jtypes.javabridge Documentation

Release 1.0.18b3

Adam Karpierz

Oct 18, 2020

Contents:

The **jtypes.javabridge** Python package makes it easy to start a Java virtual machine (JVM) from Python and interact with it. Python code can interact with the JVM using a low-level API or a more convenient high-level API.

PyPI record.

jtypes.javabridge is an almost fully compliant implementation of Lee Kamentsky's and Vebjorn Ljosa's good known **Javabridge** package.

Original **javabridge** was developed for CellProfiler, where it is used together with python-bioformats to interface to various Java code, including Bio-Formats and ImageJ.

Original documentation

CHAPTER 1

README

Currently only as placeholder (because a base package jtypes.jvm is still in development)

1.1 jtypes.javabridge

Python wrapper for the Java Native Interface.

1.2 Overview

> **jtypes.javabridge** is a bridge between Python and Java, allowing these to intercommunicate.
> It is an effort to allow python programs full access to Java class libraries.
>
> PyPI record.

> **jtypes.javabridge** is a lightweight Python package, based on the *ctypes* or *cffi* library.
> It is an almost fully compliant implementation of Lee Kamentsky's and Vebjorn Ljosa's **Javabridge** package by reimplementing whole its functionality in a clean Python instead of Cython and C.

1.2.1 About javabridge:

Borrowed from the original website:

> The **javabridge** Python package makes it easy to start a Java virtual machine (JVM) from Python and interact with it. Python code can interact with the JVM using a low-level API or a more convenient high-level API.

1.3 Requirements

- Java Runtime (JRE) or Java Development Kit (JDK), and NumPy (not mandatory but highly recommended).

1.4 Installation

Prerequisites:

- Python 2.7 or higher or 3.4 or higher
  - http://www.python.org/
  - 2.7 and 3.6 are primary test environments.
- pip and setuptools
  - http://pypi.python.org/pypi/pip
  - http://pypi.python.org/pypi/setuptools

To install run:

```
python -m pip install --upgrade jtypes.javabridge
```

To ensure everything is running correctly you can run the tests using:

```
python -m jt.javabridge.tests
```

1.5 Development

Visit development page

Installation from sources:

Clone the sources and run:

```
python -m pip install ./jtypes.javabridge
```

or on development mode:

```
python -m pip install --editable ./jtypes.javabridge
```

Prerequisites:

- Development is strictly based on *tox*. To install it run:

  ```
  python -m pip install tox
  ```

1.6 License

1.7 Authors

- Adam Karpierz <adam@karpierz.net>

CHAPTER 2

Installation and testing

2.1 Install using pip

```
python -m pip install numpy  # not mandatory but highly recommended
python -m pip install jtypes.javabridge
```

2.2 Install without pip

```
# Make sure numpy is installed (not mandatory but highly recommended)
python setup.py install
```

2.3 Dependencies

The *jtypes.javabridge* requires Python 2.7 or above, NumPy (not mandatory but highly recommended) and the Java Runtime Environment (JRE) (a C compiler is not required).

2.3.1 Linux

On CentOS 6, the dependencies can be installed as follows:

```
yum install gcc numpy java-1.7.0-openjdk-devel
curl -O https://raw.github.com/pypa/pip/master/contrib/get-pip.py
python get-pip.py
```

On Fedora 19, the dependencies can be installed as follows:

```
yum install gcc numpy java-1.7.0-openjdk-devel python-pip openssl
```

On Ubuntu 13, 14 and Debian 7, the dependencies can be installed as follows:

```
apt-get install openjdk-7-jdk python-pip python-numpy
```

On Arch Linux, the dependencies can be installed as follows:

```
pacman -S jdk7-openjdk python2-pip python2-numpy base-devel
```

2.3.2 MacOS X

1. Install the Xcode command-line tools. There are two ways:
   A. Install Xcode from the Mac App Store. (You can also download it from Apple's Mac Dev Center, but that may require membership in the Apple Developer Program.) Install the Xcode command-line tools by starting Xcode, going to Preferences, click on "Downloads" in the toolbar, and click the "Install" button on the line "Command Line Tools." For MacOS 10.9 and Xcode 5 and above, you may have to install the command-line tools by typing `xcode-select --install` and following the prompts.
   B. Download the Xcode command-line tools from Apple's Mac Dev Center and install. This may require membership in the Apple Developer Program.
2. Create and activate a *virtualenv* virtual environment if you don't want to clutter up your system-wide python installation with new packages.
3. `python -m pip install numpy` # not mandatory but highly recommended
4. `python -m pip install jtypes.javabridge`

2.3.3 Windows

If you do not have a C compiler installed, you can install Microsoft Visual C++ Build Tools to perform the compile steps. The compiler installation can be found in https://visualstudio.microsoft.com/visual-cpp-build-tools/.

You should install a Java Development Kit (JDK) appropriate for your Java project. The Windows build is tested with the Oracle JDK 1.7. You also need to install the Java Runtime Environment (JRE). Note that the bitness needs to match your python: if you use a 32-bit Python, then you need a 32-bit JRE; if you use a 64-bit Python, then you need a 64-bit JRE.

The paths to PIP and Python should be in your PATH (`set PATH=%PATH%;c:\\Python27;c:\\Python27\\scripts` if Python and PIP installed to the default locations). The following steps should perform the install:

1. Run Command Prompt as administrator. Set the path to Python and PIP if needed.
2. Issue the command:

```
python -m pip install jtypes.javabridge
```

2.4 Running the unit tests

Running the unit tests requires Nose. Some of the tests require Python 2.7 or above.

1. Build and install in the source code tree so that the unit tests can run:

```
python setup.py develop
```

2. Run the unit tests:

```
python -m jt.javabridge.tests
```

You must build the extensions in-place on Windows, then run tests if you use setup to run the tests:

```
python setup.py build_ext -i
python setup.py tests
```

See the section *Unit testing* for how to run unit tests for your own projects that use *jtypes.javabridge*.

CHAPTER 3

Hello world

Without a GUI:

```
import os
from jt import javabridge

javabridge.start_vm(run_headless=True)
try:
    print(javabridge.run_script('java.lang.String.format("Hello, %s!", greetee);',
                                dict(greetee='world')))
finally:
    javabridge.kill_vm()
```

You can also use a with block:

```
import os
from jt import javabridge

with javabridge.vm(run_headless=True)
    print(javabridge.run_script('java.lang.String.format("Hello, %s!", greetee);',
                                dict(greetee='world')))
```

With only a Java AWT GUI:

```
import os
import wx
from jt import javabridge

javabridge.start_vm()

class EmptyApp(wx.App):
    def OnInit(self):
        javabridge.activate_awt()
        return True
```

(continues on next page)

(continued from previous page)

```
try:

    app = EmptyApp(False)

    # Must exist (perhaps the app needs to have a top-level window?), but
    # does not have to be shown.
    frame = wx.Frame(None)

    javabridge.execute_runnable_in_main_thread(javabridge.run_script("""
        new java.lang.Runnable() {
            run: function() {
                with(JavaImporter(java.awt.Frame)) Frame().setVisible(true);
            }
        };"""))

    app.MainLoop()

finally:

    javabridge.kill_vm()
```

Mixing wxPython and Java AWT GUIs:

```
import os
import wx
from jt import javabridge

class EmptyApp(wx.PySimpleApp):
    def OnInit(self):
        javabridge.activate_awt()

        return True

javabridge.start_vm()

try:
    app = EmptyApp(False)

    frame = wx.Frame(None)
    frame.Sizer = wx.BoxSizer(wx.HORIZONTAL)
    launch_button = wx.Button(frame, label="Launch AWT frame")
    frame.Sizer.Add(launch_button, 1, wx.ALIGN_CENTER_HORIZONTAL)

    def fn_launch_frame(event):
        javabridge.execute_runnable_in_main_thread(javabridge.run_script("""
        new java.lang.Runnable() {
            run: function() {
                with(JavaImporter(java.awt.Frame)) Frame().setVisible(true);
            }
        };"""))
    launch_button.Bind(wx.EVT_BUTTON, fn_launch_frame)

    frame.Layout()
    frame.Show()
    app.MainLoop()

finally:
```

(continues on next page)

(continued from previous page)

```
javabridge.kill_vm()
```

CHAPTER 4

Starting and killing the JVM

4.1 API

`jt.javabridge.`**`JARS`**
a list of strings; gives the full path to some JAR files that should be added to the class path in order for all the features of the *jtypes.javabridge* to work properly.

4.1.1 Environment

In order to use the *jtypes.javabridge* in a thread, you need to attach to the JVM's environment in that thread. In order for the garbage collector to be able to collect thread-local variables, it is also necessary to detach from the environment before the thread ends.

4.2 Without GUI (headless mode)

Using the JVM in headless mode is straighforward:

```
import os
from jt import javabridge

javabridge.start_vm(run_headless=True)
try:
    print(javabridge.run_script('java.lang.String.format("Hello, %s!", greetee);',
                                dict(greetee='world')))
finally:
    javabridge.kill_vm()
```

4.3 With GUI on the Java side

Using the JVM with a graphical user interface is much more involved because you have to run an event loop on the Python side. You also have to make sure that everything executes in the proper thread; in particular, all GUI operations have to run in the main thread on Mac OS X. Here is an example, using a wxPython app to provide the event loop:

```
import os
import wx
from jt import javabridge

javabridge.start_vm()

class EmptyApp(wx.App):
    def OnInit(self):
        javabridge.activate_awt()
        return True

try:

    app = EmptyApp(False)

    # Must exist (perhaps the app needs to have a top-level window?), but
    # does not have to be shown.
    frame = wx.Frame(None)

    javabridge.execute_runnable_in_main_thread(javabridge.run_script("""
        new java.lang.Runnable() {
            run: function() {
                with(JavaImporter(java.awt.Frame)) Frame().setVisible(true);
            }
        };"""))

    app.MainLoop()

finally:

    javabridge.kill_vm()
```

4.4 With GUI on both the Java side and the Python side

Finally, an example combining AWT for GUI on the Java side with wxPython for GUI on the Python side:

```
import os
import wx
from jt import javabridge

class EmptyApp(wx.PySimpleApp):
    def OnInit(self):
        javabridge.activate_awt()

        return True

javabridge.start_vm()
```

(continues on next page)

(continued from previous page)

```
try:
    app = EmptyApp(False)

    frame = wx.Frame(None)
    frame.Sizer = wx.BoxSizer(wx.HORIZONTAL)
    launch_button = wx.Button(frame, label="Launch AWT frame")
    frame.Sizer.Add(launch_button, 1, wx.ALIGN_CENTER_HORIZONTAL)

    def fn_launch_frame(event):
        javabridge.execute_runnable_in_main_thread(javabridge.run_script("""
            new java.lang.Runnable() {
                run: function() {
                    with(JavaImporter(java.awt.Frame)) Frame().setVisible(true);
                }
            };"""))
    launch_button.Bind(wx.EVT_BUTTON, fn_launch_frame)

    frame.Layout()
    frame.Show()
    app.MainLoop()

finally:

    javabridge.kill_vm()
```

CHAPTER 5

Executing JavaScript on the JVM

As you will see in subsequent sections, navigating and manipulating the JVM's class and object structure can result in verbose and cumbersome Python code. Therefore, *jtypes.javabridge* ships with the JavaScript interpreter Rhino, which runs on the JVM. In many cases, the most convienient way to interact with the JVM is to execute a piece of JavaScript.

For more information on using Rhino with the JVM see https://developer.mozilla.org/en-US/docs/Rhino/Scripting_Java

Examples:

```
>>> from jt import javabridge
>>> javabridge.run_script("2 + 2")
4
```

```
>>> javabridge.run_script("a + b", bindings_in={"a": 2, "b": 3})
5
```

```
>>> outputs = {"result": None}
>>> javabridge.run_script("var result = 2 + 2;", bindings_out=outputs)
>>> outputs["result"]
4
```

```
>>> javabridge.run_script("java.lang.Math.abs(v)", bindings_in=dict(v=-1.5))
1.5
```

A conversion is necessary when converting from Python primitives and objects to Java and JavaScript primitives and objects. Python primitives are boxed into Java objects - Javascript will automatically unbox them when calling a method that takes primitive arguments (e.g. the call to Math.abs(double) as in the above example. The following is a table of bidirectional translations from Python to Java / Javascript and vice-versa:

Python	Java - boxed	Java-primitive
bool	java.lang.Boolean	boolean
int	java.lang.Integer	int
long	java.lang.Long	long
float	java.lang.Double	double
unicode	java.lang.String	N/A
str (Python->java only)	java.lang.String	N/A
None	null	N/A

CHAPTER 6

High-level API

The high-level API can wrap a Java object or class so that its methods and fields can be referenced by dot syntax. It also has functions that offload some of the burden of exception handling and type conversion, thus providing a mid-level compromise between ease of use and performance.

6.1 Signatures

jtypes.javabridge uses method signatures when it uses the JNI method lookup APIs. The method signatures are also used to convert between Python and Java primitives and objects. If you use the high-level API, as opposed to scripting, you will need to learn how to construct a signature for a class method. For example, java.lang.String has the following three methods:

```
public char charAt(int index)
public int indexOf(String str)
public byte [] getString(String charsetName)
```

charAt has the signature, "(I)C", because it takes one integer argument (I) and its return value is a char (C).

indexOf has the signature, "(Ljava/lang/String;)I", "L" and ";" bracket a class name which is represented as a path instead of with the dotted syntax.

getString has the signature, "(Ljava/lang/String;)[B. "[B" uses "[" to indicate that an array will be returned and "B" indicates that the array is of type, byte.

The signature syntax is described in JNI Types and Data Structures. An example: "(ILjava/lang/String;)[I" takes an integer and string as parameters and returns an array of integers.

Cheat sheet:

Z boolean

B byte

C char

S short

I int

J long

F float

D double

L class (e.g., Lmy/class;)

[array of (e.g., [B = byte array)

The signatures are difficult, but you can cheat: the JDK has a Java class file disassembler called `javap` that prints out the signatures of everything in a class.

6.2 Wrapping Java objects using reflection

6.3 Operations on Java objects

6.4 Hand-coding Python objects that wrap Java objects

The functions `make_new` and `make_method` create Python methods that wrap Java constructors and methods, respectively. The function can be used to create Python wrapper classes for Java classes. Example:

```
>>> from jt import javabridge
>>> class Integer:
        new_fn = javabridge.make_new("java/lang/Integer", "(I)V")
        def __init__(self, i):
            self.new_fn(i)
        intValue = javabridge.make_method("intValue", "()I", "Retrieve the integer␣
↪value")
>>> i = Integer(435)
>>> i.intValue()
435
```

6.5 Useful collection wrappers

The collection wrappers take a Java object that implements some interface and return a corresponding Python object that wraps the interface's methods and in addition provide Python-style access to the Java object. The Java object itself is, by convention, saved as self.o in the Python object.

6.6 Reflection

These functions make class wrappers suitable for introspection. These wrappers are examples of the kinds of wrappers that you can build yourself using `make_method` and `make_new`.

6.7 Executing in the correct thread

Ensure that callables, runnables and futures that use AWT run in the AWT main thread, which is not accessible from Python for some operating systems.

6.8 Exceptions

CHAPTER 7

The low-level API

This API wraps the Java Native Interface (JNI) at the lowest level. It provides primitives for creating an environment and making calls on it.

Java array objects are handled as numpy arrays.

Each thread has its own environment. When you start a thread, you must attach to the VM to get that thread's environment and access Java from that thread. You must detach from the VM before the thread exits.

In order to get the environment:

Examples::

```
>>> from jt.javabridge import get_env
>>> env = get_env()
>>> s = env.new_string(u"Hello, world.")
>>> c = env.get_object_class(s)
>>> method_id = env.get_method_id(c, "length", "()I")
>>> method_id
<Java method with sig=()I at 0xa0a4fd0>
>>> result = env.call_method(s, method_id)
>>> result
13
```

CHAPTER 8

Calling Python from Java

The *jtypes.javabridge* loads a Java class, org.cellprofiler.javabridge.CPython, that can be used to execute Python code. The class can be used within Java code called from the Python interpreter or it can be used within Java to run Python embedded in Java.

class org.cellprofiler.javascript.**CPython**()

The CPython class binds the Python interpreter to the JVM and provides the ability to execute Python scripts.

org.cellprofiler.javascript.CPython.**exec**()

Arguments

- **script** – The Python script to execute.
- **locals** – A map of the name of a Java object in the Python execution context to the Java object itself. The objects in the map have local scope. A null value can be used if no locals need to be defined.
- **globals** – A map of the name of a Java object to the Java object itself. The objects in the map have global scope. If a null value is used, `globals` defaults to the builtin globals.

`exec()` executes the script passed within the Python interpreter. The interpreter adds the builtin globals to the globals passed in, then executes the script. The same map may be used for both the locals and the globals - this mode may seem more familiar to those who regularly script in Python and expect the `import` statement to have a global effect.

There is no `eval` method. You can retrieve values by passing a container object such as an array or map as one of the locals and you can set elements in the object with values to be returned.

Example:

```
class MyClass {
    static final CPython cpython = CPython();

    public List<String> whereIsWaldo(String root) {
         ArrayList<String> result = new ArrayList<String>();
         Hashtable locals = new Hashtable();
         locals.put("result", result);
```

(continues on next page)

(continued from previous page)

```
        locals.put("root", root);
        StringBuilder script = new StringBuilder();
        script.append("import os\n");
        script.append("from jt import javabridge\n");
        script.append("root = javabridge.to_string(root)");
        script.append("result = javabridge.JWrapper(result)");
        script.append("for path, dirnames, filenames in os.walk(root):\n");
        script.append("  if 'waldo' in filenames:");
        script.append("     result.add(path)");
        cpython.exec(script.toString(), locals, null);
        return result;
    }

}
```

`org.cellprofiler.javascript.CPython.`**`execute`**`()`
`execute` is a synonym for `exec` which is a Python keyword. Use `execute` in place of `exec` to call Python from a *jtypes.javabridge* CWrapper for CPython.

8.1 Maintaing references to Python values

You may want to maintain references to Python objects across script executions. The following functions let a Java caller refer to a Python value (which can be a base type or an object) via a token which may be exchanged for the value at any time. The Java code is responsible for managing the reference's lifetime. Example:

```
from jt import javabridge

cpython = javabridge.JClassWrapper('org.cellprofiler.javabridge.CPython')()
d = javabridge.JClassWrapper('java.util.Hashtable')()
result = javabridge.JClassWrapper('java.util.ArrayList')()
d.put("result", result)
cpython.execute(
    'from jt import javabridge\n'
    'x = { "foo":"bar"}\n'
    'ref_id = javabridge.create_and_lock_jref(x)\n'
    'javabridge.JWrapper(result).add(ref_id)', d, d)
cpython.execute(
    'from jt import javabridge\n'
    'ref_id = javabridge.to_string(javabridge.JWrapper(result).get(0))\n'
    'assert javabridge.redeem_jref(ref_id)["foo"] == "bar"\n'
    'javabridge.unlock_jref(ref_id)', d, d)
```

CHAPTER 9

Unit testing

Unit testing of code that uses the *jtypes.javabridge* requires special care because the JVM can only be run once: after you kill it, it cannot be restarted. Therefore, the JVM cannot be started and stopped in the regular `setUp()` and `tearDown()` methods.

You should then be able to run the `tests` module:

```
python -m jt.javabridge.tests
```

On some installations, setuptools's test command will also work:

```
python setup.py test
```

If you prefer, these options can also be given on the command line:

```
nosetests --with-javabridge=True --classpath=my-project/jars/foo.jar
```

or:

```
python setup.py nosetests --with-javabridge=True --classpath=my-project/jars/foo.jar
```

CHAPTER 10

For *jtypes.javabridge* developers

10.1 Build from git repository

```
git clone git@github.com:CellProfiler/python-javabridge.git
cd python-javabridge
cython *.pyx
python setup.py build
python setup.py install
```

10.2 Make source distribution and publish

```
git tag -a -m 'A commit message' '1.0.0pr11'
git push --tags   # Not necessary, but you'll want to do it at some point
git clean -fdx
python setup.py develop
python setup.py sdist upload
python setup.py build_sphinx
python setup.py upload_sphinx
```

10.3 Upload source distribution built by Jenkins

```
git tag -a -m 'A commit message' '1.0.4'
git push --tags   # Not necessary, but you'll want to do it at some point
# Kick off a new Jenkins build manually, wait for it, and download.
twine upload javabridge-1.0.4.tar.gz
python setup.py build_sphinx
python setup.py upload_sphinx
```

CHAPTER 11

Changelog

11.1 1.0.18b3 (2018-11-08)

- Update of the required setuptools version.
- Minor setup and tests improvements.

11.2 1.0.18b1 (2018-10-01)

- Synchro with javabridge master branch (v.1.0.18+).

11.3 1.0.17b2 (2018-05-29)

- Synchro with javabridge master branch.
- Bug fixes and improvements in Java 9 support.
- Update of Mozilla Rhino.
- Update of the required setuptools version.

11.4 1.0.14b4 (2018-02-26)

- Improvement and simplification of setup and packaging.

11.5 1.0.14b3 (2018-01-29)

- Development moved to github.

- General improvements and update.

11.6 1.0.14b2 (2017-01-01)

- Second beta release.
- Version numbering in sync. with the original javabridge.

11.7 0.1.1a1 (2014-10-05)

- Initial version.

CHAPTER 12

Indices and tables

- genindex
- search

Index

J

O