**.

Java **long** is converted to Python **long**.

Java **float and double** are converted to Python **float**.

Java **boolean** is converted to Python **int** of value 1 or 0.

Java **char** is converted to Python **unicode** of length 1.

Java **String** is converted to Python **unicode**.

Java **arrays** are converted to **JArray**.

All other Java objects are converted to **JavaObject**s.

Java **Class** is converted to **JavaClass**.

Java array **Class** is converted to **JavaArrayClass**.

[1] Conversion will occur if the Python value fits in the Java native type.
[10] Only the values True and False are implicitly converted to booleans.
[2] Conversion occurs if the Python string or unicode is of length 1.
[3] The required object must be of a type compatible with `java.lang.String(java.lang.Object, java.util.Comparable)`.
[4] Number of dimensions must match, and the types must be compatible.
[5] Only when the required type is `java.lang.Object`.
[6] Only if the JObject wrapper's specified type is an compatible array class.
[7] Only if the required type is compatible with the wrappers's specified type. The actual type of the Java object is not considered.
[8] Only if the required type is compatible with the Java Object actual type.
[9] Only when the required type is `java.lang.Object` or `java.lang.Class`.

JProxy

The JProxy allows Python code to "implement" any number of Java interfaces, so as to receive callbacks through them.

Using JProxy is simple. The constructor takes 2 arguments. The first is one or a sequence of string of JClass objects, defining the interfaces to be "implemented". The second must be a keyword argument, and be either **dict** or **inst**. If **dict** is specified, then the 2nd argument must be a dictionary, with the keys the method names as defined in the interface(s), and the values callable objects. If **inst** an object instance must be given, with methods defined for the methods declared in the interface(s). Either way, when Java calls the interface method, the corresponding Python callable is looked up and called.

Of course, this is not the same as subclassing Java classes in Python. However, most Java APIs are built so that subclassing is not needed. Good examples of this are AWT and SWING. Except for relatively advanced features, it is possible to build complete UIs without creating a single subclass.

For those cases where subclassing is absolutely necessary (i.e. using Java's SAXP classes), it is generally easy to create an interface and a simple subclass that delegates the calls to that interface.

Sample code :

Assume a Java interface like:

```
public interface ITestInterface2
{
        int testMethod();
        String testMethod2();
}
```

You can create a proxy *implementing* this interface in 2 ways. First, with a class:

```
class C :
        def testMethod(self) :
                return 42

        def testMethod2(self) :
                return "Bar"

c = C()
proxy = JProxy("ITestInterface2", inst=c)
```

or you can do it with a dictionary

```
def _testMethod() :
return 32

def _testMethod2() :
return "Fooo!"

d = {
      'testMethod' : _testMethod,
      'testMethod2' : _testMethod2,
}
proxy = JProxy("ITestInterface2", dict=d)
```

Java Exceptions

Error handling is a very important part of any non-trivial program. So bridging Java's exception mechanism and Python's is very important.

Java exception classes are regular classes that extend, directly or indirectly, the java.lang.Throwable class. Python exceptions are classes that extend, directly or indirectly, the Exception class. On the surface they are similar, at the C-API level, Python exceptions are completely different from regular Python classes. This contributes to the fact that it is not possible to catch Java exceptions in a completely straightforward way.

All Java exceptions thrown end up throwing the jpype.JavaException exception. You can then use the message(), stackTrace() and javaClass() to access extended information.

Here is an example:

```
try :
        # Code that throws a java.lang.RuntimeException
except JavaException, ex :
        if JavaException.javaClass() is java.lang.RuntimeException :
                print "Caught the runtime exception : ", JavaException.message()
                print JavaException.stackTrace()
```

Alternately, you can catch the REAL Java exception directly by using the JException wrapper.

```
try :
        # Code that throws a java.lang.RuntimeException
except jpype.JException(java.lang.RuntimeException), ex :
        print "Caught the runtime exception : ", JavaException.message()
        print JavaException.stackTrace()
```

Known limitations

This section lists those limitations that are unlikely to change, as they come from external sources.

Unloading the JVM

The JNI API defines a method called destroyJVM(). However, this method does not work. That is, Sun's JVMs do not allow unloading. For this reason, after calling shutdownJVM(), if you attempt calling startupJVM() again you will get a non-specific exception. There is nothing wrong (that I can see) in JPype. So if Sun gets around to supporting its own properly, or if you use JPype with a non-SUN JVM that does (I believe IBM's JVMs support JNI invocation, but I do not know if their destroyJVM works properly), JPype will be able to take advantage of it. As the time of writing, the latest stable Sun JVM was 1.4.2_04.

Methods dependent on "current" class

There are a few methods in the Java libraries that rely on finding information on the calling class. So these methods, if called directly from Python code, will fail because there is no calling Java class, and the JNI API does not provide methods to simulate one.

At the moment, the methods known to fail are :

java.lang.Class.forName(String classname)

This method relies on the current class's classloader to do its loading. It can easily be replaced with **Class.forName(classname, True, ClassLoader.getSystemClassLoader())**.

java.sql.DriverManager.getConnection(...)

For some reason, this class verifies that the driver class as loaded in the "current" classloader is the same as previously registered. Since there is no "current" classloader, it defaults to the internal classloader, which typically does not find the driver. To remedy, simply instantiate the driver yourself and call its connect(...) method.

Unsupported Java virtual machines

The open JVM implementations *Cacao* and *JamVM* are known not to work with JPype.

Module Reference

getDefaultJVMPath method

This method tries to automatically obtain the path to a Java runtime installation. This path is needed as argument for startupJVM method and should be used in favour of hardcoded paths to make your scripts more portable. There are several methods under the hood to search for a JVM. If none of them succeeds, the method will raise a JVMNotFoundException.

Arguments

None

Return value

valid path to a Java virtual machine library (jvm.dll, jvm.so, jvm.dylib)

Exceptions

JVMNotFoundException, if none of the provided methods returned a valid JVM path.

startupJVM method

This method MUST be called before any other JPype features can be used. It will initialize the specified JVM.

Arguments

- vmPath - Must be the path to the jvm.dll (or jvm.so, depending on platform)

- misc arguments - All arguments after the first are optional, and are given as it to the JVM. Pretty much any command-line argument you can give the JVM can be passed here. A caveat, multi-part arguments (like -classpath) do not seem to work, and must e passed in as a -D option. Option **-classpath a;b;c** becomes **-Djava.class.path=a;b;c**

Return value

None

Exceptions

On failure, a RuntimeException is raised.

shutdownJVM method

For the most part, this method does not have to be called. It will be automatically executed when the jpype module is unloaded at Python's exit.

Arguments

None

Return value

None

Exceptions

On failure, a RuntimeException is raised.

attachThreadToJVM method

For the most part, this method does not have to be called. It will be automatically executed when the jpype module is unloaded at Python's exit.

Arguments

None

Return value

None

Exceptions

On failure, a RuntimeException is raised.

isThreadAttachedToJVM method

For the most part, this method does not have to be called. It will be automatically executed when the jpype module is unloaded at Python's exit.

Arguments

None

Return value

None

Exceptions

On failure, a RuntimeException is raised.

detachThreadFromJVM method

For the most part, this method does not have to be called. It will be automatically executed when the jpype module is unloaded at Python's exit.

Arguments

None

Return value

None

Exceptions

On failure, a RuntimeException is raised.

synchronized method

For the most part, this method does not have to be called. It will be automatically executed when the jpype module is unloaded at Python's exit.

Arguments

None

Return value

None

Exceptions

On failure, a RuntimeException is raised.

JPackage class

This class allows structured access to Java packages and classes. It is very similar to a Python import statement.

Only the root of the package tree need be declared with the JPackage constructor. Sub-packages will be created on demand.

For example, to import the w3c DOM package:

```
Document = JPackage('org').w3c.dom.Document
```

Predefined Java packages

For convenience, the jpype module predefines the following JPackages : **java, javax**

They can be used as-is, without needing to resort to the JPackage class.

Wrapper classes

The main problem with exposing Java classes and methods to Python, is that Java allows overloading a method. That is, 2 methods can have the same name as long as they have different parameters. Python does not allow that. Most of the time, this is not a problem. Most overloaded methods have very different parameters and no confusion takes place.

When JPype is unable to decide with overload of a method to call, the user must resolve the ambiguity. That's where the wrapper classes come in.

Take for example the java.io.PrintStream class. This class has a variant of the print and println methods!

So for the following code:

```
from jpype import *
startJVM(getDefaultJVMPath(), "-ea")
java.lang.System.out.println(1)
shutdownJVM()
```

JPype will automatically choose the println(int) method, because the Python int matches exactly with the Java int, while all the other integral types are only "implicit" matches. However, if that is not the version you wanted to call ...

Changing the line thus:

```
from jpype import *
startJVM(getDefaultJVMPath(), "-ea")
java.lang.System.out.println(JByte(1)) # <--- wrap the 1 in a JByte
shutdownJVM()
```

tells JPype to choose the byte version.

Note that wrapped object will only match to a method which takes EXACTLY that type, even if the type is compatible. Using a JByte wrapper to call a method requiring an int will fail.

One other area where wrappers help is performance. Native types convert quite fast, but strings, and later tuples, maps, etc ... conversions can be very costly.

If you're going to make many Java calls with a complex object, wrapping it once and then using the wrapper will make a huge difference.

Lastly, wrappers allow you to pass in a structure to Java to have it modified. An implicitly converted tuple will not come back modified, even if the Java method HAS changed the contents. An explicitly wrapped tuple will be modified, so that those modifications are visible to the Python program.

The available native wrappers are: **JChar, JByte, JShort, JInt, JLong, JFloat, JDouble, JBoolean and JString.**

JObject wrapper

The JObject wrapper serves a few additional purposes on top of what the other wrappers do.

While the native wrappers help to resolve ambiguities between native types, it is impossible to create one JObject wrapper for each Java Class to do the same thing.

So, the JObject wrapper accepts 2 parameters. The first is any convertible object. The second is the class to convert it to. It can be the name of the class in a string or a JavaClass object. If omitted, the second parameter will be deduced from the first.

Like other wrappers, the method called will only match EXACTLY. A JObject wrapper of type java.lang.Int will not work when calling a method requiring a java.lang.Number.

Changelog

This changelog *only* contains changes from the *first* pypi release (0.5.4.3) onwards.

- **Next version - unreleased**
- **0.6.2 - 2017-01-13**
  - Fix JVM location for OSX.
  - Fix a method overload bug.
  - Add support for synthetic methods
- **0.6.1 - 2015-08-05**
  - Fix proxy with arguments issue.
  - Fix Python 3 support for Windows failing to import winreg.
  - Fix non matching overloads on iterating java collections.
- **0.6.0 - 2015-04-13**
  - Python3 support.

  - Fix OutOfMemoryError.

- **0.5.7 - 2014-10-29**
  - No JDK/JRE is required to build anymore due to provided jni.h. To override this, one needs to set a JAVA_HOME pointing to a JDK during setup.
  - Better support for various platforms and compilers (MinGW, Cygwin, Windows)
- **0.5.6 - 2014-09-27**
  - *Note*: In this release we returned to the three point number versioning scheme.
  - Fix #63: 'property' object has no attribute 'isBeanMutator'
  - Fix #70: python setup.py develop does now work as expected
  - Fix #79, Fix #85: missing declaration of 'uint'
  - Fix #80: opt out NumPy code dependency by '–disable-numpy' parameter to setup. To opt out with pip append –install-option="–disable-numpy".
  - Use JVMFinder method of @tcalmant to locate a Java runtime
- **0.5.5.4 - 2014-08-12**
  - Fix: compile issue, if numpy is not available (NPY_BOOL n/a). Closes #77
- **0.5.5.3 - 2014-08-11**
  - Optional support for NumPy arrays in handling of Java arrays. Both set and get slice operators are supported. Speed improvement of factor 10 for setting and factor 6 for getting. The returned arrays are typed with the matching NumPy type.
  - Fix: add missing wrapper type 'JShort'
  - Fix: Conversion check for unsigned types did not work in array setters (tautological compare)
- **0.5.5.2 - 2014-04-29**
  - Fix: array setter memory leak (ISSUE: #64)
- **0.5.5.1 - 2014-04-11**
  - Fix: setup.py now runs under MacOSX with Python 2.6 (referred to missing subprocess function)
- **0.5.5 - 2014-04-11**
  - *Note* that this release is *not* compatible with Python 2.5 anymore!
  - Added AHL changes
    - replaced Python set type usage with new 2.6.x and higher
    - fixed broken Python slicing semantics on JArray objects
    - fixed a memory leak in the JVM when passing Python lists to JArray constructors
    - prevent ctrl+c seg faulting
    - corrected new[]/delete pairs to stop valgrind complaining
    - ship basic PyMemoryView implementation (based on numpy's) for Python 2.6 compatibility
  - Fast sliced access for primitive datatype arrays (factor of 10)
  - Use setter for Java bean property assignment even if not having a getter by @baztian

  - Fix public methods not being accessible if a Java bean property with the same name exists by @baztian (*Warning*: In rare cases this change is incompatibile to previous releases. If you are accessing a bean property without using the get/set method and the bean has a public method with the property's name you have to change the code to use the get/set methods.)
  - Make jpype.JException catch exceptions from subclasses by @baztian
  - Make more complex overloaded Java methods accessible (fixes https://sourceforge.net/p/jpype/bugs/69/) by @baztian and anonymous
  - Some minor improvements inferring unnecessary copies in extension code
  - Some JNI cleanups related to memory
  - Fix memory leak in array setters
  - Fix memory leak in typemanager
  - Add userguide from sourceforge project by @baztian
- **0.5.4.5 - 2013-08-25**
  - Added support for OSX 10.9 Mavericks by @rmangino (#16)
- **0.5.4.4 - 2013-08-10**
  - Rewritten Java Home directory Search by @marsam (#13, #12 and #7)
  - Stylistic cleanups of setup.py
- **0.5.4.3 - 2013-07-27**
  - Initial pypi release with most fixes for easier installation

CHAPTER 2

Indices and tables

- genindex
- modindex
- search