jpy Documentation

Release 0.9.0

Brockmann Consult GmbH

Jun 08, 2022

Contents

1 Introduction **3**
- 1.1 How it works . . . 3
- 1.2 Current limitations . . . 5
- 1.3 Other projects with similar aims . . . 5

2 Installation **7**
- 2.1 Getting the Sources . . . 7
- 2.2 Build from Sources . . . 7
- 2.3 Running Java from Python . . . 8
- 2.4 Running Python from Java . . . 8
- 2.5 Typical Build Problems . . . 10

3 Tutorial **13**
- 3.1 Using jpy with Python . . . 13
- 3.2 Using jpy with Java . . . 13

4 Reference **15**
- 4.1 Python API . . . 15
- 4.2 Java API . . . 21

5 How to Modify **23**
- 5.1 Rebuild Process . . . 23
- 5.2 C Programming Guideline . . . 23

6 Indices and tables **25**

Index **27**

jpy is a *bi-directional* Java-Python bridge allowing you to call Java from Python and Python from Java.

Contents:

CHAPTER 1

Introduction

jpy is a **bi-directional** Python-Java bridge which you can use to embed Java code in Python programs or the other way round. It has been designed particularly with regard to maximum data transfer speed between the two languages. It comes with a number of outstanding features:

- Fully translates Java class hierarchies to Python
- Transparently handles Java method overloading
- Support of Java multi-threading
- Fast and memory-efficient support of primitive Java array parameters via Python buffers (e.g. numpy arrays)
- Support of Java methods that modify primitive Java array parameters (mutable parameters)
- Java arrays translate into Python sequence objects
- Java API for accessing Python objects (`jpy.jar`)

jpy has been tested with Python 2.7, 3.3, 3.4 and Oracle Java 7 and 8 JDKs. It will presumably also work with Python 2.6 or 3.2 and a Java 6 JDK.

The initial development of jpy has been driven by the need to write Python extensions to an established scientific imaging application programmed in Java, namely the BEAM toolbox funded by the European Space Agency (ESA). Writing such Python plug-ins for a Java application usually requires a bi-directional communication between Python and Java since the Python extension code must be able to call back into the Java APIs.

1.1 How it works

The jpy Python module is entirely written in the C programming language. The same resulting shared library is used as a Python jpy module and also as native library for the Java library (`jpy.jar`).

Python programs that import the `jpy` module can load Java classes, access Java class fields, and call class constructors and methods.

Java programs with `jpy.jar` on the classpath can import Python modules, access module attributes such as class types and variables, and call any callable objects such as module-level functions, class constructors, as well as static and instance class methods.

1.1.1 Calling Java from Python

Instantiate Python objects from Java classes and call their public methods and fields:

```
import jpy

File = jpy.get_type('java.io.File')

file = File('test/it')
name = file.getName()
```

1.1.2 Calling Python from Java

Access Python attributes and call Python functions from Java:

```
PyModule sys = PyModule.importModule("sys");
PyObject path = sys.getAttribute("path");
path.call("append", "/usr/home/norman/");
String value = path.getStringValue();
```

1.1.3 Implementing Java interfaces using Python

With jpy you can implement Java interfaces using Python. We instantiating Java (proxy) objects from Python modules or classes. If you call methods of the resulting Java object, jpy will delegate the calls to the matching Python module functions or class methods. Here is how this works.

Assuming we have a Java interface `PlugIn.java`

```
public interface PlugIn {
    String[] process(String arg);
}
```

and a Python implementation `bibo_plugin.py`

```
class BiboPlugIn:
    def process(self, arg):
        return arg.split();
```

then we can call the Python code from Java as follows

```
// Import the Python module
PyModule plugInModule = PyLib.importModule("bibo_plugin");

// Call the Python class to instantiate an object
PyObject plugInObj = plugInModule.call("BiboPlugin");

// Create a Java proxy object for the Python object
PlugIn plugIn = plugInObj.createProxy(PlugIn.class);

String[] result = plugIn.process('Abcdefghi jkl mnopqr stuv wxy z');
```

1.2 Current limitations

- Java non-final, static class fields are currently not supported: The reason is that Java classes are represented in jpy's Python API as dynamically allocated, built-in extension types. Built-in extension types cannot have (as of Python 3.3) static, computed attributes which we would need for getting/setting Java static class fields.
- Public final static fields are represented as normal (non-computed) type attributes: Their values are Python representations of the final Java values. The limitation here is, that they can be overwritten from Python, because Python does not know final/constant attributes. This could only be achieved with computed attributes, but as said before, they are not supported for built-in extension types.
- It is currently not possible to shutdown the Java VM from Python and then restart it.

1.3 Other projects with similar aims

- JPype - allow python programs full access to java class libraries
- Jython - Python for the Java Platform
- JyNI - Jython Native Interface
- Jynx - improve integration of Java with Python

CHAPTER 2

Installation

jpy's installation is currently the full build process from sources. We will try to ease the installation process in the future.

After successful installation you will be able

- to use Java from Python by importing the jpy module `import jpy` and
- to use Python from Java by importing the jpy Java API classes `import org.jpy.*;` from `jpy.jar` on your Java classpath.

2.1 Getting the Sources

The first step is to clone the jpy repository or download the sources from the jpy Project page. We recommend you clone the repository using the git tool:

```
git clone https://github.com/bcdev/jpy.git
```

If you don't want to use git, you can also download stable source releases from the jpy releases page on GitHub.

In the following it is assumed that the jpy sources are either checked out or unpacked into a directory named `jpy`.

2.2 Build from Sources

Change into the checkout directory (`cd jpy`) and follow the build steps below. After successful build, the `build` directory will contain the platform-dependent jpy versions:

build/

lib-*os-platform-python-version*/ jpy.so (Unixes only) jdl.so (Unixes only) jpy.pyd (Windows only) jdl.pyd (Windows only) jpyutil.py jpyconfig.py jpyconfig.properties

2.3 Running Java from Python

In order to use jpy from Python you will have to add the respective directory `lib-``*os-platform*-“*python-version*` for your platform and Python version to your Python path. This can be done either programmatically in Python, e.g.

import sys sys.path.append(‘build/lib-*os-platform-python-version*’)

You could also alter the `PYTHONPATH` environment variable,

export PYTHONPATH=$PYTHONPATH:build/lib-*os-platform-python-version*

Finally you could copy the contained files into your Python installation’s `site-packages` directory to make jpy permanently available.

2.4 Running Python from Java

To use the jpy Java API, put `lib/jpy.jar` on your classpath. Or if you use Maven add the following dependency to your project:

<dependency> <groupId>org.jpy</groupId> <artifactId>jpy</artifactId> <version>0.8</version>

</dependency>

The jpy Java API requires a maximum of two configuration parameters:

- `jpy.jpyLib` - path to the ‘jpy’ Python module, namely the `jpy.so` (Unix) or `jpy.pyd` (Windows) file
- `jpy.jdlLib` - path to the ‘jdl’ Python module, namely the `jpy.so` (Unix) file. Not used on Windows.

Another optional parameter

- `jpy.debug` - which is either `true` or `false` can be used to output extra debugging information.

All the parameters can be passed directly to the JVM either as Java system properties or by using the single system property

- `jpy.config` - which is a path to a Java properties files containing the definitions of the two parameters named above.

Such property file is also written for each build and is found in `build/lib-<os-platform>-<python-version>/jpyconfig.properties`.

2.4.1 Setting PYTHONHOME

If the environment variable `PYTHONHOME` is not set when you call Python from Java, you may get an error about file system encodings not being found. It is possible to set the location of Python from your Java program. Use `PyLib.setPythonHome(pathToPythonHome)` to do that, where `pathToPythonHome` is a `String` that contains the location of the Python installation.

Build for Linux / Darwin

You will need

- Python 2.7 or 3.3 or higher
- Oracle JDK 7 or higher
- Maven 3 or higher

- For Linux: `gcc`
- For Darwin: Xcode

To build and test the jpy Python module use the following commands:

```
export JDK_HOME=<path to the JDK installation directory>
export JAVA_HOME=$JDK_HOME
python setup.py --maven build
```

where `JAVA_HOME` is used by Maven and `JDK_HOME` by `setup.py`. On Darwin, you may find the current JDK/Java home using the following expression:

```
export JDK_HOME=$(/usr/libexec/java_home)
```

If you encounter linkage errors during setup saying that something like a `libjvm.so` (Linux) or `libjvm.dylib` (Darwin) cannot be found, then you can try adding its containing directory to the `LD_LIBRARY_PATH` environment variable, e.g.:

```
export LD_LIBRARY_PATH=$JDK_HOME/jre/lib/server:$LD_LIBRARY_PATH
```

Build for Microsoft Windows

2.4.2 Python 2.7

You will need

- Python 2.7 or higher (2.6 may work as well but is not tested)
- Oracle JDK 7 or higher (JDK 6 may work as well)
- Maven 3 or higher
- Microsoft Visual C++ 10 or higher

Note that if you build for a 32-bit Python, make sure to also install a 32-bit JDK. Accordingly, for a 64-bit Python, you will need a 64-bit JDK. If you use the free Microsoft Visual C++ Express edition, then you only can build for a 32-bit Python.

Open the command-line and execute:

```
SET VS90COMNTOOLS=%VS100COMNTOOLS%
SET JDK_HOME=<path to the JDK installation directory>
SET JAVA_HOME=%JDK_HOME%
SET PATH=%JDK_HOME%\jre\bin\server;%PATH%
```

Then, to actually build and test the jpy Python module use the following command:

```
python setup.py --maven build
```

2.4.3 Python 3.3 and higher

You will need

- Python 3.3 or higher (3.2 may work as well but is not tested)
- Oracle JDK 7 or higher (JDK 6 may work as well)
- Maven 3 or higher

- Microsoft Windows SDK 7.1 or higher

If you build for a 32-bit Python, make sure to also install a 32-bit JDK. Accordingly, for a 64-bit Python, you will need a 64-bit JDK.

The Python setup tools (`distutils`) can make use of the command-line C/C++ compilers of the free Microsoft Windows SDK. These will by used by `distutils` if the `DISTUTILS_USE_SDK` environment variable is set. The compilers are made accessible via the command-line by using the `setenv` tool of the Windows SDK. In order to install the Windows SDK do the following

1. If you already use Microsoft Visual C++ 2010, make sure to uninstall the x86 and amd64 compiler redistributables first. Otherwise the installation of the Windows SDK will definitely fail. This may also apply to higher versions of Visual C++.
2. Download and install Windows SDK 7.1. (This step failed for me the first time. A second 'repair' install was successful.)
3. Download and install Windows SDK 7.1 SP1.

Open the command-line and execute:

```
"C:\Program Files\Microsoft SDKs\Windows\v7.1\bin\setenv" /x64 /release
```

to prepare a build of the 64-bit version of jpy. Use:

```
"C:\Program Files\Microsoft SDKs\Windows\v7.1\bin\setenv" /x86 /release
```

to prepare a build of the 32-bit version of jpy. Now set other environment variables:

```
SET DISTUTILS_USE_SDK=1
SET JDK_HOME=<path to the JDK installation directory>
SET JAVA_HOME=%JDK_HOME%
SET PATH=%JDK_HOME%\jre\bin\server;%PATH%
```

Then, to actually build and test the jpy Python module use the following command:

```
python setup.py --maven build
```

2.5 Typical Build Problems

Make sure that `JAVA_HOME` and `JDK_HOME` are always set, not only when installing, but also when using jpy. Additionally make sure that your `PATH` environment variable contains the `JAVA_HOME`.

Set environment variables on Windows

Set environment variables on Linux

When used from Python, jpy must be able to find an installed Java Virtual Machine (JVM) on your computer. This is usually the one that has been linked to the Python module during the build process.

If the JVM cannot be found, you will have to adapt the `LD_LIBRARY_PATH` (Unix) or `PATH` (Windows) environment variables to contain the path to the JVM shared libraries. That is `libjvm.dylib` (Darwin), `libjvm.so` (Linux) and `jvm.dll` (Windows). Make sure to use matching platform architectures, e.g. only use a 64-bit JVM for a 64-bit Python.

Otherwise the JVM may be found but you will get error similar to the following one (Windows in this case):

```
>>> import jpy
Exception in thread "main" java.lang.UnsatisfiedLinkError: C:\Python33-amd64\Lib\site-
↪packages\jpy.pyd: Can't load AMD 64-bit .dll on a IA 32-bit platform
```

If you build for Python 2.7, `setup.py` may fail with the following message:

```
C:\Users\Norman\JavaProjects\jpy>c:\Python27-amd64\python.exe setup.py install
Building a 64-bit library for a Windows system
running install
running build
running build_ext
building 'jpy' extension
error: Unable to find vcvarsall.bat
```

This happens, because `distutils` uses an environment variable of an older Microsoft Visual C++ version, namely `VS90COMNTOOLS`. Make sure to it to the value of your current version. For example:

```
SET VS90COMNTOOLS=%VS100COMNTOOLS%
```

`setup.py` may fail with the following message:

```
C:\Users\Norman\JavaProjects\jpy>c:\Python27\python.exe setup.py install
Building a 32-bit library for a Windows system
running install
running build
running build_ext
...
running install_lib
running install_egg_info
Removing c:\Python27\Lib\site-packages\jpy-0.7.2-py2.7.egg-info
Writing c:\Python27\Lib\site-packages\jpy-0.7.2-py2.7.egg-info
Importing module 'jpy' in order to retrieve its shared library location...
Traceback (most recent call last):
  File "setup.py", line 133, in <module>
    import jpy
ImportError: DLL load failed: %1 is not a valid Win32 application
```

Fix this by adding the path to the Java VM shared library (`jvm.dll`) to the `PATH` environment variable:

```
SET PATH=%JDK_HOME%\jre\bin\server;%PATH%
```

CHAPTER 3

Tutorial

Sorry, the jpy tutorial is not yet written. Meanwhile please refer to jpy's Python and Java unit-level tests in order to learn how to use jpy. They are located at

- src/test/python
- src/test/java

3.1 Using jpy with Python

3.1.1 Using the Java Standard Library

3.1.2 Calling your Java Classes from Python

Primitive array parameters that are mutable

Primitive array parameters that are return value

3.2 Using jpy with Java

3.2.1 Getting Started

3.2.2 Using the Python Standard Library

3.2.3 Calling your Python functions from Java

3.2.4 Extending Java with Python

CHAPTER 4

Reference

4.1 Python API

This reference addresses the jpy Python module.

4.1.1 jpy Functions

`jpy.`**`create_jvm`**(*options*)

Create the Java VM using the given *options* sequence of strings. Possible option strings are of the form:

Option	Meaning
`-D<name>=<value>`	Set a Java system property. The most important system property is `java.class.path` to include your Java libraries. You may also consider `java.library.path` if your Java code uses native code provided in shared libraries.
`-verbose[:gc,class,jni]`	Enable verbose output. The options can be followed by a comma-separated list of names indicating what kind of messages will be printed by the JVM. For example, `-verbose:gc,class` instructs the JVM to print GC and class loading related messages. Standard names include: gc, class, and jni.
`-X<value>`	Set a non-standard JVM option which usually begins with `-X` or an underscore. For example, the Oracle JDK/JRE supports `-Xms` and `-Xmx` options to allow programmers specify the initial and maximum heap size. Please refer to the documentation of the used Java Runtime Environment (JRE).

The function throws a runtime error on failure. It has no return value.

Usage example:

```
jpy.create_jvm(['-Xmx512M', '-Djava.class.path=/usr/home/norman/jpy-test/classes
↪'])
```

jpy.**destroy_jvm**()
Destroy the Java Virtual Machine. The function has no effect if the JVM is has not yet been created or has already been destroyed. No return value.

jpy.**get_type**(*name*, *resolve=False*)
Return a type object for the given, fully qualified Java type *name* which is the name of a Java primitive type, a Java class name, or a Java array type name.

Java class names must be fully qualified, e.g. `'java.awt.Point'`. For inner classes a dollar sign is used to separate it from its containing class, e.g. `'java.awt.geom.Ellipse2D$Float'`.

Java array type names have a trailing opening bracket, followed by either a Java class name and a trailing semicolon or followed by one of the primitive type indicators:

- `'Z'`, the Java `boolean` type (an 8-bit Boolean value)
- `'C'`, Java `char` type (a 16-bit unicode character)
- `'B'`, Java `byte` type (an 8-bit signed integer number)
- `'S'`, Java `short` type (a 16-bit signed integer number)
- `'I'`, Java `int` type (a 32-bit signed integer number)
- `'J'`, Java `long` type (a 64-bit signed integer number)
- `'F'`, Java `float` type (a 32-bit floating point number)
- `'D'`, Java `double` type (a 64-bit floating point number)

Examples: `'[java.awt.Point;'` (1d object array), `'[[[F'` (3d float array).

If the returned Java type has public constructors it can be used to create Java object instances in the same way Python objects are created from their types, e.g.:

```
String = jpy.get_type('java.lang.String')
s = String('Hello jpy!')
s = s.substring(0, 5)
```

The returned Java types are also used to access the type's static fields and methods:

```
Runtime = jpy.get_type('java.lang.Runtime')
rt = Runtime.getRuntime()
tm = rt.totalMemory()
```

The returned Java types have a *jclass* attribute which returns the actual Java object. This allows for using the Java types where a Java method would expect a parameter of type *java.lang.Class*.

To instantiate Java array objects, the *jpy.array()* function is used.

Implementation note: All types loaded so far from the Java VM are stored in the global *jpy.types* variable. If the requested type does not already exists in *jpy.types*, the class is newly loaded from the Java VM. The root class of all Java types retrieved that way is *jpy.JType*.

Make sure that *jpy.create_jvm()* has already been called. Otherwise the function fails with a runtime exception.

jpy.**array**(*item_type*, *init*)
Create a Java array object for the given *item_type* and of the given initializer *init*.

item_type may be a *type* object as returned by the *jpy.get_type()* function or a type *name* as it is used for the *jpy.get_type()* function. In addition, the name of a Java primitive type can be used:

- `'boolean'` (an 8-bit Boolean value)

- `'char'` (a 16-bit unicode character)
- `'byte'` (an 8-bit signed integer number)
- `'short'` (a 16-bit signed integer number)
- `'int'` (a 32-bit signed integer number)
- `'long'` (a 64-bit signed integer number)
- `'float'` (a 32-bit floating point number)
- `'double'` (a 64-bit floating point number)

The value for the *init* parameter may bei either an array length in the range `0` to `2**31-1` or a sequence of objects which all must be convertible to the given *item_type*.

Make sure that *jpy.create_jvm()* has already been called. Otherwise the function fails with a runtime exception.

Examples::

```
a = jpy.array('java.lang.String', ['A', 'B', 'C'])
a = jpy.array('int', [1, 2, 3])
a = jpy.array('float', 512)
```

jpy.**cast**(*jobj*, *type*)

Convert a Java object to a Java object with the given Java *type* (type object or type name, see *jpy.get_type()*). If *jobj* is already of *type*, *jobj* is returned. If *jobj* is an instance of *type*, a new wrapper object will be created for this type, otherwise `None` is returned.

This function is useful if you need to convert the *java.util.Object* values returned e.g. by Java collections (implementations of the *java.util.Set*, *java.util.Map*, *java.util.List* & Co.) to specific types. For example:

```
ArrayList = jpy.get_type('java.util.ArrayList')
File = jpy.get_type('java.io.File')
al = ArrayList()
al.add(File('/home/bibo/.jpy'))
item = al.get(0)
# item has type java.util.Object, but actually is a java.io.File
print(type(item))
item = jpy.cast(item, File)
# item has now type java.io.File
print(type(item))
```

Make sure that *jpy.create_jvm()* has already been called. Otherwise the function fails with a runtime exception.

4.1.2 Variables

jpy.**types**

A dictionary that maps Java class names to the respective Python type objects (wrapped Java classes). You should never modify the value of this variable nor directly modify the dictionary's contents.

jpy.**type_callbacks**

Contains callbacks which are called before jpy translates Java methods to Python methods while Java classes are being loaded. These callbacks can be used to annotate Java methods so that jpy can better translate them to Python. This is a powerful but advanced jpy feature that you usually don't have to use.

Consider a Java method:

```
double[] readData(long offset, int length, double[] data);
```

of some Java class `Reader`. From the method's documentation we know that if we pass `null` for *data*, it will create a new array of the given length, read data into it and the return that instance. If we pass an existing array it will be reused instead. From plain Java class introspection, jpy can neither detect if a primitive array parameter is modified by a method and/or whether it shall serve as the method's return value.

To overcome the problem of such semantics inherent to a Java method implementation, jpy uses a dictionary `type_callbacks` in which you can register a Java class name with a callable of following signature:

```
callback(type, method)
```

This can be used to equip specific Java methods of a class with additional information while the Java class is being loaded from the Java VM. *type* is the Java class and *method* is the current class method being loaded. *method* is of type *jpy.JMethod*. The callback should return either `True` or `False`. If it returns `False`, jpy will not add the given method to the Python version of the Java class.

Here is an example:

```
def annotate_Reader_readData_methods(type, method):
    if method.name == 'readData' and method.param_count == 3:
        param_type_str = str(method.get_param_type(1))
        if param_type_str == "<class '[I'>" || param_type_str == "<class '[D'>":
            method.set_param_mutable(2, True)
            method.set_param_return(2, True)
    return True

class_name = 'com.acme.Reader'
jpy.type_callbacks[class_name] = annotate_Reader_readData_methods
# This will invoke the callback above
Reader = jpy.get_type(class_name)
```

Once a method parameter is annotated that way, jpy can transfer the semantics of a Java method to Python. For example:

```
import numpy as np

r = Reader('test.tif')
a = np.array(1024, np.dtype=np.float64)
a = r.read(0, len(a), a)
r.close()
```

Here a call to the `read` method will modify the numpy array's content as desired and return the same array instance as indicated by the Java method's specification.

`jpy.`**`type_translations`**
: Contains callbacks which are called when instantiating a Python object from a Java object. After the standard wrapping of the Java object as a Python object, the Java type name is looked up in this dictionary. If the returned item is a callable, the callable is called with the JPy object as an argument, and the callable's result is returned to the user.

`VerboseExceptions.`**`enabled`**
: If set to true, then jpy will produce more verbose exception messages; which include the full Java stack trace. If set to false, then jpy produces exceptions using only the underlying Java exception's toString method.

`jpy.`**`diag`**
: An object used to control output of diagnostic information for debugging. This variable is only useful for jpy modification and further development.

diag.**flags**
Integer bit-combination of diagnostic flags (see following F_* constants). If this value is not zero, diagnostic messages are printed to the standard output stream for any subsequent jpy library calls. Its default value is `jpy.diag.F_OFF` which is zero.

For example:

```
jpy.diag.flags = jpy.diag.F_EXEC + jpy.diag.F_JVM
```

The following flags are defined:

- `F_OFF` - Don't print any diagnostic messages
- `F_ERR` - Errors: print diagnostic information when erroneous states are detected
- `F_TYPE` - Type resolution: print diagnostic messages while generating Python classes from Java classes
- `F_METH` - Method resolution: print diagnostic messages while resolving Java overloaded methods
- `F_EXEC` - Execution: print diagnostic messages when Java code is executed
- `F_MEM` - Memory: print diagnostic messages when wrapped Java objects are allocated/deallocated
- `F_JVM` - JVM: print diagnostic information usage of the Java VM Invocation API
- `F_ALL` - Print all possible diagnostic messages

4.1.3 Types

You will never have to use the following type directly. But it may be of use to know where they come from when they are referred to, e.g. in error messages.

class jpy.**JType**
This type is the base class for all type representing Java classes. It is actually a meta-type used to dynamically create Python type instances from loaded Java classes. Such derived types are returned by *jpy.get_type()* instead or can be directly looked up in *jpy.types*.

class jpy.**JOverloadedMethod**
This type represents an overloaded Java method. It is composed of one or more *jpy.JMethod* objects.

class jpy.**JMethod**
This type represents a Java method. It is part of a *jpy.JOverloadedMethod*.

name
The method's name. Read-only attribute.

return_type
The method's return type. Read-only attribute.

param_count
The method's parameter count. Read-only attribute.

get_param_type(*i*) → type
Get the type of the *i*-th Java method parameter.

is_param_return(*i*) → bool
Return `True` if arguments passed to the *i*-th Java method parameter will be the return value of the method, `False` otherwise.

set_param_return(*i*, *value*)
Set if arguments passed to the *i*-th Java method parameter will be the return value of the method, with *value* being a Boolean.

is_param_output(*i*) → bool
Return `True` if the arguments passed to the *i*-th Java method parameter is a mere output (and not read from), `False` otherwise.

set_param_output(*i*, *value*)
Set if arguments passed to the *i*-th Java method parameter is a mere output (and not read from), with *value* being a Boolean. Used to optimise Python buffer to Java array parameter passing.

is_param_mutable(*i*) → bool
Return `True` if the arguments passed to the *i*-th Java method parameter is mutable, `False` otherwise.

set_param_mutable(*i*, *value*)
Set if arguments passed to the *i*-th Java method parameter is mutable, with *value* being a Boolean.

class jpy.**JField**
This type represents is used to represent Java class fields.

4.1.4 Type Conversions

This section describes the type possible type conversions made by jpy when Python values are passed as arguments to Java typed parameters. In the tables given below are the generated match values ranging from (types never match) to 100 (full match) when comparing a given Java parameter type (rows) with a provided Python value (columns). These match values are also used for finding the best matching Java method overload for a given Python argument tuple.

Java primitive types

	NoneType	bool	int	float	number
boolean	1	100	10	0	0
char	0	10	100	0	0
byte	0	10	100	0	0
short	0	10	100	0	0
int	0	10	100	0	0
long	0	10	100	0	0
float	0	1	10	90	50
double	0	1	10	100	50

Java object types

	NoneType	bool	int	float	str
java.lang.Boolean	1	100	10	0	0
java.lang.Character	1	10	100	0	0
java.lang.Byte	1	10	100	0	0
java.lang.Short	1	10	100	0	0
java.lang.Integer	1	10	100	0	0
java.lang.Long	1	10	100	0	0
java.lang.Float	1	1	10	90	0
java.lang.Double	1	1	10	100	0
java.lang.String	1	0	0	0	100

java.lang.String | 1 | 0 | 0 | 0 | 100 |

Java primitive array types

	NoneType	seq	buf('b')	buf('B')	buf('u')	buf('h')	buf('H')	buf('i')	buf('I')	buf('l')	buf('L')	buf('q')	buf('Q')	buf('f')	buf('d')
boolean[]	1	10	100	100	0	0	0	0	0	0	0	0	0	0	0
char[]	1	10	0	0	100	80	90	0	0	0	0	0	0	0	0
byte[]	1	10	100	90	0	0	0	0	0	0	0	0	0	0	0
short[]	1	10	0	0	0	100	90	0	0	0	0	0	0	0	0
int[]	1	10	0	0	0	0	0	100	90	100	90	0	0	0	0
long[]	1	10	0	0	0	0	0	0	0	0	0	100	90	0	0
float[]	1	10	0	0	0	0	0	0	0	0	0	0	0	100	0
double[]	1	10	0	0	0	0	0	0	0	0	0	0	0	0	100

If a python buffer is passed as argument to a primitive array parameter, but it doesn't match the buffer types given above, the a match value of 10 applies, as long as the item size of a buffer matches the Java array item size.

Java object array types

For String arrays, if a sequence is matched with a value of 80 if all the elements in the sequence are Python strings.

todo

4.2 Java API

jpy's Java API documentation has been generated from Java source code using the javadoc tool. It can be found here.

CHAPTER 5

How to Modify

5.1 Rebuild Process

jpy's source distribution directory layout uses the Maven common directory structure.

- `setup.py` - Python build/installation script, will compile Python and Java sources, install the libraries and run all unit-level tests.
- `pom.xml` - Maven project file to build the Java sources. Called by `setup.py`.
- `src/main/c` - C source files for the jpy Python API
- `src/test/python` - Python API test cases
- `src/main/java` - Java source files for the jpy Java API
- `src/test/java` - Java API test cases

After changing any source code just run setup again as indicated in the *Build from Sources* process.

After changing signatures of native methods in `src/main/java/org/jpy/PyLib.java`, you need to compile the Java classes and regenerate the C headers for the `PyLib` class using Maven:

```
mvn compile
javah -d src/main/c/jni -v -classpath target/classes org.jpy.PyLib
```

Then always adapt changes `org_jpy_PyLib.c` according to newly generated `org_jpy_PyLib.h` and `org_jpy_PyLib_Diag.h`. Files are found in `src/main/c/jni/`. Then run setup again as indicated above.

5.2 C Programming Guideline

- Follow style used in Python itself
- Python type global variable names: *J<type>_Type*
- Python type instance structs: *JPy_J<type>*

- Python function decl for a type: *J<type>_<FunctionName>(JNIEnv* jenv, JPy_J<type>* <type>, ...)*
- The pointer is always the first parameter, only type slots obtain their *jenv* from *JPy_GetJEnv()*
- Python slots function for a type: *J<type>_<slot_name>(JNIEnv* jenv, JPy_J<type>* self, ...)*
- Usually functions shall indicate errors by returning NULL or -1 on error. Callers can expect that the PyErr_SetError has been set correctly and thus simply return NULL or -1 again. Exception: very simple functions, e.g. *JObj_Check()*, can go without error status indication.
- Naming conventions:
  - **jpy_jtype.h/c - The Java Meta-Type**
    - JPy_JType type
    - JType_xxx() functions
  - **jpy_jobj.h/c - The Java Object Wrapper**
    - JPy_JObj type
    - JObj_xxx() functions
  - **jpy_jmethod.h/c - The Java Method Wrapper**
    - JPy_JMethod type
    - JPy_JOverloadedMethod type
    - JMethod_xxx() functions
    - JOverloadedMethod_xxx() functions
  - **jpy_jfield.h/c - The Java Field Wrapper**
    - JPy_JField type
    - JField_xxx() functions
  - **jpy_conv.h/c - Conversion of Python objects from/to Java values**
    - JPy_From<JType> functions / JPy_FROM_<JTYPE> macros create Python objects (new references!) from Java types
    - JPy_As<JType> functions / JPy_AS_<JTYPE> macros convert from Python objects to Java types
  - **jpy_diag.h/c - Control of outputting diagnostic info**
    - JPy_Diag type
    - JPy_DIAG_F_<name> macros
    - JPy_DIAG_PRINT(flags, format, ...) macros
  - **jpy_module.h/c - The 'jpy' module definition**
    - JPy_xxx() functions
  - jni/org_jpy_PyLib.h - generated by javah from PyLib.java
  - jni/org_jpy_PyLib_Diag.h - generated by javah from PyLib.java
  - jni/org_jpy_PyLib.c - native implementations from PyLib.java

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

A

array() (*in module jpy*), 16

C

cast() (*in module jpy*), 17
create_jvm() (*in module jpy*), 15

D

destroy_jvm() (*in module jpy*), 15
diag (*in module jpy*), 18
diag.flags (*in module jpy*), 18

G

get_param_type() (*jpy.JMethod method*), 19
get_type() (*in module jpy*), 16

I

is_param_mutable() (*jpy.JMethod method*), 20
is_param_output() (*jpy.JMethod method*), 19
is_param_return() (*jpy.JMethod method*), 19

J

JField (*class in jpy*), 20
JMethod (*class in jpy*), 19
JOverloadedMethod (*class in jpy*), 19
JType (*class in jpy*), 19

N

name (*jpy.JMethod attribute*), 19

P

param_count (*jpy.JMethod attribute*), 19

R

return_type (*jpy.JMethod attribute*), 19

S

set_param_mutable() (*jpy.JMethod method*), 20
set_param_output() (*jpy.JMethod method*), 20
set_param_return() (*jpy.JMethod method*), 19

T

type_callbacks (*in module jpy*), 17
type_translations (*in module jpy*), 18
types (*in module jpy*), 17

V

VerboseExceptions.enabled (*in module jpy*), 18