JyNI

Jython Native Interface

Title info
- short
- full
- logo

title

contact

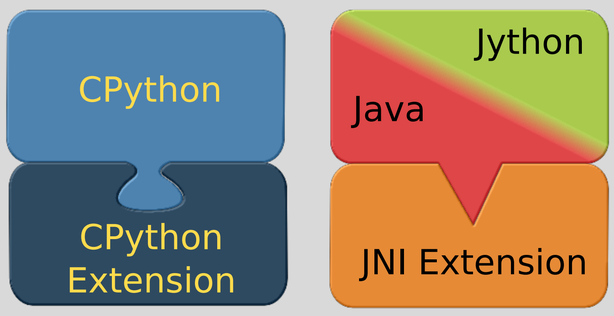
author → Stefan Richthofer

email → contact@jyni.org

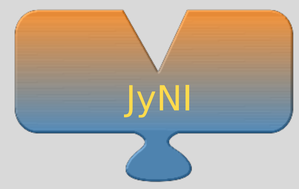
home → www.jyni.org

description

Contact info



JyNI is a compatibility layer with the goal to enable Jython to use native CPython extensions like NumPy or SciPy. This way we aim to enable scientific Python code to run on Jython.
Since Java is rather present in industry, while Python is more present in science, JyNI is an important step to lower the cost of using scientific code in industrial environments.



Non-technical info

no

Start

Wanna read technical info?

yes

Usage, implementation, capabilities?

usage

Thanks to Jython's hooking capabilities, it is sufficient to place JyNI.jar on the classpath (and some native libraries on the library path) when Jython is launched.
Then Jython should “magically” be able to load native extensions, as far as the needed Python C-API is already implemented by JyNI.

No recompilation, no forking – it just works with original Jython and original extensions.

capabilities

JyNI is currently available for Linux only. Once it is sufficiently complete and stable, we will work out a cross platform version compilable on Windows, MacOS and others.
The following built-in types are already supported:

- PyString
- PyTuple
- PyDict
- PyList
- PyBool
- PyInt
- PyLong
- PyFloat
- PyComplex
- PySlice
- PyModule
- PySet
- PyFrozenSet
- PyClass
- PyInstance
- PyMethod
- Singletons PyNone, PyNotImplemented, PyEllipsis
- Natively defined types up to some constraints

The function families PyArg_ParseTuple and Py_BuildValue are also supported.

implementation

overview

To create JyNI we took the source code of CPython 2.7 and stripped away all functionality that can be provided by Jython and is not needed for mirroring objects (see below). We kept the interface unchanged and reimplemented it to delegate calls to Jython via JNI and vice versa.
The most difficult thing is to present JNI-jobjects from Jython to extensions such that they look like PyObject* from Python (C-API). For this task, we use the three different approaches explained below, depending on the way a native type is implemented.

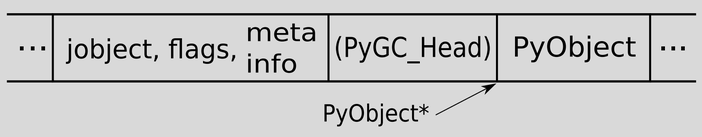
Overview, object sync and lookup

object sync

Object synchronization by mirroring or wrapping.

object lookup

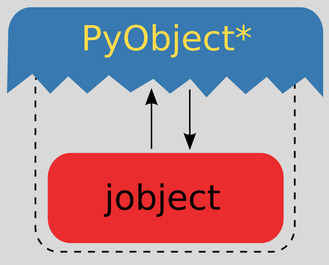
To lookup the corresponding jobject of a PyObject*, we prepend an additional header before each PyObject in memory. (If a PyGC_Head is present, we prepend our header even before that.)



To reserve the additional memory, allocation is adjusted wherever it occurs. The other lookup-direction is done via a hash map.

Python wraps Java

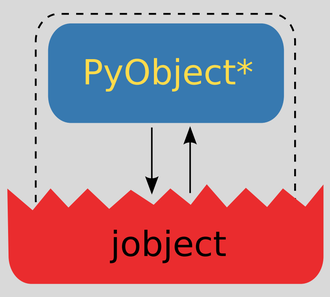
This is the basic approach. It is not feasible if the C-API provides access to the object's structure via macros.



Examples:
PyDict, PySlice, PyModule

Java wraps Python

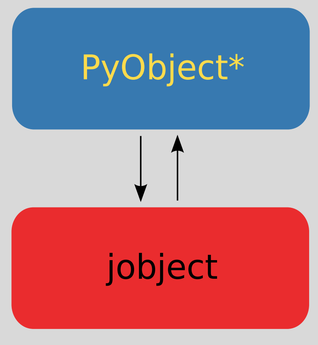
This approach is the only way to deal with types that are unknown to Jython, i.e. types natively defined by an extension.



Example: PyCFunction

objects are mirrored

If the Python C-API provides macros to access an object's data, the object is mirrored. Since most of such objects are immutable, an initial data-synchronization is often sufficient.



Examples:
PyTuple, PyList, PyString, PyInt, PyLong, PyFloat, PyComplex

license

JyNI is available under the GNU GPL v. 3 with classpath exception. See www.jyni.org for more details.

roadmap

The main goal of JyNI is compatibility with NumPy and SciPy, since these extensions are of most scientific importance.

The next steps toward this goal are support for exceptions, the remaining built-in types and garbage collection. Cross-platform support will be addressed as a subsequent goal.