JPype Documentation

Release 0.7.0

Steve Menard, Luis Nell and others

Jul 22, 2019

Contents

JPype is a Python module to provide full access to Java from within Python. It allows Python to make use of Java only libraries, exploring and visualization of Java structures, development and testing of Java libraries, scientific computing, and much more. By gaining the best of both worlds using Python for rapid prototyping and Java for strong typed production code, JPype provides a powerful environment for engineering and code development.

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. This shared memory based approach achieves decent computing preformance, while providing the access to the entirety of CPython and Java libraries.

CHAPTER 1

Parts of the documentation

1.1 Installation

JPype is available either as a pre-compiled binary for Anaconda, or may be build from source though several methods.

1.1.1 Binary Install

JPype can be installed as pre-compiled binary if you are using the Anaconda Python stack. Binaries are available for Linux, OSX, ad windows are available on conda-forge.

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
```

1.1.2 Source Install

Installing from source requires

Python JPype works with CPython 2.6 or later including the 3 series. Both the runtime and the development package are required.

Java Either the Sun/Oracle JDK/JRE Variant or OpenJDK. Python 2.6+ (including Python 3+). JPype source distribution includes a copy of the Java JNI header and precompiled Java code, so the development kit is not required from the source distribution. JPype has been used with Java versions from Java 1.7 to Java 11.

C++ A C++ compiler which matches the ABI used to build CPython.

Ant and JDK *(Optional)* JPype contains sections of Java code. These sections are precompiled in the source distribution but must be build with installing from git directly.

Once these requirements have been met one can use pip to build either from the source distribution or directly from the repo. Specific requirements from different achitectures are shown *below*.

Build using pip

JPype may be build and installed with one easy step using pip.

To install the latest JPype using the source distribute use:

```
pip install JPype1
```

On some systems (e.g. Fedora Linux), this installation was found not to enable numpy support. If this happens, the following forces compilation and enables numpy support if it is available:

```
pip install --no-binary :all: JPype1
```

To install from the current github master use:

```
pip install git+https://github.com/jpype-project/jpype.git
```

More details on installing from git can be found at Pip install. The git version does not include a prebuilt jar and thus both ant and JDK are required.

Build and install manually

JPype can be built entirely from source. Just follow these three easy steps.

1. Get the JPype source

Either from github or from PyPi.

2. Build the source with desired options

Compile JPype using the included `setup.py` script with:

```
python setup.py build
```

The setup script recognizes several arguments.

--ant	Define the location of ant on your system using `--ant=path`. This option is useful when building when ant is not in the path.
--enable-build-jar	Force setup to recreate the jar from scratch.
--enable-tracing	Build a verison of JPype with full logging to the console. This can be used to diagnose tricky JNI issues.
--disable-numpy	Do not compile with numpy extenstions.

After building, JPype can be tested using the test bench. The test bench requires ant and JDK to build.

3. Test JPype with (optional):

```
python setup.py test
```

4. Install JPype with:

```
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/jdk1.8.0/ python setup.py install
```

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

Specific requirements

JPype is known to work on Linx, OSX, Windows, and Cygwin. To make it easier to those who have not built CPython modules before here are some helpful tips for different machines.

Debian/Ubuntu

Debian/Ubuntu users will have to install `g++`, `python-dev`, and `ant` (optional) Use:

```
sudo apt-get install g++ python-dev python3-dev ant
```

Windows

Windows users need a CPython installation and C++ compilers specificly for CPython:

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. For Python 2 series, Install Windows C++ Compiler
3. For Python 3 series, Install Microsoft Visual Studio 2010 Redistributable Package (x64) and Microsoft Build Tools 2015 Update 3
4. (optional) Install Apache Ant (tested using 1.9.13)

Netbeans ant can be used in place of Apache Ant. Netbeans ant is located in `${netbeans}/extide/ant/bin/ant.bat`.

Due to differences in the C++ API, only the version specified will work to build CPython modules. The Build Tools 2015 is a pain to find. Microsoft really wants people to download the latest version. To get to it from the above URL, click on "Redistributables and Build Tools", then select Microsoft Build Tools 2015 Update 3.

When building for windows you must use the Visual Studio developer command prompt.

1.1.3 Known Bugs/Limitations

- Java classes outside of a package (in the `<default>`) cannot be imported.
- Because of lack of JVM support, you cannot shutdown the JVM and then restart it.
- Structural issues prevent managing objects from more than one JVM at a time.
- 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 Guide* lists all the known methods like that.

- Mixing 64 bit Python with 32 bit Java and vice versa crashes on import jpype.

1.2 User Guide

1.2.1 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.

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: `startJVM` and `shutdownJVM`.

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

1.2.2 Core Ideas

1.2.3 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 two 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. The return value is a monitor object that will keep the synchronization on as long as the object is kept alive. Use Python `with` statement to control the exact scope. Do not hold onto the object indefinitly without a `with` statement, the lock will be not be broken until the monitor is garbage collected. CPython and PyPy have difference GC rules. See *synchronized* for details of how to properly synchronize.

The other synchronization methods are available as-is on any `JObject`. However, as general rule one should not use synchronization methods on Java String as internal string representations may not be complete objects.

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.

1.2.4 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.

1.2.5 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.
- Inner classes appear as member of the containing class. Thus to access them simply import the outer class and call them as members.
- Non-static inner classes cannot be instantiated from Python code. Instances received from Java code that can be used without problem.

1.2.6 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 *"–global-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 Java Class 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.

1.2.7 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.

There are special rules for `java.lang.Object` as compared with a specific Java object. In Java, primitives are boxed automatically when passing to a `java.lang.Object`.

Python\Java	byte	short	int	long	float	double	boolean	char	String	Array	Object	java.lang.Object	java.lang.Class
int	I[1]	I[1]	X	I	I[3]	I[3]	X[8]					I[11]	
long	I[1]	I[1]	I[1]	X	I[3]	I[3]						I[11]	
float					I[1]	X						I[12]	
sequence													
dictionary													
string								I[2]	X			I	
unicode								I[2]	X			I	
JByte	X											I[9]	
JShort		X										I[9]	
JInt			X									I[9]	
JLong				X								I[9]	
JFloat					X							I[9]	
JDouble						X						I[9]	
JBoolean							X					I[9]	
JChar								X				I[9]	
JString									X			I	
JArray										I/X[4]		I	
JObject										I/X[6]	I/X[7]	I/X[7]	
JClass												I	X
"Boxed"[10]	I	I	I	I	I	I	I					I	

[1] Conversion will occur if the Python value fits in the Java native type.
[3] Java defines conversions from integer types to floating point types as implicit conversion. Java's conversion rules are based on the range and can be lossy. See (http://stackoverflow.com/questions/11908429/java-allows-implicit-conversion-of-int-to-float-why)
[8] Only the values True and False are implicitly converted to booleans.
[11] Boxed to `java.lang.Long` as there is no difference between long and int in Python3,
[12] Boxed to `java.lang.Double`
[2] Conversion occurs if the Python string or unicode is of length 1.
[9] Primitives are boxed as per Java rules.
[4] Number of dimensions must match, and the types must be compatible.
[6] Only if the specified type is an compatible array class.
[7] Exact is the object class is an exact match, otherwise implicit.
[10] Java boxed types are mapped to python primitives, but will produce an implicit conversion even if the python type is an exact match. This is to allow for resolution between methods that take both a java primitve and a java boxed type.

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.
- All Java objects are converted to `JObject`.
- Java `Throwable` is converted to `JException` derived from `JObject`.
- Java `String` is converted to `JString` derived from `JObject`.
- Java **arrays** are converted to `JArray` derived from `JObject`.
- Java **boxed** types are converted to `JObject` with extensions of python primitives on return.

Casting

The main problem with exposing Java classes and methods to Python, is that Java allows overloading a method. That is, multiple 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.

Casting using the Java types is also usedful when placing objects in generic containers such as Java List or Map. Both primitive and boxed type Java object derive from the corresponding Python type, so they will work with any Python call.

Boxed types

Both python primitives and Boxed types are immutable. Thus boxed types are inherited from the python primitives. This means that a boxed type regardless of whether produced as a return or created explicitely are treated as python types. They will obey all the conversion rules corresponding to a python type as implicit matches. In addition, they will produce an exact match with their corresponding java type. The type conversion for this is somewhat looser than java. While java provides automatic unboxing of a Integer to a double primitive, jpype can implicitly convert Integer to a Double boxed.

To box a primitive into a specific type such as to place in on a `java.util.List` use `JObject` on the desired boxed type. For example:

```
from jpype.types import *
from jpype import java
# ...
lst = java.util.ArrayList()
lst.add(JObject(JInt(1)))
print(type(lst.get(0)))
```

1.2.8 Implementing interfaces

At times it is necessary to implement an interface in python especially to use classes that require java lambdas. To implement an interface contruct a python class and decorate it with annotations `@JImplements` and `@JOverride`.

```
from jpype import JImplements, JOverride
from java.lang.util import DoubleUnaryOperator
# ...
@JImplements(DoubleUnaryOperator)
class MyImpl(object):
    @JOverride
    def applyAsDouble(self, value):
        return 123+value
```

The java interface may specified by a java wrapper or using a string naming the class. Multiple interfaces can be implemented by a single class by giving a list of interfaces. Alternatively, the interface can be implemented using JProxy.

In a future release, Python callables will be able to automatically match to interfaces that have the Java annotation `@FunctionalInterface`.

1.2.9 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)
```

1.2.10 Java Exceptions

Error handling is an important part of any non-trivial program. All Java exceptions occurring within java code raise a `jpype.JException` which derives from python Exception. These can be caught either using a specific java exception or generically as a `jpype.JException` or `java.lang.Throwable`. You can then use the `stacktrace()`, `str()`, and args to access extended information.

Here is an example:

```
try :
        # Code that throws a java.lang.RuntimeException
except java.lang.RuntimeException as ex:
      print("Caught the runtime exception : ", str(ex))
      print(ex.stacktrace())
```

Multiple java exceptions can be caught together or separately:

```
try:
      #  ...
except (java.lang.ClassCastException, java.lang.NullPointerException) as ex:
      print("Caught multiple exceptions : ", str(ex))
      print(ex.stacktrace())
except java.lang.RuntimeException as ex:
      print("Caught runtime exception : ", str(ex))
      print(ex.stacktrace())
except jpype.JException:
      print("Caught base exception : ", str(ex))
      print(ex.stacktrace())
except Exception as ex:
      print("Caught python exception :", str(ex))
```

Exceptions can be raised in proxies to throw an exception back to java.

Exceptions within the jpype core are issued with the most appropriate python exception type such as `TypeError`, `ValueError`, `AttributeError`, or `OSError`.

Using `jpype.JException` with a class name as a string was supported in previous JPype versions but is currently deprecated.

1.2.11 Customizers

Java wrappers can be customized to better match the expected behavior in python. Customizers are defined using annotations. Currently the annotations `@JImplementionFor` and `@JOverride` can be applied to a regular class to customize an existing class. `@JImplementationFor` requires the class name as a string so that it can be applied to the class before the JVM is started. `@JOverride` can be applied method to hide the java implementation allowing a python functionality to be placed into method. If a java method is overridden it is renamed with an proceeding underscore to appear as a private method. Optional arguments to `@JOverride` can be used to control the renaminging and force the method override to apply to all classes that derive from a base class ("sticky").

Generally speaking, a customizer should be defined before the first instance of a given class is created so that the class wrapper and all instances will have the customization.

Example taken from JPype `java.util.Collection` customizer:

```
@JImplementationFor("java.util.Collection")
class _JCollection(object):

    # Support of len(obj)
    def __len__(self):
        return self.size()

    def __delitem__(self, i):
        return self.remove(i)

    # addAll does not automatically convert to
    # a Collection, so we can augment that
```

(continues on next page)

(continued from previous page)

```
    # behavior here.
    @JOverride(sticky=True)
    def addAll(self, v):
        if isPythonSequence(v):
            r = False
            for i in v:
                r = self.add(i) or r
            return r
        else:
            return self._addAll(v)
```

The name of the class does not matter for the purposes of customizer though it should probabily be a private class so that it does not get used accidentally. The customizer code will steal from the prototype class rather than acting as a base class, thus ensuring that the methods will appear on the most derived python class and are not hidden by the java implementations. The customizer will copy methods, callable objects, `__new__`, class member strings, and properties.

1.2.12 Known limitations

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

Restarting the JVM

JPype caches many resources to the JVM. Those resource are still allocated after the JVM is shutdown as there are still Python objects that point to those resources. If the JVM is restarted, those stale Python objects will be in a broken state and the new JVM instance will obtain the references to these resulting in a memory leak. Thus it is not possible to start the JVM after it has been shutdown with the current implementation.

Running multiple JVM

JPype uses the Python global import module dictionary, a global Python to Java class map, and global JNI typemanager map. These resources are all tied to the JVM that is started or attached. Thus operating more than one JVM does not appear to be possible under the current implementation. Difficulties that would need to be overcome to remove this limitation include:

- Which JVM would a static class method call. Thus the class types would need to be JVM specific (ie. `JClass('org.MyObject', jvm=JVM1)`)
- How would can a wrapper for two different JVM coexist in the `jpype._jclass` module with the same name if different class is required for each JVM.
- How would the user specify which JVM a class resource is created in when importing a module.
- How would objects in one JVM be passed to another.
- How can boxed and String types hold which JVM they will box to on type conversion.

Thus it appears prohibitive to support multiple JVMs in the JPype class model.

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 `startJVM()` 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.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 Python versions

PyPy 2.7 has issues with the Python meta class programming. PyPy 3 appears to work, but does not have very aggressive memory deallocation. Thus PyPy 3 fails the leak test.

Unsupported Java virtual machines

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

Cygwin

Cygwin is currently usable in JPype, but has a number of issues for which there is no current solution. The python2 compilation on cygwin has bugs in a threading implementation that lead to crashes in the test bench. Cygwin does not appear to pass environment variables to the JVM properly resulting in unusual behavior with certain windows calls. The path separator for Cygwin does not match that of the Java dll, thus specification of class paths must account for this. Subject to these issues JPype is usable.

PyPy

The GC routine in PyPy does not play well with Java. It runs when it thinks that Python is running out of resources. Thus a code that allocates a lot of Java memory and deletes the Python objects will still be holding the Java memory until Python is garbage collected. This means that out of memory failures can be issued during heavy operation.

1.2.13 Advanced Topics

Using JPype for debugging Java code

One common use of JPype is not to develop programs in Python, but rather to function as a Read-Eval-Print Loop for Java. When operating Java though Python as a method of developing or debugging Java there are a few tricks that can be used to simplify the job, beyond being able to probe and plot the Java data structures interactively. These methods include:

1) Attaching a debugger to the Java JVM being run under JPype.
2) Attaching debugging information to a Java exception.
3) Serializing the state of a Java process to be evaluated at a later point.

We will briefly discuss each of these methods.

Attaching a Debugger

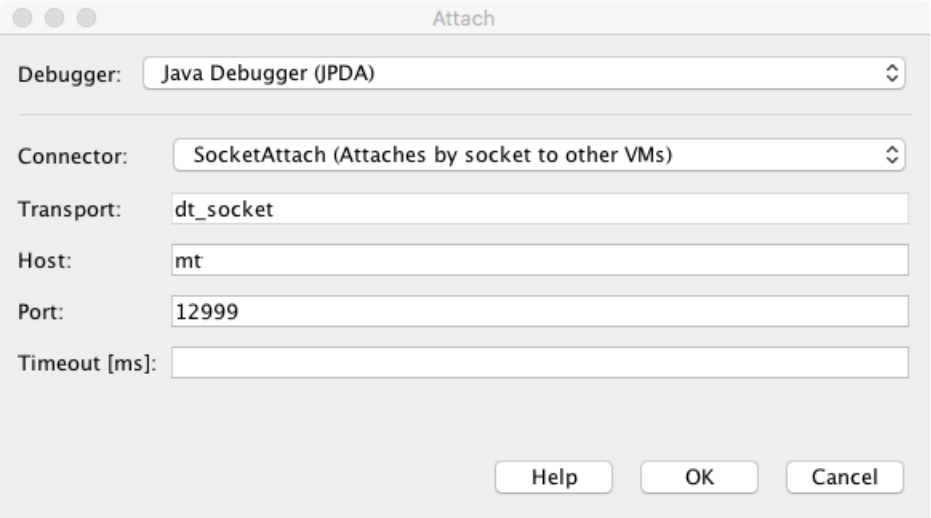
Interacting with Java through a shell is great, but sometimes it is necessary to drop down to a debugger. To make this happen we need to start the JVM with options to support remote debugging.

```
jpype.startJVM("-Xint", "-Xdebug", "-Xnoagent",
  "-Xrunjdwp:transport=dt_socket,server=y,address=12999,suspend=n")
```

Then add a marker in your program when it is time to attach the debugger in the form of a pause statement.

```
input("pause to attach debugger")
myobj.callProblematicMethod()
```

When Python reaches that point during execution, switch to a Java IDE such as Netbeans and select Debug : Attach Debugger. That brings up a window (see example below). After attaching (and setting desired break points) go back to Python and hit enter to continue. Netbeans should come to the foreground when a breakpoint is hit.



Attach data to an Exception

Sometimes getting to the level of a debugger is challenging especially if the code is large and error occurs rarely. In this case it is often benefitial to simply attach data to an exception. To do this, we need to write a small utility class. Java exceptions are not strictly speaking expandable, but they can be chained. Thus, it we create a dummy exception holding a `java.util.Map` and attach it to as the cause of the exception, it will be passed back down the call stack until it reaches Python. We can then use `getCause()` to retrieve the map containing the relevant data.

Capturing the state

If the program is not running in an interactive shell or the program run time is long, we may not want to deal with the problem during execution. In this case, we can serialize the state of the relevant classes and variables. To use this option, we simply make sure all of the classes in Java that we are using are Serializable, then add a condition that detects the faulty algorithm state. When the fault occurs, we create a `java.util.HashMap` and populate it with the values we wish to be able to examine from within Python. We then use Java serialization to write this state file to disk. We then execute the program and collect the resulting state files.

We can then return later with an interactive Python shell, and launch JPype with a classpath for the jars and possibly a connection to debugger. We load the state file into memory and we can then probe or execute the methods that lead up to the fault.

1.3 QuickStart Guide

Quick start quide to using JPype. This quide will show a series of simple examples with the corresponding commands in both Java and Python for using JPype. The JPype *User Guide* and *API Reference* have addition details on the use of the JPype module.

JPype uses two factory classes (`JArray` and `JClass`) to produce class wrappers which can be used to create all Java objects. These serve as both the base class for the corresponding hierarchy and as the factory to produce new wrappers. Casting operators are used to construct specify types of Java types (`JObject`, `JString`, `JBoolean`, `JByte`, `JChar`, `JShort`, `JInt`, `JLong`, `JFloat`, `JDouble`). Two special classes serve as the base classes for exceptions (`JException`) and interfaces (`JInterface`). There are a small number of support methods to help in controlling the JVM. Last, there are a few annotations used to create customized wrappers.

For the purpose of this guide, we will assume that the following classes were defined in Java. We will also assume the reader knows enough Java and Python to be dangerous.

```
package org.pkg;

publc class BassClass
{
   public callMember(int i)
   {}
}

public class MyClass extends BaseClass
{
   final public static int CONST_FIELD = 1;
   public static int staticField = 1;
   public int memberField = 2;
   int internalField =3;

   public MyClass() {}
   public MyClass(int i) {}

   public static void callStatic(int i) {}
   public void callMember(int i) {}

   // Python name conflict
   public void pass() {}

   public void throwsException throws java.lang.Exception {}
```

(continues on next page)

(continued from previous page)

```
    // Overloaded methods
    public call(int i) {}
    public call(double d) {}
}
```

1.3.1 Starting JPype

The hardest thing about using JPype is getting the jars loaded into the JVM. Java is curiously unfriendly about reporting problems when it is unable to find a jar. Instead, it will be reported as an `ImportError` in python. These patterns will help debug problems regarding jar loading.

Once the JVM is started Java packages that are within a top level domain (TLD) are exposed as python modules allowing Java to be treated as part of python.

Description	Java	Python

Start Java Virtual Machine (JVM)		`# Import module`
		`import jpype`
		
		`# Enable Java imports`
		`import jpype.imports`
		
		`# Pull in types`
		`from jpype.types import *`
		
		`# Launch the JVM`
		`jpype.startJVM()`

Import default Java namespace[1]		`import java.lang`
Add a set of jars from a directory[2]		`jpype.addClassPath("/my/path/*")`
Add a specific jar to the classpath[2]		`jpype.addClassPath('/my/path/myJar.jar')`
Print JVM CLASSPATH[3]		`from java.lang import System` `print(System.getProperty("java.class.path"))`

[1] All `java.lang.*` classes are available.
[2] Must happen prior to starting the JVM
[3] After JVM is started

1.3.2 Classes/Objects

Java classes are presented whereever possible exactly like Python classes. The only major difference is that Java classes and objects are closed and cannot be modified. As Java is strongly typed, casting operators are used to select specific overloads when calling methods. Classes are either imported using as a module or loaded with the `JClass` factory.

Description	Java	Python
Import a class[4]	`import org.pkg.MyClass`	`from org.pkg import MyClass`
Import a class and rename[4]		`from org.pkg import MyClass as OurClass`
Import multiple classes from a package[5]		`from org.pkg import MyClass, AnotherClass`
Import a java package for long name access[6]		`import org.pkg`
Import a class static[7]	`import org.pkg.MyClass.CONST_FIELD`	`from org.pkg.MyClass import CONST_FIELD`
Import a class without tld[8]	`import zippy.NonStandard`	`NonStandard = JClass('zippy.NonStandard')`
Construct an object	`MyClass myObject = new MyClass(1);`	`myObject = MyClass(1)`
Constructing a cless with full class name		`import org.pkg` `myObject = org.pkg.MyClass(args)`
Get a static field	`int var = MyClass.staticField;`	`var = MyClass.staticField`
Get a member field	`int var = myObject.memberField;`	`var = myObject.memberField`
Set a static field[9]	`MyClass.staticField = 2;`	`MyClass.staticField = 2`
Set a member field[9]	`myObject.memberField = 2;`	`myObject.memberField = 2`
Call a static method	`MyClass.callStatic(1);`	`MyClass.callStatic(1)`
Call a member method	`myObject.callMember(1);`	`myObject.callMember(1)`
Access member with python naming conflict[10]	`myObject.pass()`	`myObject.pass_()`
Checking inheritance		
	`if (obj instanceof MyClass) {...}`	`if (isinstance(obj, MyClass): ...`
Checking if Java class wrapper		

1.3.3 Exceptions

Java exceptions extend from python exceptions and can be dealt with no different that Python native exceptions. JException serves as the base class for all Java exceptions.

Description	Java	Python

Catch an exception	`try {`
	`  myObject.`
	`↪throwsException();`
	`} catch (java.lang.`
	`↪Exception ex)`
	`{ ... }`
		`try:`
		`  myObject.`
		`↪throwsException()`
		`except java.lang.`
		`↪Exception as ex:`
		`  ...`

Throw an exception to Java	`throw new java.lang.`
	`↪Exception(`
	`  "Problem");`
		`raise java.lang.Exception(`
		`  "Problem")`

Checking if Java exception wrapper		`if (isinstance(obj,␣` `↪JException): ...`

Closeable items	`try (InputStream is`
	`  = Files.`
	`↪newInputStream(file))`
	`{ ... }`
		`with Files.`
		`↪newInputStream(file) as␣`
		`↪is:`
		`  ...`

1.3.4 Primitives

Most python primitives directly map into Java primitives. However, python does not have the same primitive types, thus sometimes it is necessary to cast to a specific Java primitive type especially if there are Java overloads that would otherwise be in conflict. Each of the Java types are exposed in JPype (`JBoolean`, `JByte`, `JChar`, `JShort`, `JInt`, `JLong`, `JFloat`, `JDouble`).

Python int is equivalent to Java long.

[4] This will report an error if the class is not found.
[5] This will report an error if the classes are not found
[6] Does not report errors if the package is invalid
[7] Constants, static fields, and static methods can be imported.
[8] `JClass` loads any class by name including inner classes.
[9] Produces error for final fields
[10] Underscore is added during wrapping.

Description	Java	Python
Casting to hit an overload[11]	`myObject.call((int)v);`	`myObject.call(JInt(v))`
Create a primitive array	`int[] array = new int[5]`	`array = JArray(JInt)(5)`
Create an initialized primitive array[12]	`int[] array = new int[]{1, ↪2,3}`	`array = JArray(JInt)([1,2, ↪3])`
Put a specific primitive type on a list	`List<Integer> myList = new ArrayList<>(); myList.add(1);`	`from java.util import␣ ↪ArrayList myList = ArrayList() myList.add(JInt(1))`
Boxing a primitive[13]	`Integer boxed = 1;`	`boxed = JObject(JInt(1))`

1.3.5 Strings

Java strings are similar to python strings. They are both immutable and produce a new string when altered. Most operations can use Java strings in place of python strings, with minor exceptions as python strings are not completely duck typed. When comparing or using as dictionary keys JString should be converted to python.

[11] `JInt` acts as a casting operator
[12] list, sequences, or np.array can be used to initialize.
[13] `JInt` specifies the prmitive type. `JObject` boxes the primitive.

Description	Java	Python
Create a Java string[14]	`String javaStr = new String("foo");`	`myStr = JString("foo")`
Create a Java string from bytes[15]	`byte[] b;` `String javaStr = new String(b, "UTF-8");`	`b= b'foo'` `myStr = JString(b, "UTF-8")`
Converting Java string		`str(javaStr)`
Comparing Python and Java strings[16]		`str(javaStr) == pyString`
Comparing Java strings	`javaStr.equals("foo")`	`javaStr == "foo"`
Checking if java string		`if (isinstance(obj, JString): ...`

1.3.6 Arrays

Arrays are create using JArray class factory. They operate like python lists, but they are fixed in size.

[14] `JString` constructs a `java.lang.String`
[15] All `java.lang.String` constuctors work.
[16] `str()` converts the object for comparison

Description	Java	Python
Create a single dimension array	`MyClass[] array = new MyClass[5];`	`array = JArray(MyClass)(5)`
Create a multi dimension array	`MyClass[][] array2 = new MyClass[5][];`	`array2 = JArray(MyClass, 2)(5)`
Access an element	`array[0] = new MyClass()`	`array[0] = MyClass()`
Size of an array	`array.length`	`len(array)`
Convert to python list		`pylist = list(array)`
Iterate elements	`for (MyClass element: array) {...}`	`for element in array: ...`
Checking if java array wrapper		`if (isinstance(obj, JArray): ...`

1.3.7 Collections

Java standard containers are available and are overloaded with python syntax where possible to operate in a similar fashion to python objects. It is not currently possible to specify the template types for generic containers, but that will be introduced in Java 9.

Description	Java	Python
Import list type	`import java.util.ArrayList;`	`from java.util import ArrayList`
Construct a list	`List<Integer> myList=new ArrayList<>();`	`myList=ArrayList()`
Get length of list	`int sz = myList.size();`	`sz = len(myList)`
Get list item	`Integer i = myList.get(0)`	`i = myList[0]`
Set list item[17]	`myList.set(0, 1)`	`myList[0]=Jint(1)`
Iterate list elements	`for (Integer element: myList) {...}`	`for element in myList: ...`
Import map type	`import java.util.HashMap;`	`from java.util import HashMap`
Construct a map	`Map<String,Integer> myMap=new HashMap<>();`	`myMap=HashMap()`
Get length of map	`int sz = myMap.size();`	`sz = len(myMap)`
Get map item	`Integer i = myMap.get("foo")`	`i = myMap["foo"]`
Set map item[17]	`myMap.set("foo", 1)`	`myMap["foo"]=Jint(1)`
Iterate map entries	`for (Map.Entry<String, Integer> e : myMap.entrySet()) {...}`	`for e in myMap.entrySet(): ...`

[17] Casting is required to box primitives to the correct type.

1.3.8 Reflection

For operations that are outside the scope of the JPype syntax, Using Java reflection, any Java operation include calling a specific overload or even accessing private methods and fields.

Description	Java	Python
Access Java reflection class	`MyClass.class`	`MyClass.class_`

Access a private field by name		`cls = myObject.class_`
		`field = cls.`
		`↪getDeclaredField(`
		`"internalField")`
		`field.setAccessible(True)`
		`field.get()`

Accessing a specific overload[18]		`cls = MyClass.class_`
		`cls.getDeclaredMethod(`
		`↪"call", JInt)`
		`cls.invoke(myObject,␣`
		`↪JInt(1))`

Convert a `java.lang.Class` into python wrapper[19]		`# Something returned a␣`
		`↪java.lang.Class`
		`MyClassJava =␣`
		`↪getClassMethod()`
		
		`# Convert to it to Python`
		`MyClass =␣`
		`↪JClass(myClassJava)`

Load a class with a external class loader	`ClassLoader cl`
	`= new␣`
	`↪ExternalClassLoader();`
	`Class cls`
	`= Class.forName(`
	`↪"External",`
	`True,␣`
	`↪cl)`
		`cl = ExternalClassLoader()`
		`cls = JClass("External",␣`
		`↪loader=cl)`

Accessing base method implementation		`from org.pkg import \`
		`BaseClass, MyClass`
		`myObject = MyClass(1)`
		`BaseClass.`
		`↪callMember(myObject, 2)`

[18] types must be exactly specified.
[19] Rarely required unless the class was supplied external such as generics.

1.3.9 Implements and Extension

JPype can implement a Java interface by annotating a python class. Each method that is required must be implemented.

JPype does not support extending a class directly in python. Where it is necessary to exend a Java class, it is required to create a Java extension with an interface for each methods that are to be accessed from python. For some deployments this may be be an option. If that is the case, the JPype inline compiler can be used to create the dynamic class on the fly.

Description	Java	Python
Implement an interface	`public class PyImpl implements MyInterface { public void call() {...} }`	`@JImplements(MyInterface) class PyImpl(object): @JOverride def call(self): pass`
Extending classes[20]		None
Lambdas[20]		None

Don't like the formatting? Feel the guide is missing something? Submit a pull request at the project page.

1.4 API Reference

1.4.1 JVM Functions

These functions control and start the JVM.

jpype.**startJVM**(**args*, ***kwargs*)

Starts a Java Virtual Machine. Without options it will start the JVM with the default classpath and jvmpath.

The default classpath is determined by `jpype.getClassPath()`. The default jvmpath is determined by `jpype.getDefaultJVMPath()`.

Parameters ***args** (*Optional, str[]*) – Arguments to give to the JVM. The first argument may be the path the JVM.

Keyword Arguments

- **jvmpath** (*str*) – Path to the jvm library file, Typically one of (`libjvm.so`, `jvm.dll`, ...) Using None will apply the default jvmpath.
- **classpath** (*str, [str]*) – Set the classpath for the jvm. This will override any classpath supplied in the arguments list. A value of None will give no classpath to JVM.
- **ignoreUnrecognized** (*bool*) – Option to JVM to ignore invalid JVM arguments. Default is False.
- **convertStrings** (*bool*) – Option to JPype to force Java strings to cast to Python strings. This option is to support legacy code for which conversion of Python strings was the default. This will globally change the behavior of all calls using strings, and a value of True is NOT recommended for newly developed code.

[20] Support for use of python function as Java 8 lambda is WIP.

The default value for this option during 0.7 series is True. The option will be False starting in 0.8. A warning will be issued if this option is not specified during the transition period.

Raises

- `OSError` – if the JVM cannot be started or is already running.
- `TypeError` – if an invalid keyword argument is supplied or a keyword argument conflicts with the arguments.

`jpype.`**`shutdownJVM`**`()`
Shuts down the JVM.

This method shuts down the JVM and thus disables access to existing Java objects. Due to limitations in the JPype, it is not possible to restart the JVM after being terminated.

`jpype.`**`getDefaultJVMPath`**`()`
Retrieves the path to the default or first found JVM library

Returns The path to the JVM shared library file

Raises

- `JVMNotFoundException` – If there was no JVM found in the search path.
- `JVMNotSupportedException` – If the JVM was found was not compatible with Python due to cpu architecture.

`jpype.`**`getClassPath`**`(`*env=True*`)`
Get the full java class path.

Includes user added paths and the environment CLASSPATH.

Parameters **`env`** (*`Optional, bool`*) – If true then environment is included. (default True)

1.4.2 Class importing

JPype supports several styles of importing. The newer integrated style is provided by the *imports* module. The older `JPackage` method is available for accessing package trees with less error checking. Direct loading of Java classes can be made with *JClass*.

For convenience, the JPpype module predefines the following `JPackage` instances for `java` and `javax`.

`class` `jpype.`**`JPackage`**`(`*name*, *strict=False*, *pattern=None*`)`
Gateway for automatic importation of Java classes.

This class allows structured access to Java packages and classes. This functionality has been replaced by `jpype.imports`, but is still useful in some cases.

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
```

Under some situations such as a missing jar the resulting object will be a JPackage object rather than the expected java class. This results in rather challanging debugging messages. Thus the `jpype.imports` module is preferred. To prevent these types of errors a package can be declares as `strict` which prevents expanding package names that do not comply with Java package name conventions.

Parameters

- **path** (*str*) – Path into the Java class tree.
- **strict** (*bool, optional*) – Requires Java paths to conform to the Java package naming convention. If a path does not conform and a class with the required name is not found, the AttributeError is raise to indicate that the class was not found.

Example

```
# Alias into a library
google = JPackage("com.google")

# Access members in the library
result = google.common.IntMath.pow(x,m)
```

1.4.3 Class Factories

class jpype.**JClass**(**args*, ***kwargs*)

Meta class for all java class instances.

JClass when called as an object will contruct a new java Class wrapper.

All python wrappers for java classes derived from this type. To test if a python class is a java wrapper use `isinstance(obj, jpype.JClass)`.

Parameters **className** (*str*) – name of a java type.

Keyword Arguments

- **loader** (*java.lang.ClassLoader*) – specifies a class loader to use when creating a class.
- **initialize** (*bool*) – Passed to class loader when loading a class using the class loader.

Returns a new wrapper for a Java class

Return type JavaClass

Raises `TypeError` – if the component class is invalid or could not be found.

class jpype.**JArray**(**args*, ***kwargs*)

class jpype.**JException**(**args*, ***kwargs*)

Base class for all `java.lang.Throwable` objects.

When called as an object `JException` will produce a new exception class. The arguments may either be a string or an existing Java throwable. This functionality is deprecated as exception classes can be created with `JClass`.

Use `issubclass(cls, JException)` to test if a class is derived from `java.lang.Throwable`.

Use `isinstance(obj, JException)` to test if an object is a `java.lang.Throwable`.

1.4.4 Java Types

JPype has types for each of the Java primitives: `JBoolean`, `JByte`, `JShort`, `JInt`, `JLong`, `JFloat` and `JDouble`. In addition there is one class for working with Java objects, `JObject`. These serve to be able to cast to a specified type and specify types with the `JArray` factory. There is a `JString` type provided for convenience when creating or casting to strings.

class jpype.**JObject**(**args*)

Base class for all object instances.

It can be used to test if an object is a java object instance with `isinstance(obj, JObject)`.

Calling `JObject` as a function can be used to covert or cast to specific Java type. It will box primitive types and supports an option type to box to.

This wrapper functions four ways.

- If the no type is given the object is automatically cast to type best matched given the value. This can be used to create a boxed primitive. `JObject(JInt(i))`
- If the type is a primitve, the object will be the boxed type of that primitive. `JObject(1, JInt)`
- If the type is a Java class and the value is a Java object, the object will be cast to the Java class and will be an exact match to the class for the purposes of matching arguments. If the object is not compatible, an exception will be raised.
- If the value is a python wrapper for class it will create a class instance. This is aliased to be much more obvious as the `class_` member of each Java class.

Parameters

- **value** – The value to be cast into an Java object.
- **type** (*Optional, type*) – The type to cast into.

Raises `TypeError` – If the object cannot be cast to the specified type, or the requested type is not a Java class or primitive.

class jpype.**JString**(**args*)

Base class for `java.lang.String` objects

When called as a function, this class will produce a `java.lang.String` object. It can be used to test if an object is a Java string using `isinstance(obj, JString)`.

1.4.5 Threading

jpype.**synchronized**(*obj*)

Creates a resource lock for a Java object.

Produces a monitor object. During the lifespan of the monitor the Java will not be able to acquire a thread lock on the object. This will prevent multiple threads from modifying a shared resource.

This should always be used as part of a Python `with` startment.

Parameters **obj** – A valid Java object shared by multiple threads.

Example:

```
with synchronized(obj):
    # modify obj values

# lock is freed when with block ends
```

jpype.**isThreadAttachedToJVM**()

Checks if a thread is attached to the JVM.

Python automatically attaches threads when a Java method is called. This creates a resource in Java for the Python thread. This method can be used to check if a Python thread is currently attached so that it can be disconnected prior to thread termination to prevent leaks.

Returns True if the thread is attached to the JVM, False if the thread is not attached or the JVM is not running.

jpype.**attachThreadToJVM**()
Attaches a thread to the JVM.

The function manually connects a thread to the JVM to allow access to Java objects and methods. JPype automaticatlly attaches when a Java resource is used, so a call to this is usually not needed.

Raises `RuntimeError` – If the JVM is not running.

jpype.**detachThreadFromJVM**()
Detaches a thread from the JVM.

This function detaches the thread and frees the associated resource in the JVM. For codes making heavy use of threading this should be used to prevent resource leaks. The thread can be reattached, so there is no harm in detaching early or more than once. This method cannot fail and there is no harm in calling it when the JVM is not running.

1.4.6 Decorators

JPype uses ordinary Python classes to implement functionality in Java. Adding these decorators to a Python class will mark them for use by JPype to interact with Java classes.

1.4.7 Proxies

JPype can implement Java interfaces either using decorators or by manually creating a JProxy. Java only support proxying interfaces, thus we cannot extend an existing Java class.

class jpype.**JProxy**(*intf*, *dict=None*, *inst=None*)
Define a proxy for a Java interface.

This is an older style JPype proxy interface that uses either a dictionary or an object instance to implement methods defined in java. The python object can be held by java and its lifespan will continue as long as java holds a reference to the object instance. New code should use `@JImplements` annotation as it will support improved type safety and error handling.

Name lookups can either made using a dictionary or an object instance. One of these two options must be specified.

Parameters

- **intf** – either a single interface or a list of java interfaces. The interfaces can either be defined by strings or JClass instance. Only interfaces may be used in a proxy,
- **dict** (*dict[string, callable], optional*) – specifies a dictionary containing the methods to be called when executing the java interface methods.
- **inst** (*object, optional*) – specifies an object with methods whose names matches the java interfaces methods.

1.4.8 Customized Classes

JPype provides standard customizers for Java interfaces so that Java objects have syntax matching the corresponding Python objects. The customizers are automatically bound the class on creation without user intervention. We are documentating the functions that each customizer adds here.

These internal classes can be used as example of how to implement your own customizers for Java classes.

class jpype._jcollection.**_JIterable**
Customizer for `java.util.Iterable`

This customizer adds the Python iterator syntax to classes that implement Java Iterable.

class jpype._jcollection.**_JCollection**
Customizer for `java.util.Collection`

This customizer adds the Python functions `len()` and `del` to Java Collions to allow for Python syntax.

class jpype._jcollection.**_JList**
Customizer for `java.util.List`

This customizer adds the Python list operator to function on classes that implement the Java List interface.

class jpype._jcollection.**_JMap**
Customizer for `java.util.Map`

This customizer adds the Python list and len operators to classes that implement the Java Map interface.

class jpype._jcollection.**_JIterator**
Customizer for `java.util.Iterator`

This customizer adds the Python iterator concept to classes that implement the Java Iterator interface.

class jpype._jcollection.**_JEnumeration**
Customizer for `java.util.Enumerator`

This customizer adds the Python iterator concept to classes that implement the Java Enumerator interface.

class jpype._jio.**_JCloseable**
Customizer for `java.io.Closable`

This customizer adds support of the *with* operator to all Java classes that implement Java Closable interface.

Example:

```
from java.nio.files import Files, Paths
with Files.newInputStream(Paths.get("foo")) as fd:
  # operate on the input stream

# Input stream closes at the end of the block.
```

1.4.9 Modules

Optional JPype behavior is stored in modules. These optional modules can be imported to add additional functionality.

JPype Imports Module

Once imported this module will place the standard TLDs into the python scope. These tlds are `java`, `com`, `org`, and `gov`. Java symbols from these domains can be imported using the standard Python syntax.

Import customizers are supported in Python 3.6 or greater.

Forms supported:

- **import <java_pkg> [as <name>]**
- **import <java_pkg>.<java_class> [as <name>]**
- **from <java_pkg> import <java_class>[,<java_class>*]**

- **from <java_pkg> import <java_class> [as <name>]**
- **from <java_pkg>.<java_class> import <java_static> [as <name>]**
- **from <java_pkg>.<java_class> import <java_inner> [as <name>]**

For further information please read the *JImport* guide.

Requires: Python 2.7 or 3.6 or later

Example:

```
import jpype
import jpype.imports
jpype.startJVM()

# Import java packages as modules
from java.lang import String
```

jpype.imports.**registerDomain**(*mod*, *alias=None*)
Add a java domain to python as a dynamic module.

This can be used to bind a Java path to a Python path.

Parameters

- **mod** (*str*) – Is the Python module to bind to Java.
- **alias** (*str, optional*) – Is the name of the Java path if different than the Python name.

jpype.imports.**registerImportCustomizer**(*customizer*)
Import customizers can be used to import python packages into java modules automatically.

class jpype.imports.**JImportCustomizer**
Base class for Import customizer.

Import customizers should implement canCustomize and getSpec.

Example:

```
# Site packages for each java package are stored under $DEVEL/<java_pkg>/py
class SiteCustomizer(jpype.imports.JImportCustomizer):
    def canCustomize(self, name):
        if name.startswith('org.mysite') and name.endswith('.py'):
            return True
        return False
    def getSpec(self, name):
        pname = name[:-3]
        devel = os.environ.get('DEVEL')
        path = os.path.join(devel, pname,'py','__init__.py')
        return importlib.util.spec_from_file_location(name, path)
```

JPype Beans Module

This customizer finds all occurances of methods with get or set and converts them into Python properties. This behavior is sometimes useful in programming with JPype with interactive shells, but also leads to a lot of confusion. Is this class exposing a variable or is this a property added JPype. It was the default behavior until 0.7.

As an unnecessary behavior that violates both the Python principle *"There should be one– and preferably only one –obvious way to do it."* and the C++ principle *"You only pay for what you use"*. Thus this misfeature was removed

from the distribution as a default. However, given that it is at times useful to have methods appear as properties, it was moved to a an optional module.

To use beans as properties:

```
import jpype.beans
```

The beans property modification is a global behavior and applies retroactively to all classes currently loaded. Once started it can never be undone.

JPype Types Module

Optional module containing only the Java types and factories used by JPype. Classes in this module include `JArray`, `JClass`, `JBoolean`, `JByte`, `JChar`, `JShort`, `JInt`, `JLong`, `JFloat`, `JDouble`, `JString`, `JObject`, and `JException`.

Example

```
from jpype.types import *
```

1.5 JImport

Module for dynamically loading Java Classes using the import system.

This is a replacement for the jpype.JPackage("com").fuzzy.Main type syntax. It features better safety as the objects produced are checked for class existence. To use java imports, import the domains package prior to importing a java class.

This module supports three different styles of importing java classes.

1.5.1 1) Import of the package path

import <java_package_path>

Importing a series of package creates a path to all classes contained in that package. It does not provide access the the contained packages. The root package is added to the global scope. Imported packages are added to the directory of the base module.

```
import java.lang      # Adds java as a module
import java.util

mystr = java.lang.String('hello')
mylist = java.util.LinkedList()
path = java.nio.files.Paths.get() # ERROR java.nio.files not imported
```

1.5.2 2) Import of the package path as a module

import <java_package> as <var>

A package can be imported as a local variable. This provides access to all java classes in that package. Contained packages are not available.

Example:

```
import java.nio as nio
bb = nio.ByteBuffer()
path = nio.file.Path()    # ERROR subpackages file must be imported
```

1.5.3 3) Import a class from an object

from <java_package> import <class>[,<class>*] [as <var>]

An individual class can be imported from a java package. This supports inner classes as well.

Example:

```
# Import one class
from java.lang import String
mystr = String('hello')

# Import multiple classes
from java.lang import Number,Integer,Double
# Import java inner class java.lang.ProcessBuilder.Redirect
from java.lang.ProcessBuilder import Redirect
```

This method can also be used to import a static variable or method from a class.

1.5.4 Import caveats

Wild card Imports

Wild card imports for classes will import all static method and fields into the global namespace. They will also import any inner classes that have been previously be accessed.

Wild card importation of package symbols are not currently supported and have unpredictable effects. Because of the nature of class loaders it is not possible to determine what classes are currently loaded. Some classes are loaded by the boot strap loader and thus are not available for discovery.

As currently implemented [from <java_package> import *] will import all classes and static variables which have already been imported by another import call. As a result which classes will be imported is based on the code pat and thus very unreliable.

It is possible to determine the classes available using Guava for java extension jars or for jars specifically loaded in the class path. But this is sufficiently unreliable that we recommend not using wildcards for any purpose.

Keyword naming

Occasionally a java class may contain a python keyword. Python keywords as automatically remapped using trailing underscore.

Example:

```
from org.raise_ import Object  => imports "org.raise.Object"
```

Controlling Java package imports

By default domains imports four top level domains (TLD) into the python import system (com, gov, java, org). Additional domains can be added by calling registerDomain. Domains can be an alias for a java package path.

Example:

```
domains.registerDomain('jname')
from jname.framework import FrameObject
domains.registerDomain('jlang', alias='java.lang')
from jlang import String
```

Limitations

- Wildcard imports are unreliable and should be avoided. Limitations in the Java specification are such that there is no way to get class information at runtime. Python does not have a good hook to prevent the use of wildcard loading.
- Non-static members can be imported but can not be called without an instance. Jpype does not provide an easy way to determine which functions objects can be called without an object.

1.6 Changelog

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

- **Next version - unreleased**
  - Corrected segfault when converting null elements while accessing a slice from a Java object array.
- **0.7.0 - 2019** - Doc strings are generated for classes and methods.
  - Complete rewrite of the core module code to deal unattached threads, improved hardening, and member management. Massive number of internal bugs were identified during the rewrite and corrected. See the ChangeLog-0.7 for details of all changes.
  - API breakage:
    - Java strings conversion behavior has changed. The previous behavior was switchable, but only the default convert to Python was working. Converting to automatically lead to problems in which is was impossible to work with classes like StringBuilder in Java. To convert a Java string use `str()`. Therefore, string conversion is currently selected by a switch at the start of the JVM. The default shall be False starting in JPype 0.8. New code is encouraged to use the future default of False. For the transition period the default will be True with a warning if not policy was selected to encourage developers to pick the string conversion policy that best applies to their application.
    - Java exceptions are now derived from Python exception. The old wrapper types have been removed. Catch the exception with the actual Java exception type rather than `JException`.
    - Undocumented exceptions issued from within JPype have been mapped to the corresponding Python exception types such as `TypeError` and `ValueError` appropriately. Code catching exceptions from previous versions should be checked to make sure all exception paths are being handled.
    - Undocumented property import of Java bean pattern get/set accessors was removed as the default. It is available with `import jpype.beans`, but its use is discouraged.
  - API rework:
    - JPype factory methods now act as base classes for dynamic class trees.

    - Static fields and methods are now available in object instances.
    - Inner classes are now imported with the parent class.
    - `jpype.imports` works with Python 2.7.
    - Proxies and customizers now use decorators rather than exposing internal classes. Existing `JProxy` code still works.
    - Decorator style proxies use `@JImplements` and `@JOverload` to create proxies from regular classes.
    - Decorator style customizers use `@JImplementionFor`
    - Module `jpype.types` was introduced containing only the Java type wrappers. Use `from jpype.types import *` to pull in this subset of JPype.
  - `synchronized` using the Python `with` statement now works for locking of Java objects.
  - Previous bug in initialization of arrays from list has been corrected.
  - Added extra verbiage to the to the raised exception when an overloaded method could not be matched. It now prints a list of all possible method signatures.
  - The following is now DEPRECATED
    - `jpype.reflect.*` - All class information is available with `.class_`
    - Unncessary `JException` from string now issues a warning.
  - The followind is now REMOVED
    - Python thread option for `JPypeReferenceQueue`. References are always handled with with the Java cleanup routine. The undocumented `setUsePythonThreadForDaemon()` has been removed.
    - Undocumented switch to change strings from automatic to manual conversion has been removed.
    - Artifical base classes `JavaClass` and `JavaObject` have been removed.
    - Undocumented old style customizers have been removed.
    - Many internal jpype symbols have been removed from the namespace to prevent leakage of symbols on imports.
  - promoted *'–install-option'* to a *'–global-option'* as it applies to the build as well as install.
  - Added *'–enable-tracing'* to setup.py to allow for compiling with tracing for debugging.
  - Ant is required to build jpype from source, use `--ant=` with setup.py to direct to a specific ant.
- **0.6.3 - 2018-04-03**
  - Java reference counting has been converted to use JNI PushLocalFrame/PopLocalFrame. Several resource leaks were removed.
  - `java.lang.Class<>.forName()` will now return the java.lang.Class. Work arounds for requiring the class loader are no longer needed. Customizers now support customization of static members.
  - Support of `java.lang.Class<>`
    - `java.lang.Object().getClass()` on Java objects returns a java.lang.Class rather than the Python class
    - `java.lang.Object().__class__` on Java objects returns the python class as do all python objects

    - `java.lang.Object.class_` maps to the java statement `java.lang.Object.class` and returns the `java.lang.Class<java.lang.Object>`
    - java.lang.Class supports reflection methods
    - private fields and methods can be accessed via reflection
    - annotations are avaiable via reflection
  - Java objects and arrays will not accept setattr unless the attribute corresponds to a java method or field whith the exception of private attributes that begin with underscore.
  - Added support for automatic conversion of boxed types.
    - Boxed types automatically convert to python primitives.
    - Boxed types automatically convert to java primitives when resolving functions.
    - Functions taking boxed or primitives still resolve based on closest match.
  - Python integer primitives will implicitly match java float and double as per Java specification.
  - Added support for try with resources for `java.lang.Closeable`. Use python "with MyJavaResource() as resource:" statement to automatically close a resource at the end of a block.
- **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

1.7 Developer Guide

1.7.1 Overview

This document describes the guts of jpype. It is intended lay out the architecture of the jpype code to aid intrepid lurkers to develop and debug the jpype code once I am run over by a bus. For most of this document I will use the royal we, except where I am giving personal opinions expressed only by yours truly, the author Thrameos.

History

When I started work on this project it had already existed for over 10 years. The original developer had intended a much larger design with modules to support multiple languages such as Ruby. As such it was constructed with three layers of abstraction. It has a wrapper layer over Java in C++, a wrapper layer for the Python api in C++, and an abstraction layer intended to bridge Python and other interpreted languages. This multilayer abstraction ment that every debugging call had to drop through all of those layers. Memory management was split into multiple pieces with Java controlling a portion of it, C++ holding a bunch of resources, Python holding additional resources, and HostRef controlling the lifetime of objects shared between the layers. It also had its own reference counting system for handing Java references on a local scale.

This level of complexity was just about enough to scare off all but the most hardened programmer. Thus I set out to eliminate as much of this as I could. Java already has its own local referencing system to deal in the form of LocalFrames. It was simiply a matter of setting up a C++ object to hold the scope of the frames to eliminate that layer. The Java abstraction was laid out in a fashion somewhat orthagonally to the Java inheritance diagram. Thus that was reworked to something more in line which could be safely completed without disturbing other layers. The multilanguage abstraction layer was already pierced in multiple ways for speed. However, as the abastraction interwove throughout all the library it was a terrible lift to remove and thus required gutting the Python layer as well to support the operations that were being performed by the HostRef.

The remaining codebase is fairly slim and reasonably streamlined. This rework cut out about 30% of the existing code and sped up the internal operations. The Java C++ interface matches the Java class hierachy.

Architecture

JPype is split into several distinct pieces.

`jpype` Python module The majority of the front end logic for the toolkit is in Python jpype module. This module deals with the construction of class wrappers and control functions. The classes in the layer are all prefixed by `J`.

`_jpype` CPython module The native module is supported by a CPython module called `_jpype`. The `_jpype` module is located in `native/python` and has C style classes with a prefix `PyJP`.

This CPython layer acts as a front end for passing to the C++ layer. It performs some error checking. In addition to the module functions in `PyJPModule`, the module has multiple Python classes to support the native jpype code such as `PyJPClass`, `PyJPArray`, `PyJPValue`, `PyJPValue`, etc.

CPython API wrapper In addition to the exposed Python module layer, there is also a C++ wrapper for the Python API. This is located in `native/python` and has the prefix `JPPy` for all classes.

There are two parts of this api wrapper, `jp_pythonenv` and `jp_pythontypes`. The `jp_pythonenv` holds all of the resources that C++ needs to communicate with the jpype native module. `jp_pythontypes` wraps the required parts of the CPython API in C++ for use in the C++ layer.

C++ JNI layer The guts that drive Java are in the C++ layer located in `native/common`. This layer has the namespace `JP`. The code is divided into wrappers for each Java type, a typemanager for mapping from Java names to class instances, support classes for proxies, and a thin JNI layer used to help ensure rigerous use of the same design patterns in the code. The primary responsibility of this layer is type conversion and matching of method overloads.

Java layer In addition to the C++ layer, jpype has a native Java layer. This code is compiled as a "thunk" which is loaded into the JVM in the form of a a binary stored as a string. Code for Java is found in `native/java`. The Java layer is divided into two parts, a bootstrap loader and a jar containing the support classes. The Java layer is responsible managing the lifetime of shared Python, Java, and C++ objects.

1.7.2 `jpype` module

The `jpype` module itself is made of a series of support classes which act as factories for the individual wrappers that are created to mirror each Java class. Because it is not possible to wrap all Java classes with staticly created wrappers, instead jpype dynamically creates Python wrappers as requested by the user.

The wrapping process is triggered in two ways. The user can manually request creating a class by importing a class wrapper with jpype.imports or `JPackage` or by manually invoking it with `JClass`. Or the class wrapper can be created automatically as a result of a return type or exception thrown to the user.

Because the classes are created dynamically, the class structure uses a lot of Python meta programming. As the programming model for Python 2.7 and 3 series are rather different, the exact formuation is restricted to a set of common formulations that are shared between the versions.

Each class wrapper derives from the class wrappers of each of the wrappers corresponding to the Java classes that each class extends and implements. The key to this is to hacked `mro`. The `mro` orders each of the classes in the tree such that the most drived class methods are exposed, followed by each parent class. This must be ordered to break ties resulting from multiple inheritance of interfaces.

The `mro` has one aspect that breaks the Python object model. Normally it is a requirement that every class that inherits from another class must inherit all of the previous parents. However, Java has two distinct types of inheritance, extension and implementation. As such we delete the `JInterface` parent from all concrete class implementation during the mro resolution phase.

resource types

jpype objects work with the inner layers primarily through duck typing using a series of special fields. These fields correspond to a JNI type such as `jvalue`, `jclass`, `jobject`, and `jstring`. But as these resources cannot be held directly, they are provided as resources exposed as `_jpype.PyJP` classes.

`jvalue`

In the earlier design, wrappers, primitives and objects were all seperate concepts. At the JNI layer these are unified by a common element called jvalue. A `jvalue` is a union of all primitives with the jobject. The jobject can represent anything derived from Java object including the pseudo class jstring. To represent these in a common way all jpype objects that represent an instance of a Java resource have a `__javavalue__` field which should be of type `_jpype.PyJPValue`. Rather than forcing checking of individual types, we use duck typing of simply checking for a `PyJPValue` in this field. In addition to the union, the jvalue also carries a hidden type. Thus hidden type is used to help in casting the object and resolving method overloads. We will discuss this object further in the CPython section.

jclass

In addition class wrappers have a `_jpype.PyJPClass` field which represents the internal class wrapper. This class wrapper holds the reflection api used to create the class methods and fields. This field is stored as the `__javaclass__` in the class wrapper. As the class wrapper is used to create an object instance `__javaclass__` also appears in the objects. Only objects that have a `__javaclass__` and lack a `__javavalue__` are treated as class wrappers for the purposes of duck typing.

Because the Java class is both an object and type, we used the duck typing to allow the class pointer to be converted into a class instance. This is exposed as property called `class_`. The `class_` property is the equivalent of the .class member of a Java class name or `getClass()` of a class instance. As it is an object instance of `java.lang.Class` it can be used for any reflection needs by the user. However, when working with the jpype core module, on needs to be aware of that the this class instance is not available until the key wrappers are created. See *bootstrapping* for further details.

jarray

Java arrays are a special form of objects. They have no real methods as they are not extendable. To help in accessing the additonal special methods associated with an array, Java array instances have an additional field `__javaarray__` of type `PyJPArray`.

jstring

For most practical purposes Java strings are treated as objects. However, they also need to be able to interact with Python strings. In the previous version, strings were automatically converted to Python strings on return. This resulted in rather strange behavior when interacting with the methods of `java.lang.String` as rather than getting the expected Java object for chaining of commands, the string object would revert to Python. To avoid this fate, we now require string objects to be convert manually with the `str()` method in Python. There are still places where the conversion will trigger automatically such as pushing the string into string substitution. This does generate some potential for errors especially since it makes order important when using the equals operator when comparing a Java and Python string. Also it causes minor issues when using a Java string as a key to a `dict`. There is no special fields associated with the `jstring`.

Bootstrapping

The most challenging part in working with the jpype module other than the need to support both major Python versions with the same codebase is the bootstrapping of resources. In order to get the system working, we must pass the Python resources so the `_jpype` CPython module can acquire resources and then construct the wrappers for `java.lang.Object` and `java.lang.Class`. The key difficulty is that we need reflection to get methods from Java and those are part of `java.lang.Class`, but class inherits from `java.lang.Object`. Thus Object and the interfaces that Class inherits must all be created blindly. To support this process, a partial implmentation of the class reflection is implemented in `PyJPClass`.

The bootstrapping issue currently prevents further simplification of the internal layer as we need these hard coded support paths. To help keep the order of the bootstrapping consistent and allow the module to load before the JVM is started, actions are delayed in the jpype module. Those delayed actions are placing in initialize routines that are automatically called once the JVM is started.

Where accessing the class instance is required while building the class, the module globals are checked. If these globals are not yet loaded, the class structure can't be accessed. This means that `java.lang.Object`, `java.lang.Class`, and a few interfaces don't get the full class wrapper treatment. Fortunately, these classes are pretty light and don't contain addition resources such as inner classes that would require full reflection.

Factories

The key objects exposed to the user (`JClass`, `JObject`, and `JArray`) are each factory meta classes. These classes serve as the gate keepers to creating the meta classes or object instances. We found only one working pattern that support both Python versions. The pattern requires the class to have a polymorphic `__new__` function that depends on the arguments called. When called to access the factory, the `__new__` method redirects to the meta producing factory function. If called with any other arguments, falls to the correct `__new__` super method.

When dealing with these remember they are called typically three ways. The user calls with the specified arguments to create a resource. The factory calls the `__new__` method when creating an instance of the derived object. And the C++ wrapper calls the method with internally construct resource such as `PyJPClass` or `PyJPValue`. Deciding which of the three ways it has been called is usually simple, but it does constrain the operation of the factories as conflicts in the three paths would lead to a failure. Forcing keyword arguments for one of the paths could be used to resolve the dependency.

For example, if `JClass` was called with a string argument it originates from the user. If it was called with a `PyJPClass`, it came from internal module. If called with a class name, tuple, and list of members, it was a request from Python to create a new dynamic type. As the first two formulas have only one argument, both transfer the factory dispatch to create the dynamic resource. The other method transfers to the Python type object to create the actual class instance.

Style

One of the key aspects of the jpype design is elegance of the factory patterns. Rather than expose the user a large number of distinct concepts with different names, the factories provide powerfull functionality with the same syntax for related things. Boxing a primitive, casting to a specific type, and creating a new object are all tied together in one factory, `JObject`. By also making that factory an effective base class, we allow it to be used for `issubtype` and `isinstance`.

This philosophy is further enhanced by silent customizers which integrate Python functionality into the wrappers such that Java classes can be used effectively with Python syntax. Consistent use and misuse of Python concepts such as `with` for defining blocks such as try with resources and synchronized hide the underlying complexity and give the feeling to the user that the module is integrated completely as a solution such as jython.

When adding a new feature to the Python layer, consider carefully if the feature needs to be exposed a new function or if it can be hidden in the normal Python syntax.

JPype does somewhat break the Python naming conventions. Because Java and Python have very different naming schemes, at least part of the kit would have a different convention. To avoid having one portion break Python conventions and another part conform, we choose to use Java notation consistently throughout. Package names should be lower with underscores, classes should camel case starting upper, functions and method should be camel case starting lower. All private methods and classes start with a leading underscore and are not exported.

Customizers

There was a major change in the way the customizers work between versions. The previous system was undocumented and has now been removed, but as someone may have used of it previously, we will contrast it with the revised system so that the customizers can be converted.

In the previous system, a global list stored all customizers. When a class was created, it went though the list and asked the class if it matched that class name. If it matched, it altered the dict of members to be created so when the dynamic class was finished it had the custome behavior. This system wasn't very scalable as each customizer added more work to the class construction process.

The revised system works by storing a dictionary keyed to the class name. Thus the customizer only applies to the specific class targeted to the customizer. The customizer is specified using annotation of a prototype class making

methods automatically copy onto the class. However, sometimes a customizer needs to be applied to an entire tree of classes such as all classes that implement `java.util.List`. To handle this case, the class creation system looks for a special method `__java_init__` in the tree of base classes and calls it on the newly created class. Most of the time the customization was the same simple pattern so we added a `sticky` flag to build the initialization method directly. This method can alter the class to make it add the new behavior. Note the word alter. Where before we changed the member prior to creating the class, here we are altering the class. Thus the customizer is expected to monkey patch the existing class. There is only one pattern of monkey patching that works on both Python 2 and Python 3 so be sure to use the `type.__setattr__` method of altering the class dictionary.

It is possible to apply customizers after the class has already been created because we operate by monkey patching. But there is a limitation that there can only be one `__java_init__` method and thus two customizers specifying a global behavior on the same class wrapper will lead to unexpected behavior.

1.7.3 `_jpype` CPython module

Diving deeper into the onion, we have the Python front end. This is divided into a number of distinct pieces. Each piece is found under `native/python` and is named according to the piece it provides. For example, `PyJPModule` is found in the file `native/python/pyjp_module.cpp`

Earlier versions of the module had all of the functionality in the modules global space. This functionality is now split into a number of classes. These classes each have a constructor that is used to create an instance which will correspond to a Java resource such as class, array, method, or value.

`PyJPModule` module

This is the front end for all the global functions required to support the Python native portion. Most of the functions provided in the module are for control and auditing.

One important method is the setResource command. The `setResource` takes a name and a Python function or class, and passes it to `jp_pythonenv.cpp`. Prior to using duck typing to recognize jpype entities, a large number of resources had to be loaded to function. With the rewrite this has been reduced considerably to just function required to create a Python wrapper for a Java class, `GetClassMethod`. However, now that the kit has been streamlined additional Python resources will likely be required for new features.

`PyJPClass` class

This class supplied the portion of the reflection API required to create classes without the aid of the `java.lang.Class` structure. It can be constructed either from within JPype or directly from Python. It points to a `JPClass`. It has methods for diagnostics and reflection.

`PyJPMethod` class

This class acts as descriptor with a call method. As a descriptor accessing its methods through the class will trigger its `__get__` function, thus getting ahold of it within Python is a bit tricky. The `__get__` mathod is used to bind the static unbound method to a particular object instance so that we can call with the first argument as the `this` pointer.

It has some reflection and diagnostics methods that can be useful it tracing down errors. The beans methods are there just to support the old properties API.

The naming on this class is a bit deceptive. It does not correspond to a single method but rather all the overloads with the same name. When called it passes to with the arguments to the C++ layer where it must be resolved to a specific overload.

This class is stored directly in the class wrappers.

`PyJPField` class

This class is a descriptor with `__get__` and `__set__` methods. When called at the static class layer it operates on static fields. When called on a Python object, it binds to the object making a `this` pointer. If the field is static, it will continue to access the static field, otherwise, it will provide access to the member field. This trickery allows both static and member fields to wrap as one type.

This class is stored directly in the class wrappers.

`PyJPArray` class

This class serves as to provide the extra methods that are needed for working with Java arrays. It is not a descriptor and thus is hidden in the python class as `__javaarray__`. A Python array wrapper should have both the `__javavalue__` and the `__javaarray__` field to function.

`PyJPMonitor` class

This class provides `synchronized` to JPype. Instances of this class are created and held using `with`. It has two methods `__enter__` and `__exit__` which hook into the Python RAII system.

`PyJPValue` class

This class holds all types of Java resources that correspond to the Java jvalue union. This includes both objects and primitives. It provides a cache for string conversions so the when `str()` is called on a Java string we only pay for the conversion cost once. Both Java and Python strings are immutable thus this is a valid cache operation.

Unlike `jvalue` we hold the object type in the C++ `JPValue` object. The class reference is used to determine how to match the arguments to methods. The class may not correspond to the actual class of the object. Using a class other than the actual class serves to allow an object to be cast and thus treated like another type for the purposes of overloading. This mechanism is what allows the `JObject` factory to perform a typecast.

`PyJPValue` is held in the private python field `__javavalue__`.

1.7.4 CPython API layer

To make creation of the C++ layer easier a thin wrapper over the CPython API was developed. This layer provided for handling the CPython referencing using a smart pointer, defines the exception handling for Python, and provides resource hooks for duck typing of the `_jpype` classes.

This layer is located with the rest of the Python codes in `native/python`, but has the prefix `JPPy` for its classes. As the bridge between Python and C++, these support classes appear in both the `_jpype` CPython module and the C++ JNI layer.

Exception handling

A key piece of the jpype interaction is the transfer of exceptions from Java to Python. To accomplish this Python method that can result in a call to Java must have a `try` block around the contents of the function.

We use a routine pattern of code to interact with Java to achieve this:

```
PyObject* dosomething(PyObject* self, PyObject* args)
{
   // Tell the logger where we are
   JP_TRACE_IN("dosomething");

   // Start a block to catch Java emitted errors
   try
   {
      // Make sure there is a jvm to receive the call.
      ASSERT_JVM_RUNNING("dosomething");

      // Make a resource to capture any Java local references
      JPJavaFrame frame;

      // Call our Java methods
      ...

      // Return control to Python
      return obj.keep();
   }

   // Use the standard catch to transfer any exceptions back
   // to Python
   PY_STANDARD_CATCH;

   // Close out tracing
   JP_TRACE_OUT;
}
```

All entry points from Python into `_jpype` should be guarded with this pattern.

There are exceptions to this pattern such as removing the logging, operating on a call that does not need the jvm running, or operating where the frame is already supported by the method being called.

Python referencing

One of the most miserable aspects of programming with CPython is the relative inconsistancy of referencing. Each method in Python may use a Python object or steal it, or it may return a borrowed reference or give a fresh reference. Similar command such as getting an element from a list and getting an element from a tuple can have different rules. This was a constant source of bugs requiring consultation of the Python manual for every line of code. Thus we wrapped all of the Python calls we were required to work with in `jp_pythontypes`.

Included in this wrapper is a Python reference counter called `JPPyObject`. Whenever an object is returned from Python it is immediately placed in smart pointer `JPPyObject` with the policy that it was created with such as `use_`, `borrowed_`, `claim_` or `call_`.

`use_` This policy means that the reference counter needs to be incremented and the start and the end. We must reference it because if we don't and some Python call destroys the refernce out from under us, the system may crash and burn.

`borrowed_` This policy means we were to be give a borrowed reference that we are expected to reference and unreference when complete, but the command that returned it can fail. Thus before reference it, the system must check if an error has occurred. If there is an error, it is promoted to an exception.

`claim_` This policy is used when we are given a new object with is already referenced for us. Thus we are to steal the reference for the duration of our use and then dereference when we are done to keep it from leaking.

call_ This policy both steals the reference and verifies there were no errors prior to continuing. Errors are promoted to exceptions when this reference is created.

If we need to pass an object which is held in a smart pointer to Python which requires a reference, we call keep on the reference which transfers control to a `PyObject*` and prevents the pointer from removing the reference. As the object handle is leaving our control keep should only be called the return statement. The smart pointer is not used on method passing in which the parent explicitly holds a reference to the Python object. As all tuples passed as arguments operate like this, that means much of the API accepts bare `PyObject*` as arguments. It is the job of the caller to hold the reference for its scope.

1.7.5 C++ JNI layer

The C++ layer has a number of tasks. It is used to load thunks, call JNI methods, provide reflection of classes, determine if a conversion is possible, perform conversion, match arguments to overloads, and convert return values back to Java.

Memory management

Java provides built in memory management for controlling the lifespan of Java objects that are passed through JNI. When a Java object is created or returned from the JVM it returns a handle to object with a reference counter. To manage the lifespan of this reference counter a local frame is created. For the duration of this frame all local references will continue to exist. To extend the lifespan either a new global reference to the object needs to be created, or the object needs to be kept. When the local frame is destroyed all local references are destroyed with the exception of an optional specified local return reference.

We have wrapped the Java reference system with the wrapper `JPLocalFrame`. This wrapper has three functions. It acts as a RAII (Resource acquisition is initialization) for the local frame. Further, as creating a local frame requires creating a Java env reference and all JNI calls require access to the env, the local frame acts as the front end to call all JNI calls. Finally as getting ahold of the env requires that the thread be attached to Java, it also serves to automatically attach threads to the JVM. As accessing an unbound thread will cause a segmentation fault in JNI, we are now safe from any threads created from within Python even those created outside our knowledge. (I am looking at you spyder)

Using this pattern makes the JPype core safe by design. Forcing JNI calles to be called using the frame ensures:

- Every local reference is destroyed.
- Every thread is properly attached before JNI is used.
- The pattern of keep only one local reference is obeyed.

To use a local frame, use the pattern shown in this example.

```
jobject doSomeThing(std::string args)
{
    // Create a frame at the top of the scope
    JPLocalFrame frame;

    // Do the required work
    jobject obj =frame.CallObjectMethodA(globalObj, methodRef, params);

    // Tell the frame to return the reference to the outer scope.
    //   once keep is called the frame is destroyed and any
    //   call will fail.
    return frame.keep(obj);
}
```

Note that the value of the object returned and the object in the function will not be the same. The returned reference is owned by the enclosing local frame and points to the same object. But as its lifespan belongs to the outer frame, its location in memory is different. You are allowed to `keep` a reference that was global or was passed in, in either of those case, the outer scope will get a new local reference that points to the same object. Thus you don't need to track the origin of the object.

The changing of the value while pointing is another common problem. A routine error is to get a local reference, call `NewGlobalRef` and then keeping the local reference rather than the shiny new global reference it made. This is not like the Python reference system where you have the object that you can ref and unref. Thus make sure you always store only the global reference.

```
jobject global;

// we are getting a reference, may be local, may be global.
// either way it is borrowed and it doesn't belong to us.
void elseWhere(jvalue value)
{
  JPLocalFrame frame;

  // Bunch of code leading us to decide we need to
  // hold the resource longer.
  if (cond)
  {
    // okay we need to keep this reference, so make a
    // new global reference to it.
    global = frame.NewGlobalRef(value.l);
  }
}
```

But don't mistake this as an invitation to make global references everywhere. Global reference are global, thus will hold the member until the reference is destroyed. C++ exceptions can lead to missing the unreference, thus global references should only happen when you are placing the Java object into a class member variable or a global variable.

To help manage global references, we have `JPRef<>` which holds a global reference for the duration of the C++ lifespace. This is the base class for each of the global reference types we use.

```
typedef JPRef<jclass> JPClassRef;
typedef JPRef<jobject> JPObjectRef;
typedef JPRef<jarray> JPArrayRef;
typedef JPRef<jthrowable> JPThrowableRef;
```

For functions that expect the outer scope to already have created a frame for this context, we use the pattern of extending the outer scope rather than creating a new one.

```
jobject doSomeThing(JPLocalFrame& frame, std::string args)
{
    // Do the required work
    jobject obj = frame.CallObjectMethodA(globalObj, methodRef, params);

    // We must not call keep here or we will terminate
    // a frame we do not own.
    return obj;
}
```

Although the system we have set up is "safe by design", there are things that can go wrong is misused. If the caller fails to create a frame prior to calling a function that returns a local reference, the reference will go into the program scoped local references and thus leak. Thus, it is usually best to force the user to make a scope with the frame extension pattern. Second, if any JNI references that are not kept or converted to global, it becomes invalid. Further, since JNI

recycles the reference pointer fairly quickly, it most likely will be pointed to another object whose type may not be expected. Thus, best case is using the stale reference will crash and burn. Worse case, the reference will be a live reference to another object and it will produce an error which seems completely irrelevant to anything that was being called. Horrible case, the live object does not object to bad call and it all silently proceeds down the road another two miles before coming to flaming death.

Moral of the story, always create a local frame even if you are handling a global reference. If passed or returned a reference of any kind, it is a borrowed reference belonging to the caller or being held by the current local frame. Thus it must be treated accordingly. If you have to hold a global use the appropraite `JPRef` class to ensure it is exception and dtor safe. For further information read `native/common/jp_javaframe.h`.

Type wrappers

Each Java type has a C++ wrapper class. These classes provide a number of methods. Primitives each have their own unit type wrapper. Object, arrays, and class instances share a C++ wrapper type. Special instances are used for `java.lang.Object` and `java.lang.Class`. The type wrapper are named for the class they wrap such as `JPIntType`.

Type conversion

For type conversion, a C++ class wrapper provides four methods.

`canConvertToJava` This method must consult the supplied Python object to determine the type and then make a determination of whether a conversion is possible. It reports `none_` if there is no possible conversion, `explicit_` if the conversion is only acceptable if forced such as returning from a proxy, `implicit_` if the conversion is possible and acceptable as part of an method call, or `exact_` if this type converts without ambiguity. It is excepted to check for something that is already a Java resource of the correct type such as `JPValue`, or something this is implementing the behavior as an interface in the form of a `JPProxy`.

`convertToJava` This method consults the type and produces a conversion. The order of the match should be identical to the `canConvertToJava`. It should also handle values and proxies.

`convertToPythonObject` This method takes a jvalue union and converts it to the corresponding Python wrapper instance.

`getValueFromObject` This converts a Java object into a `JPValue` corresponding. This unboxes primitives.

Array conversion

In addition to converting single objects, the type rewrappers also serve as the gateway to working with arrays of the specified type. Five methods are used to work with arrays: `newArrayInstance`, `getArrayRange`, `setArrayRange`, `getArrayItem`, and `setArrayItem`.

Invocation and Fields

To convert a return type produced from a Java call, each type needs to be able to invoke a method with that return type. This corresponses the underlying JNI design. The methods invoke and invokeStatic are used for this purpose. Similarly accessing fields requires type conversion using the methods `getField` and `setField`.

Instance versus Type wrappers

Instances of individual Java classes are made from `JPClass`. However, two special sets of conversion rules are required. These are in the form of specializations `JPObjectBaseClass` and `JPClassBaseClass` corresponding to `java.lang.Object` and `java.lang.Class`.

Support classes

In addition to the type wrappers, there are several support classes. These are:

`JPTypeManager` The typemanager serves as a dict for all type wrappers created during the operation.

`JPReferenceQueue` Lifetime manager for Java and Python objects.

`JPProxy` Proxies implement a Java interface in Python.

`JPClassLoader` Loader for Java thunks.

`JPEncoding` Decodes and encodes Java UTF strings.

`JPTypeManager`

C++ typewrappers are created as needed. Instance of each of the primitives along with `java.lang.Object` and `java.lang.Class` are preloaded. Additional instances are created as requested for individual Java classes. Currently this is backed by a C++ map of string to class wrappers.

The typemanager provides a number lookup methods.

```
// Call from within Python
JPClass* JPTypeManager::findClass(const string& name)

// Call from a defined Java class
JPClass* JPTypeManager::findClass(jclass cls)

// Call used when returning an object from Java
JPClass* JPTypeManager::findClassForObject(jobject obj)
```

`JPReferenceQueue`

When a Python object is presented to Java as opposed to a Java object, the lifespan of the Python object must be extended to match the Java wrapper. The reference queue adds a reference to the Python object that will be removed by the Java layer when the garbage collection deletes the wrapper. This code is almost entirely in the Java library, thus only the portion to support Java native methods appears in the C++ layer.

Once started the reference queue is mostly transparent. registerRef is used to bind a Python object live span to a Java object.

```
void JPReferenceQueue::registerRef(jobject obj, PyObject* hostRef)
```

`JPProxy`

In order to call Python functions from within Java, a Java proxy is used. The majority of the code is in Java. The C++ code holds the Java native portion. The native implement of the proxy call is the only place in with the pattern for reflecting Python exceptions back into Java appears.

As all proxies are ties to Python references, this code is strongly tied to the reference queue.

JPClassLoader

This code is responsible for loading the Java class thunks. As it is difficult to ensure we can access a Java jar from within Python, all Java native code is stored in a binary thunk compiled into the C++ layer as a header. The class loader provides a way to load this embedded jar first by bootstrapping a custom Java classloader and then using that classloader to load the internal jar.

The classloader is mostly transparent. It provides one method called findClass which loads a class from the internal jar.

```
jclass JPClassLoader::findClass(string name)
```

JPEncoding

Java concept of UTF is pretty much out of sync with the rest of the world. Java used 16 bits for its native characters. But this was inadequate for all of the unicode characters, thus longer unicode character had to be encoded in the 16 bit space. Rather the directly providing methods to convert to a standard encoding such as UTF8, Java used UTF16 encoded in 8 bits which they dub Modified-UTF8. `JPEncoding` deals with converting this unusual encoding into something that Python can understand.

The key method in this module is transcribe with signature

```
std::string transcribe(const char* in, size_t len,
    const JPEncoding& sourceEncoding,
    const JPEncoding& targetEncoding)
```

There are two encodings provided, `JPEncodingUTF8` and `JPEncodingJavaUTF8`. By selecting the source and traget encoding transcribe can convert to or from Java to Python encoding.

Incidentally that same modified UTF coding is used in storing symbols in the class files. It seems like a really poor design choice given they have to document this modified UTF in multiple places. As far as I can tell the internal converter only appears on `java.io.DataInput` and `java.io.DataOutput`.

1.7.6 Java native code

At the lowest level of the onion is the native Java layer. Although this layer is most remote from Python, ironically it is the easiest layer to communicate with. As the point of jpype is to communicate with Java, it is possible to directly communicate with the jpype Java internals. These can be imported from the package `org.jpype`. The code for the Java layer is located in `native/java`. It is compiled into a jar in the build directory and then converted to a C++ header to be compiled into the `_jpype` module.

The Java layer currently houses the reference queue, a classloader which can load a Java class from a bytestream source, the proxy code for implementing Java interfaces, and a memory compiler module which allows Python to directly create a class from a string.

1.7.7 Tracing

Because the relations between the layers can be daunting especially when things go wrong. The CPython and C++ layer have a built in logger. This logger must be enabled with a compiler switch to activate. To active the logger, touch one of the cpp files in the native directory to mark the build as dirty, then compile the `jpype` module with:

```
python setup.py --enable-tracing devel
```

Once built run a short test program that demonstrates the problem and capture the output of the terminal to a file. This should allow the developer to isolate the fault to specific location where it failed.

To use the logger in a function start the `JP_TRACE_IN(function_name)` which will open a `try catch` block.

1.7.8 Future directions

Although the majority of the code has been reworked for JPype 0.7, there is still further work to be done. Almost all Java constructs can be exercised from within Python, but Java and Python are not static. Thus, we are working on further improvements to the jpype core focusing on making the package faster, more efficient, and easier to maintain. This section will discuss a few of these options.

Java based code is much easier to debug as it is possible to swap the thunk code with an external jar. Further, Java has much easier management of resources. Thus pushing a portion of the C++ layer into the Java layer could further reduce the size of the code base. In particular, deciding the order of search for method overloads in C++ attempts to reconstruct the Java overload rules. But these same rules are already available in Java. Further, the C++ layer is designed to make many frequent small calls to Java methods. This is not the preferred method to operate in JNI. It is better to have specialized code in Java which preforms large tasks such as collecting all of the fields needed for a type wrapper and passing it back in a single call, rather than call twenty different general purpose methods. This would also vastly reduce the number of `jmethods` that need to be bound in the C++ layer.

The world of JVMs is currently in flux. Jpype needs to be able to support other JVMs. In theory, so long a JVM provides a working JNI layer, there is no reason the jpype can't support it. But we need loading routines for these JVMs to be developed if there are differences in getting the JVM launched.

There is a project page on github shows what is being developed for the next release. Series 0.6 was usable, but early versions had notable issues with threading and internal memory management concepts had to be redone for stability. Series 0.7 is the first verion after rewrite for simplication and hardening. I consider 0.7 to be at the level of production quality code suitable for most usage though still missing some needed features. Series 0.8 will deal with higher levels of Python/Java integration such as Java class extension and pickle support. Series 0.9 will be dedicated to any additional hardening and edge cases in the core code as we should have complete integration. Assuming everything is completed, we will one day become a real boy and have a 1.0 release.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

Symbols

A

D

G

I

J

R

S