
Pyjnius Documentation

Release 1.0a1

Kivy Team and other contributors

Dec 11, 2017

Contents

1	Installation	3
1.1	Installation on the Desktop	3
1.2	Installation for Android	3
1.3	Installation for Windows	4
2	Quickstart	5
2.1	A minimal example	5
2.2	Automatic recursive inspection	5
3	Android	7
3.1	Get the DPI	7
3.2	Recording an audio file	7
3.3	Playing an audio file	8
3.4	Accessing the Activity	8
3.5	Accelerometer access	9
3.6	Using TextToSpeech	10
4	API	13
4.1	Reflection classes	13
4.2	Reflection functions	15
4.3	Java class implementation in Python	15
4.4	Java signature format	17
4.5	JVM options and the class path	18
4.6	Pyjnius and threads	18
5	Indices and tables	21
	Python Module Index	23

Pyjnius is a Python library for accessing Java classes.

This documentation is divided into different parts. We recommend you to start with *Installation*, and then head over to the *Quickstart*. You can also check *Android* for specific example for the Android platform. If you'd rather dive into the internals of Pyjnius, check out the api documentation.

Pyjnius depends on [Cython](#) and [Java](#).

1.1 Installation on the Desktop

You need the Java JDK and JRE installed (openjdk will do), and Cython. Then, just type:

```
sudo python setup.py install
```

If you want to compile the extension within the directory for any development, just type:

```
make
```

You can run the tests suite to make sure everything is running right:

```
make tests
```

1.2 Installation for Android

To use pyjnius in an Android app, you must include it in your compiled Python distribution. This is done automatically if you build a [Kivy](#) app, but you can also add it to your requirements explicitly as follows.

If you use [buildozer](#), add pyjnius to your requirements in `buildozer.spec`:

```
requirements = pyjnius
```

If you use [python-for-android](#) directly, add pyjnius to the requirements argument when creating a dist or apk:

```
p4a apk --requirements=pyjnius
```

1.3 Installation for Windows

Python and pip must be installed and present in PATH.

1. **Download and install JDK and JRE:** <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
2. **Edit your system and environment variables (use the appropriate Java version):**

Add to Environment Variables:

- JDK_HOME: C:\Program Files\Java\jdk1.7.0_79\
- PATH: C:\Program Files\Java\jdk1.7.0_79\jre\bin\server\

Add to System Variables:

- PATH: C:\Program Files\Java\jdk1.7.0_79\bin\

3. **Download and install Microsoft Visual C++ Compiler for Python 2.7:** <http://aka.ms/vcpython27>
4. Update pip and setuptools:

```
python -m pip install --upgrade pip setuptools
```

5. Install Cython:

```
python -m pip install --upgrade Cython
```

6. Install Pyjnius:

```
pip install pyjnius
```


Eager to get started? This page will give you a good introduction to Pyjnius. It assumes you have already Pyjnius installed. If you do not, head over the *Installation* section.

2.1 A minimal example

A minimal Pyjnius example looks something like this:

```
from jnius import autoclass

Stack = autoclass('java.util.Stack')
stack = Stack()
stack.push('hello')
stack.push('world')

print stack.pop() # --> 'world'
print stack.pop() # --> 'hello'
```

Just save it as *test.py* (or something similar) and run it with your Python interpreter. Make sure not to call your application *jnius.py* because it would conflict with Pyjnius itself:

```
$ python test.py
world
hello
```

2.2 Automatic recursive inspection

Pyjnius uses Java reflection to give you a new `autoclass()` if the return type is not a native type. Let's see this example:

```
System = autoclass('java.lang.System')
System.out.println('Hello World')
```

We only declared the first System class, but we are able to use all the static fields and methods naturally. Let's go deeper:

```
>>> System = autoclass('java.lang.System')
>>> System
<class 'jnius.java.lang.System'>
>>> System.out
<java.io.PrintStream at 0x234df50 jclass=java/io/PrintStream jself=37921360>
>>> System.out.println
<jnius.JavaMethodMultiple object at 0x236adb8>
```

The recursive reflection always gives you an appropriate object that reflects the returned Java object.

Android has a great and extensive API to control devices, your application etc. Some parts of the Android API are directly accessible with Pyjnius but some of them require you to code in Java.

3.1 Get the DPI

The `DisplayMetrics` contains multiple fields that can return a lot of information about the device's screen:

```
from jnius import autoclass
DisplayMetrics = autoclass('android.util.DisplayMetrics')
metrics = DisplayMetrics()
print 'DPI', metrics.getDeviceDensity()
```

Note: To access nested classes, use \$ like: `autoclass('android.provider.MediaStore$Images$Media')`.

3.2 Recording an audio file

By looking at the [Audio Capture](#) guide for Android, you can see the simple steps for recording an audio file. Let's do it with Pyjnius:

```
from jnius import autoclass
from time import sleep

# get the needed Java classes
MediaRecorder = autoclass('android.media.MediaRecorder')
AudioSource = autoclass('android.media.MediaRecorder$AudioSource')
OutputFormat = autoclass('android.media.MediaRecorder$OutputFormat')
AudioEncoder = autoclass('android.media.MediaRecorder$AudioEncoder')
```

```
# create out recorder
mRecorder = MediaRecorder()
mRecorder.setAudioSource(AudioSource.MIC)
mRecorder.setOutputFormat(OutputFormat.THREE_GPP)
mRecorder.setOutputFile('/sdcard/testrecorder.3gp')
mRecorder.setAudioEncoder(AudioEncoder.AMR_NB)
mRecorder.prepare()

# record 5 seconds
mRecorder.start()
sleep(5)
mRecorder.stop()
mRecorder.release()
```

And tada, you'll have a `/sdcard/testrecorder.3gp` file!

3.3 Playing an audio file

Following the previous section on how to record an audio file, you can read it using the Android Media Player too:

```
from jnius import autoclass
from time import sleep

# get the MediaPlayer java class
MediaPlayer = autoclass('android.media.MediaPlayer')

# create our player
mPlayer = MediaPlayer()
mPlayer.setDataSource('/sdcard/testrecorder.3gp')
mPlayer.prepare()

# play
print 'duration:', mPlayer.getDuration()
mPlayer.start()
print 'current position:', mPlayer.getCurrentPosition()
sleep(5)

# then after the play:
mPlayer.release()
```

3.4 Accessing the Activity

This example will show how to start a new Intent. Be careful: some Intents require you to setup parts in the *Android-Manifest.xml* and have some actions performed within your Activity. This is out of the scope of Pyjnius but we'll show you what the best approach is for playing with it.

Using the Python-for-android project, you can access the default *PythonActivity*. Let's look at an example that demonstrates the *Intent.ACTION_VIEW*:

```
from jnius import cast
from jnius import autoclass

# import the needed Java class
```

```

PythonActivity = autoclass('org.renpy.android.PythonActivity')
Intent = autoclass('android.content.Intent')
Uri = autoclass('android.net.Uri')

# create the intent
intent = Intent()
intent.setAction(Intent.ACTION_VIEW)
intent.setData(Uri.parse('http://kivy.org'))

# PythonActivity.mActivity is the instance of the current Activity
# BUT, startActivity is a method from the Activity class, not from our
# PythonActivity.
# We need to cast our class into an activity and use it
currentActivity = cast('android.app.Activity', PythonActivity.mActivity)
currentActivity.startActivity(intent)

# The website will open.

```

3.5 Accelerometer access

The accelerometer is a good example that shows how to write a little Java code that you can access later with Pyjnius.

The `SensorManager` lets you access the device's sensors. To use it, you need to register a `SensorEventListener` and overload 2 abstract methods: `onAccuracyChanged` and `onSensorChanged`.

Open your python-for-android distribution, go in the `src` directory, and create a file `org/myapp/Hardware.java`. In this file, you will create everything needed for accessing the accelerometer:

```

package org.myapp;

import org.renpy.android.PythonActivity;
import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;

public class Hardware {

    // Contain the last event we got from the listener
    static public SensorEvent lastEvent = null;

    // Define a new listener
    static class AccelListener implements SensorEventListener {
        public void onSensorChanged(SensorEvent ev) {
            lastEvent = ev;
        }
        public void onAccuracyChanged(Sensor sensor , int accuracy) {
        }
    }

    // Create our listener
    static AccelListener accelListener = new AccelListener();

    // Method to activate/deactivate the accelerometer service and listener
    static void accelerometerEnable(boolean enable) {

```

```
Context context = (Context) PythonActivity.mActivity;
SensorManager sm = (SensorManager) context.getSystemService(Context.SENSOR_
↳SERVICE);
Sensor accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

    if (accel == null)
        return;

    if (enable)
        sm.registerListener(accelListener, accel, SensorManager.SENSOR_DELAY_
↳GAME);
    else
        sm.unregisterListener(accelListener, accel);
}
```

So we created one method named *accelerometerEnable* to activate/deactivate the listener. And we saved the last event received in *Hardware.lastEvent*. Now you can use it in Pyjnius:

```
from time import sleep
from jnius import autoclass

Hardware = autoclass('org.myapp.Hardware')

# activate the accelerometer
Hardware.accelerometerEnable(True)

# read it
for i in xrange(20):

    # read the last event
    lastEvent = Hardware.lastEvent

    # we might not get any events.
    if not lastEvent:
        continue

    # show the current values!
    print lastEvent.values

    sleep(.1)

# don't forget to deactivate it
Hardware.accelerometerEnable(False)
```

You'll obtain something like this:

```
[-0.0095768067985773087, 9.4235782623291016, 2.2122423648834229]
...
```

3.6 Using TextToSpeech

Same as the audio capture, by looking at the [An introduction to Text-To-Speech in Android](#) blog post, it's easy to do it with Pyjnius:

```
from jnius import autoclass
Locale = autoclass('java.util.Locale')
PythonActivity = autoclass('org.renpy.android.PythonActivity')
TextToSpeech = autoclass('android.speech.tts.TextToSpeech')
tts = TextToSpeech(PythonActivity.mActivity, None)

# Play something in english
tts.setLanguage(Locale.US)
tts.speak('Hello World.', TextToSpeech.QUEUE_FLUSH, None)

# Queue something in french
tts.setLanguage(Locale.FRANCE)
tts.speak('Bonjour tout le monde.', TextToSpeech.QUEUE_ADD, None)
```

This part of the documentation covers all the interfaces of Pyjnius.

4.1 Reflection classes

class jnius.JavaClass

Base for reflecting a Java class. The idea is to subclass this `JavaClass`, add few `JavaMethod`, `JavaStaticMethod`, `JavaField`, `JavaStaticField`, and you're done.

You need to define at minimum the `__javaclass__` attribute, and set the `__metaclass__` to `MetaJavaClass`.

So the minimum class definition would look like:

```
from jnius import JavaClass, MetaJavaClass

class Stack(JavaClass):
    __javaclass__ = 'java/util/Stack'
    __metaclass__ = MetaJavaClass
```

`__metaclass__`

Must be set to `MetaJavaClass`, otherwise, all the methods/fields declared will be not linked to the `JavaClass`.

`__javaclass__`

Represent the Java class name, in the format `'org/lang/Class'`. (eg: `'java/util/Stack'`), not `'org.lang.Class'`.

`__javaconstructor__`

If not set, we assume the default constructor to take no parameters. Otherwise, it can be a list of all possible signatures of the constructor. For example, a reflection of the `String` java class would look like:

```
class String(JavaClass):
    __javaclass__ == 'java/lang/String'
    __metaclass__ = MetaJavaClass
```

```
__javaconstructor__ == (  
    '()V',  
    '(Ljava/lang/String;)V',  
    '([C)V',  
    '([CII)V',  
    # ...  
)
```

class `jnius.JavaMethod`

Reflection of a Java method.

__init__ (*signature, static=False*)

Create a reflection of a Java method. The signature is in the JNI format. For example:

```
class Stack(JavaClass):  
    __javaaclass__ = 'java/util/Stack'  
    __metaclass__ = MetaJavaClass  
  
    peek = JavaMethod('()Ljava/lang/Object;')  
    empty = JavaMethod('()Z')
```

The name associated to the method is automatically set from the declaration within the `JavaClass` itself.The signature can be found with the `javap -s`. For example, if you want to fetch the signatures available for `java.util.Stack`:

```
$ javap -s java.util.Stack  
Compiled from "Stack.java"  
public class java.util.Stack extends java.util.Vector{  
public java.util.Stack();  
    Signature: ()V  
public java.lang.Object push(java.lang.Object);  
    Signature: (Ljava/lang/Object;)Ljava/lang/Object;  
public synchronized java.lang.Object pop();  
    Signature: ()Ljava/lang/Object;  
public synchronized java.lang.Object peek();  
    Signature: ()Ljava/lang/Object;  
public boolean empty();  
    Signature: ()Z  
public synchronized int search(java.lang.Object);  
    Signature: (Ljava/lang/Object;)I  
}
```

class `jnius.JavaStaticMethod`

Reflection of a static Java method.

class `jnius.JavaField`

Reflection of a Java field.

__init__ (*signature, static=False*)

Create a reflection of a Java field. The signature is in the JNI format. For example:

```
class System(JavaClass):  
    __javaaclass__ = 'java/lang/System'  
    __metaclass__ = MetaJavaClass  
  
    out = JavaField('()Ljava/io/InputStream;', static=True)
```

The name associated to the method is automatically set from the declaration within the `JavaClass` itself.

class `jnius.JavaStaticField`
Reflection of a static Java field

class `jnius.JavaMultipleMethod`
Reflection of a Java method that can be called from multiple signatures. For example, the method `getBytes` in the `String` class can be called from:

```
public byte[] getBytes(java.lang.String)
public byte[] getBytes(java.nio.charset.Charset)
public byte[] getBytes()
```

Let's see how you could declare that method:

```
class String(JavaClass):
    __javaclass__ = 'java/lang/String'
    __metaclass__ = MetaJavaClass

    getBytes = JavaMultipleMethod([
        '(Ljava/lang/String;) [B',
        '(Ljava/nio/charset/Charset;) [B',
        '() [B']
```

Then, when you will try to access to this method, we'll take the best method available according to the type of the arguments you're using. Internally, we are calculating a "match" score for each available signature, and take the best one. Without going into the details, the score calculation look like:

- a direct type match is +10
- a indirect type match (like using a *float* for an *int* argument) is +5
- object with unknown type (`JavaObject`) is +1
- otherwise, it's considered as an error case, and return -1

4.2 Reflection functions

`jnius.autoclass` (*name*)

Return a `JavaClass` that represent the class passed from *name*. The name must be written in the format: *a.b.c*, not *a/b/c*.

```
>>> from jnius import autoclass
>>> autoclass('java.lang.System')
<class 'jnius.java.lang.System'>
```

`autoclass` can also represent a nested Java class:

```
>>> autoclass('android.provider.Settings$Secure')
<class 'jnius.reflect.android.provider.Settings$Secure'>
```

4.3 Java class implementation in Python

class `jnius.PythonJavaClass`

Base for creating a Java class from a Python class. This allow to implement java interface completely in Python.

In reality, you'll create a Python class that mimic the list of declared `__javainterfaces__`. When you'll give an instance of this class to Java, Java will just accept it and call the interfaces methods as declared. Under the hood, we are catching the call, and redirecting to use your declared Python method.

Your class will act as a Proxy to the Java interfaces.

You need to define at minimum the `__javainterfaces__` attribute, and declare java methods with the `java_method()` decorator.

Note: Static methods and static fields are not supported

For example, you could implement the `java/util/ListIterator` interface in Python like that:

```
from jnius import PythonJavaClass, java_method

class PythonListIterator(PythonJavaClass):
    __javainterfaces__ = ['java/util/ListIterator']

    def __init__(self, collection, index=0):
        super(TestImplemIterator, self).__init__()
        self.collection = collection
        self.index = index

    @java_method('()Z')
    def hasNext(self):
        return self.index < len(self.collection.data) - 1

    @java_method('()Ljava/lang/Object;')
    def next(self):
        obj = self.collection.data[self.index]
        self.index += 1
        return obj

# etc...
```

`__javainterfaces__`

List of the Java interfaces you want to proxify, in the format `'org/lang/Class'`. (eg: `'java/util/Iterator'`), not `'org.lang.Class'`.

`__javacontext__`

Indicate which class loader to use: `'system'` or `'app'`, default to `'system'`:

- By default, we assume that you are going to implement a Java interface declared in the Java API. It will use the `'system'` class loader.
- On android, all the java interfaces that you ship within the APK are not accessible with the system class loader, but with the application thread class loader. So if you wish to implement a class from an interface you've done in your app, use `'app'`.

`jnius.java_method` (*java_signature*, *name=None*)

Decoration function to use with `PythonJavaClass`. The *java_signature* must match the wanted signature of the interface. The *name* of the method will be the name of the Python method by default. You can still force it, in case of multiple signature with the same Java method name.

For example:

```
class PythonListIterator(PythonJavaClass):
    __javainterfaces__ = ['java/util/ListIterator']
```

```
@java_method('()Ljava/lang/Object;')
def next(self):
    obj = self.collection.data[self.index]
    self.index += 1
    return obj
```

Another example with the same Java method name, but 2 different signatures:

```
class TestImplem(PythonJavaClass):
    __javainterfaces__ = ['java/util/List']

    @java_method('()Ljava/util/ListIterator;')
    def listIterator(self):
        return PythonListIterator(self)

    @java_method('(I)Ljava/util/ListIterator;',
                 name='ListIterator')
    def listIteratorWithIndex(self, index):
        return PythonListIterator(self, index)
```

4.4 Java signature format

Java signatures have a special format that could be difficult to understand at first. Let's see in details. A signature is in the format:

```
(<argument1><argument2><...><return type>
```

All the types for any part of the signature can be one of:

- L<java class>; = represent a Java object of the type <java class>
- Z = represent a java/lang/Boolean;
- B = represent a java/lang/Byte;
- C = represent a java/lang/Character;
- S = represent a java/lang/Short;
- I = represent a java/lang/Integer;
- J = represent a java/lang/Long;
- F = represent a java/lang/Float;
- D = represent a java/lang/Double;
- V = represent void, available only for the return type

All the types can have the / prefix to design an array. The return type can be V or empty.

A signature like:

```
(ILjava/util/List;)V
-> argument 1 is an integer
-> argument 2 is a java.util.List object
-> the method doesn't return anything.

(java.util.Collection;[Ljava/lang/Object;)V
-> argument 1 is a Collection
```

```
-> argument 2 is an array of Object
-> nothing is returned

([B)Z
-> argument 1 is a Byte []
-> a boolean is returned
```

When you implement Java in Python, the signature of the Java method must match. Java provides a tool named *javap* to get the signature of any java class. For example:

```
$ javap -s java.util.Iterator
Compiled from "Iterator.java"
public interface java.util.Iterator{
  public abstract boolean hasNext();
    Signature: ()Z
  public abstract java.lang.Object next();
    Signature: ()Ljava/lang/Object;
  public abstract void remove();
    Signature: ()V
}
```

The signature for methods of any android class can be easily seen by following these steps:

```
1. $ cd path/to/android/sdk/
2. $ cd platforms/android-xx/ # Replace xx with your android version
3. $ javap -s -classpath android.jar android.app.Activity # Replace android.app.
   ↳Activity with any android class whose methods' signature you want to see
```

4.5 JVM options and the class path

JVM options need to be set before *import jnius* is called, as they cannot be changed after the VM starts up. To this end, you can:

```
import jnius_config
jnius_config.add_options('-Xrs', '-Xmx4096')
jnius_config.set_classpath('.', '/usr/local/fem/plugins/*')
import jnius
```

If a classpath is set with these functions, it overrides any CLASSPATH environment variable. Multiple options or path entries should be supplied as multiple arguments to the *add_* and *set_* functions. If no classpath is provided and CLASSPATH is not set, the path defaults to `'.'`. This functionality is not available on Android.

4.6 Pyjnius and threads

```
jnius.detach()
```

Each time you create a native thread in Python and uses Pyjnius, any call to Pyjnius methods will force attachment of the native thread to the current JVM. But you must detach it before leaving the thread, and Pyjnius cannot do it for you.

Example:

```
import threading
import jnius
```

```
def run(...):
    try:
        # use pyjnius here
    finally:
        jnius.detach()
```

If you don't, it will crash on dalvik and ART / Android:

```
D/dalvikvm(16696): threadid=12: thread exiting, not yet detached (count=0)
D/dalvikvm(16696): threadid=12: thread exiting, not yet detached (count=1)
E/dalvikvm(16696): threadid=12: native thread exited without detaching
E/dalvikvm(16696): VM aborting
```

Or:

```
W/art      (21168): Native thread exiting without having called DetachCurrentThread_
↳(maybe it's going to use a pthread_key_create destructor?): Thread[16,tid=21293,
↳Native,Thread*=0x4c25c040,peer=0x677eaa70,"Thread-16219"]
F/art      (21168): art/runtime/thread.cc:903] Native thread exited without calling_
↳DetachCurrentThread: Thread[16,tid=21293,Native,Thread*=0x4c25c040,peer=0x677eaa70,
↳"Thread-16219"]
F/art      (21168): art/runtime/runtime.cc:203] Runtime aborting...
F/art      (21168): art/runtime/runtime.cc:203] (Aborting thread was not attached to_
↳runtime!)
F/art      (21168): art/runtime/runtime.cc:203] Dumping all threads without_
↳appropriate locks held: thread list lock mutator lock
F/art      (21168): art/runtime/runtime.cc:203] All threads:
F/art      (21168): art/runtime/runtime.cc:203] DALVIK THREADS (16):
...

```


CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`

j

jnius, 13

Symbols

`__init__()` (jnius.JavaField method), 14
`__init__()` (jnius.JavaMethod method), 14
`__javaclass__` (jnius.JavaClass attribute), 13
`__javaconstructor__` (jnius.JavaClass attribute), 13
`__javacontext__` (jnius.PythonJavaClass attribute), 16
`__javainterfaces__` (jnius.PythonJavaClass attribute), 16
`__metaclass__` (jnius.JavaClass attribute), 13

A

`autoclass()` (in module jnius), 15

D

`detach()` (in module jnius), 18

J

`java_method()` (in module jnius), 16
`JavaClass` (class in jnius), 13
`JavaField` (class in jnius), 14
`JavaMethod` (class in jnius), 14
`JavaMultipleMethod` (class in jnius), 15
`JavaStaticField` (class in jnius), 14
`JavaStaticMethod` (class in jnius), 14
`jnius` (module), 13

P

`PythonJavaClass` (class in jnius), 15