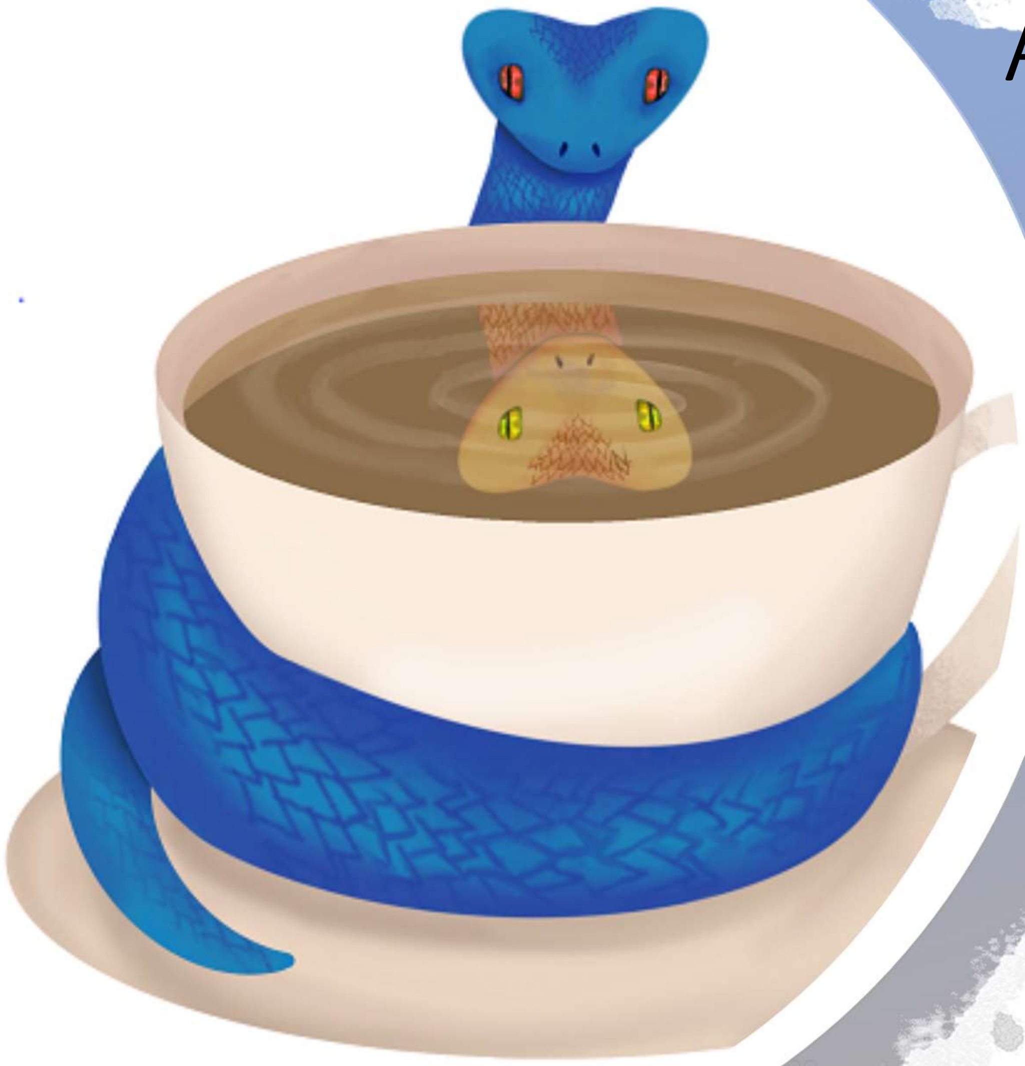


JPype
A Python to Java Bridge

Karl Einar Nelson

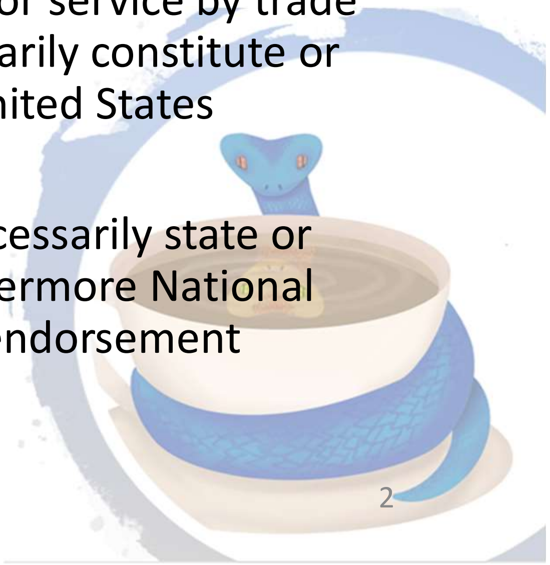
Auspices

This work was produced under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

This work was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

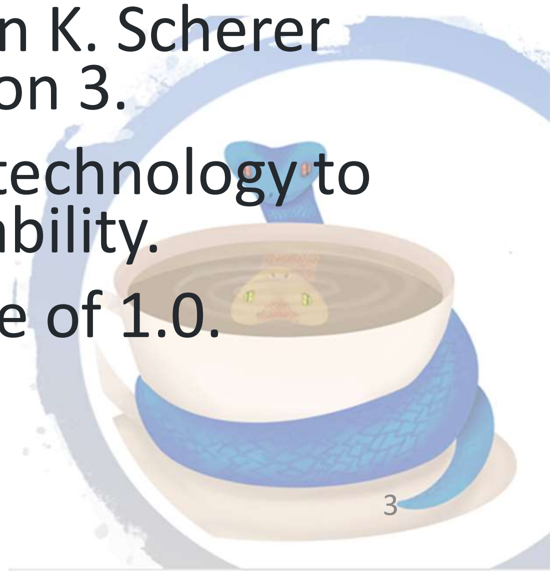
Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC.

The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.



History of JPype

- 2004 -2007 JPype development began by Steve Menard and hosted by Sourceforge.
- 2012 Luis Nell transfers source control to Git to begin development and becomes the maintainer with project hosted on Github.
- 2013 A fork by Thomas Calmant begun to focus on Python 3 support.
- 2014-2015 A group of users lead by Martin K. Scherer rewrote JPype to create bindings for Python 3.
- 2017 Karl E. Nelson joins to rewrite core technology to improve memory usage, speed, maintainability.
- 2020 All core work completed with release of 1.0.



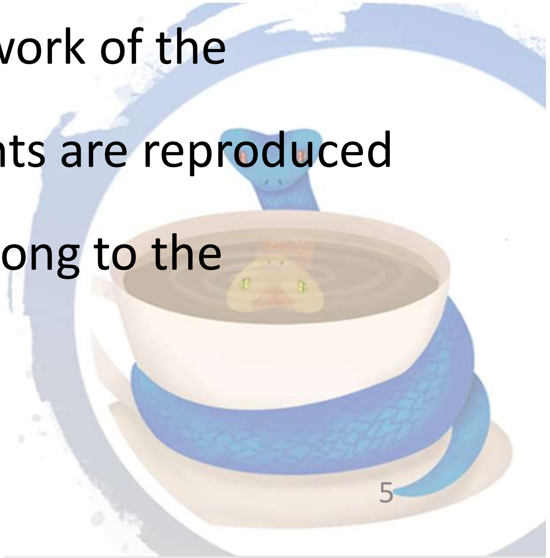
Purpose

Provide complete access to Java libraries and syntax as Python modules for use by Python developers. Intended audience is scientific and engineering programmers who need to freely mix libraries from Python and Java.

License

JPype is licensed under Apache License v2 which allows the licensee to:

- copy, modify and distribute the covered software in source and/or binary forms
- exercise patent rights that would normally only extend to the licensor provided that:
  - all copies, modified or unmodified, are accompanied by a copy of the license
  - all modifications are clearly marked as being the work of the modifier
  - all notices of copyright, trademark and patent rights are reproduced accurately in distributed copies
  - the licensee does not use any trademarks that belong to the licensor.



Project status

- JPype is a community maintained open source project with one maintainer (Martin K. Scherer) and one developer (Karl E. Nelson).
- Karl E. Nelson is a senior scientist at Lawrence Livermore National Laboratory and develops JPype in his personal time.
    - Karl is a government contractor and cannot receive compensation for his JPype work under conflict of interest rules.
- LLNL sponsors this project by providing equipment (computers) its development.   LLNL neither warrantees nor endorses JPype.
- JPype is fully functionally, though additional advanced features such as a “reverse” bridge (Java calling Python) are still under development.
- No commercial support is currently available for JPype but the developer and maintainer frequently answer questions, resolve issues, and enhance features as requested by the community.

Features

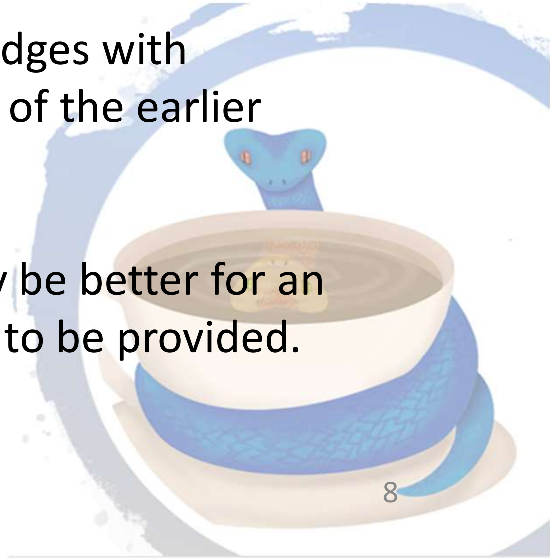
- Provides Java syntax at defined in the Java specification for method resolution and return types.
- All returns except for null (None), True, and False produce a Java “wrapper” which acts as a handle to a Java object.
- Introduces strong typing to Python for working with Java.
- Direct importation of Java classes as Python modules without modification of the Java source.
- Each wrapper is a fully functional natively implemented Python type for speed.
- Class wrappers can by customized using Python decorators on ordinary Python class.

Actively maintained alternatives

Alternative	Advantages	Disadvantage
Jython	Provides a complete Python 2 implementation in Java.	Poor support for using CPython libraries. Losses much of the advantages of Python. Nearing end of lifespan, currently being rebooted.
PyJnius	Focused on providing support for Android. Actively supported.	Missing many basic features for use in scientific development such as multidimensional array support.
Py4J	Remote protocol for controlling the JVM from Python. Multiple JVMs can be spawned and controlled.	Remote operation requires a lot of wrapping to provide a native like access for a Java library.

Only JPype and PyJnius are in the realm of providing forward bridges with transparent presentation of Java syntax in Python, though some of the earlier libraries may also be useful (Jpy, Jep, Javabridge).

Alternatives such as automatic generation of JNI stubs (JCC) may be better for an application than a bridge code depending on the level of access to be provided.



Documentation resources

JPype provides

- A userguide showing usage of each of the JPype features and details on type conversion.
- A quickstart guide to define how to translate Java to Python syntax.
- An installation guide and debugging instructions for handling installation on ordinary and challenging systems.
- A developer guide with technical discussion in case Karl is run over by a bus or retires to mountains.

Key syntax concepts

- Python is a dictionary based language, but not every Java syntax can be represented directly. Representing Java's strong typing in Python requires adding concepts such as casting to Python.
- Wherever there is a naming conflict, a trailing underscore is added.
- Java libraries should not use leading or trailing underscores for names to prevent conflicts with Python syntax.
- Most Java code can just be cut and paste into Python simply by dropping the declaration and new statements.

Concept	Java	Python
Casting from to a Java type	(MyObject) obj;	MyObject@obj
Declaring an array type	Class A = MyObject[].class;	A = MyObject[:]
Creating an array	MyObject[] a = new MyObject[5];	a = MyObject[5]
Access class via reflection	Class C = MyObject.class;	C = MyObject.class_

Proxies

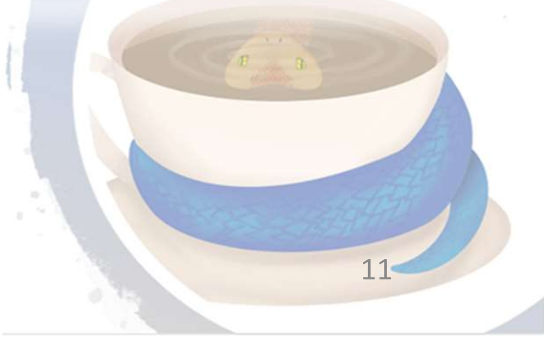
- A proxy is a Python object masquerading as a Java object.
- Proxies implement a Java interface which gives them an identity of how to be treated in Java.
- Proxies can be implemented either by decorating an ordinary class or by wrapping an existing object with JProxy.
- Any Python object can be held as a Java object by wrapping it as "Serializable" interface.

CREATE A PROXY CLASS:

```
@Jimplement(Consumer)
class MyConsumer:
    def accept(self, obj):
        pass
ja = MyConsumer()
```

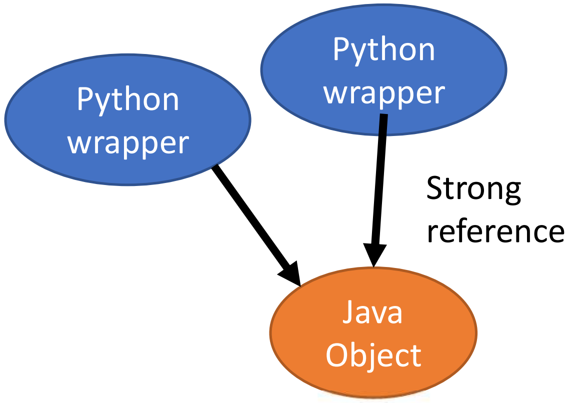
MAKE AN EXISTING OBJECT INTO A PROXY:

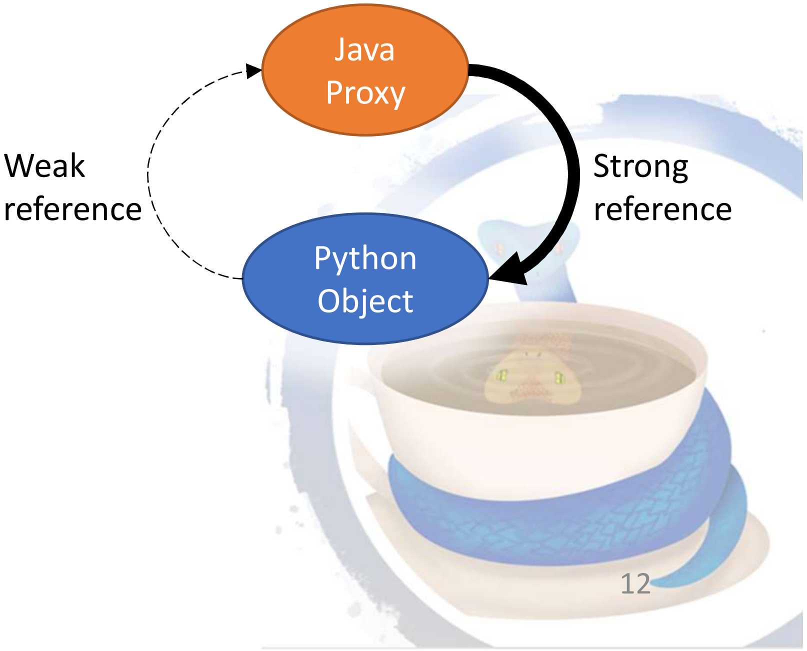
```
a=MyPythonObject()
ja = JProxy(Serializable, inst=a)
```



Memory linkage model

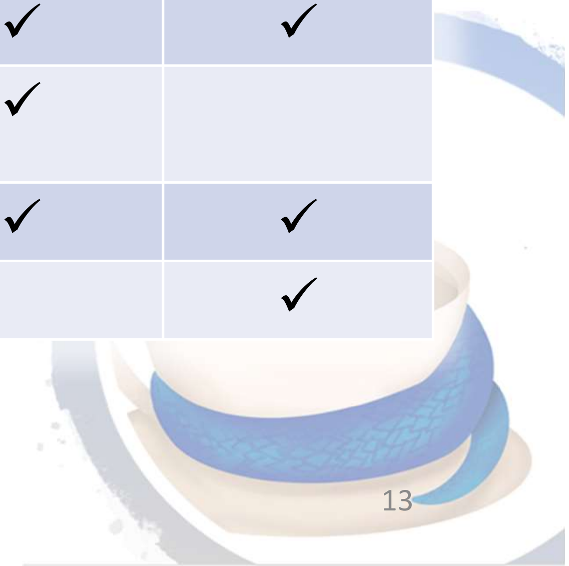
- Each Python wrapper acts as a handle to a Java object and holds a reference.
- Multiple handles can exist for one Java object so each return produces a new handle.
- Python objects can be held in Java space as “Proxies” which implement a Java interface.
- Java proxies hold a reference back to the Python object so that they cannot be lost while Java is using it.
- When a Java proxy is held in Python space it is a “weak” reference to
- Garbage collectors are “linked” such that operation of one will conditionally trigger the other.





Feature comparison with PyJnius

Feature	JPype	PyJnius
Provides access to Python class and objects	✓	✓
Linked memory management model	✓	
Handles Java arrays as native objects (single/multidimension)	✓	
Handles Java buffers as native objects (memoryview)	✓	
Support of serialization of Java and Python objects (Pickling)	✓	
Access to Javadoc and function prototype stubs in Python	✓	
Python can implement Java interfaces (Proxies)	✓	✓
Support for Numpy, Matplotlib, and other scientific Python libraries	✓	
Customization of Java classes to give Python native syntax	✓	✓
Support for Android		✓



Speed

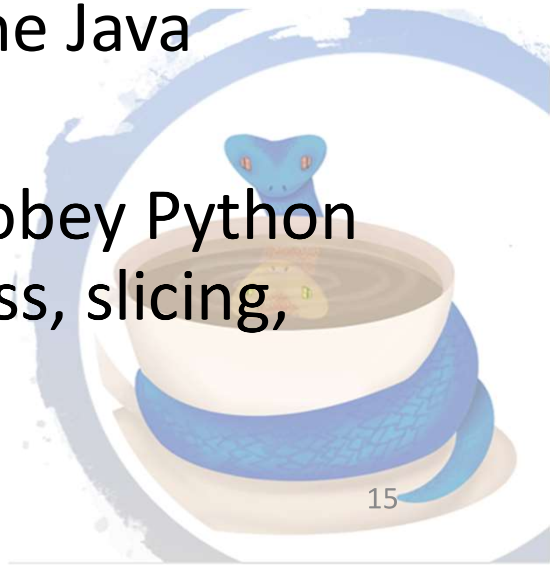
- Speed is not the primary focus of JPype as ease of use, code maintainability, and providing complete features are considered higher priorities.
- JPype interface layer was written in Python until Spring 2020 and was ported to CPython natives improving speed by factors of 3 to 300 depending on the operation. Older benchmarks should be disregarded.
- JPype has implemented an advanced method dispatch algorithm to boost speeds when disambiguating overloaded Java methods from Python.

Operation	JPype	PyJnius
Insert integer in ArrayList	2.24 µs	4.62 µs
Iterate each element of ArrayList	4.66 µs	4.32 µs
Access each element with get()	3.35 µs	6.76 µs

Benchmarking performed on JPype 1.0.1 with speed patch and PyJnius 1.3.0 on an older model laptop.

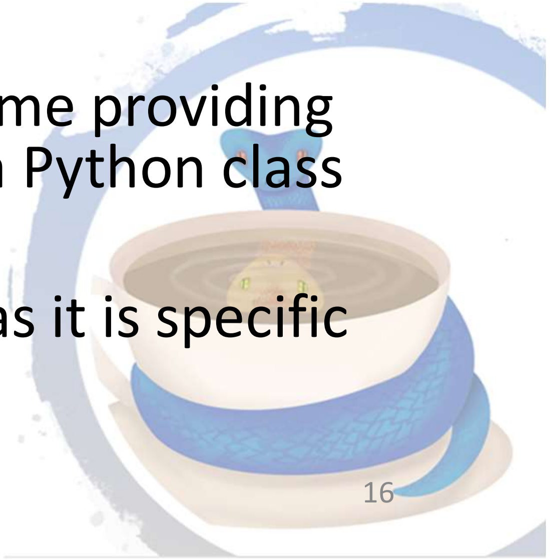
Customizers to give native Python feel

- To keep Java objects from standing out as being different from Python objects, a customizer can be defined for Java classes or interfaces.
- Customizers are declared as Python decorators applied to ordinary Python classes.
- JPype will “steal” the methods from this class and add these native Python methods to the Java wrapper.
- This can be used to make Java classes obey Python syntax such as “with”, “for”, array access, slicing, and list comprehension.



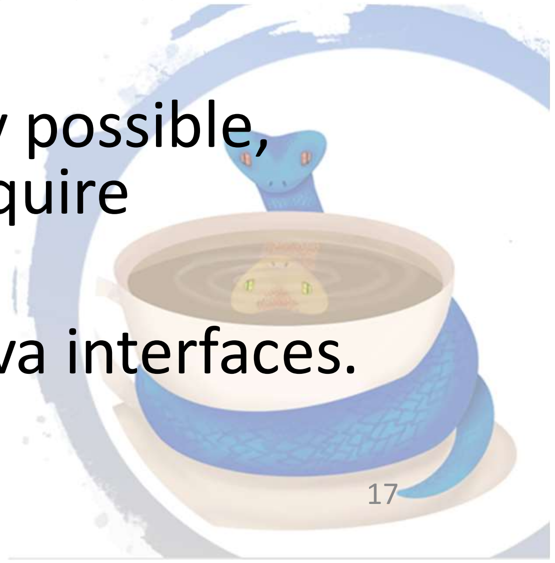
Java slots

- JPype needs to extend many different Python types such as Object, Exception, String, Long, and Float which have different memory layouts.
- Python does not provide a way to add additional native slots nor use multiple inherence on native CPython objects.
- To allow extension CPython objects with multiple memory layouts, JPype overrides the Python memory management system and adds extra memory on Python objects with Java components.
- The new “Java slot” can be resolved in O(1) time providing the extra space needed to store Java data in a Python class and object.
- This approach prevents use of JPype in PyPy as it is specific to CPython.

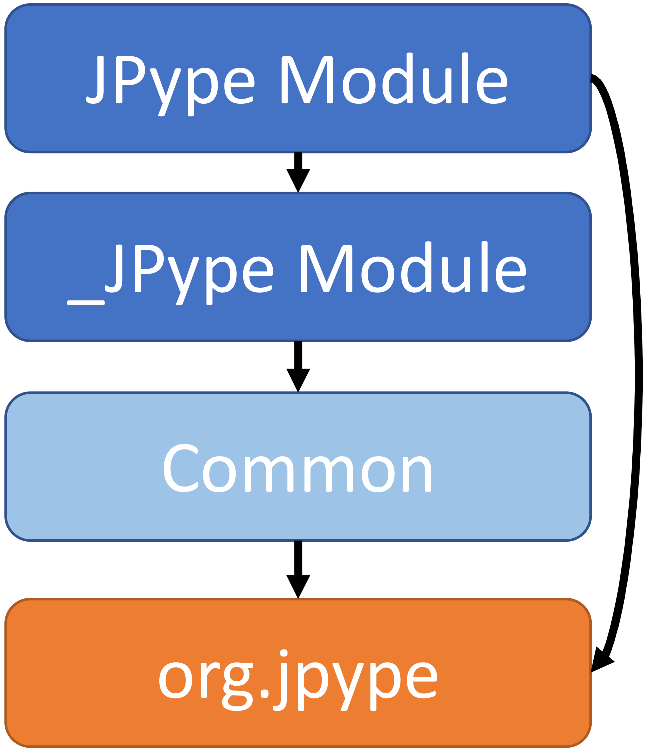


Limitations

- The JVM and Python virtual machine are linked at the process level.
  - It is not possible to stop and restart the JVM.
  - It is not possible to spawn a second JVM copy.
- It is possible programmatically to create an unresolvable reference loop as Python and Java cannot see each others memory space.
  - Python containers holding Java objects should not be wrapped as Proxies.
- Extension of Java classes are not currently possible, thus Java libraries that are designed to require extension of may be limited in use.
- Java can only call Python code through Java interfaces.



Technology Overview



Language	Layer	Function
Python	JPype Module	Presents API and provides hooks for customization. (jpype)
C++/CPython	_JPype Module	Provides base classes and Java slots. Provides all special Python methods. Handles all entry point defense. (native/python)
C++/JNI	Common	Method resolution, array access, type conversions. (native/common)
Java	org.jpype	Utility functions and services. (native/java)

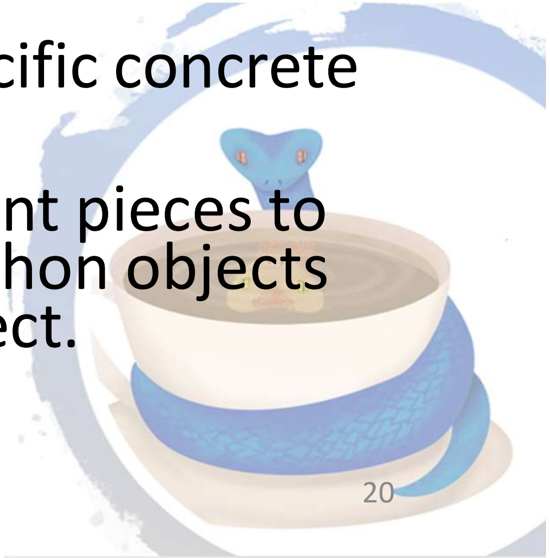
Everything is exposed to Python except for Common, the C++ backend.

Major JPype components

Unit	Language	Function
JPype Context	Java/C++	Global resource to serve as a gatekeeper for all Java services.
TypeManager	Java	Creates C++ backend classes as requested and creates method resolution order cache.
PackageManager	Java	Provides name lookup for Java packages to determine if a resource is a package or class.
ReferenceQueue	Java	Holds references for Python objects and memory buffers so that Python object are constrained to Java lifespans.
Dynamic Classloader	Java	Dynamically loads Java classes after JVM is started and bootstraps JPype class loading.
Javadoc Extractor	Java	Translates Javadoc to RST on demand.

Effective use of Python with JPype

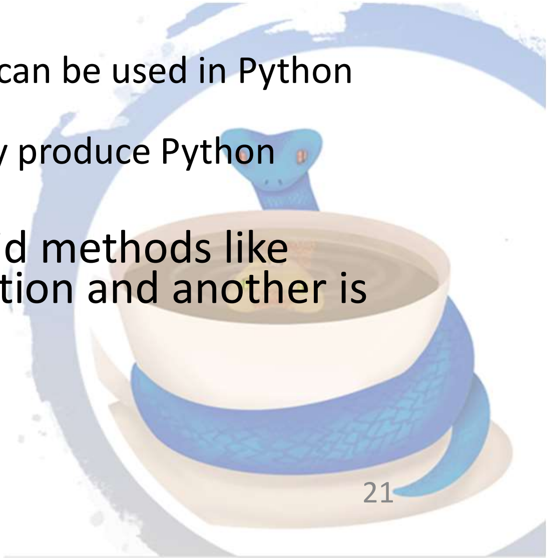
- Minimize the number of times that data needs to be passed across the interface.
  - If the same array is passed multiple times as a Java array convert it once.
  - Don’t force conversion of Python type on return unless required. JPype classes duck type to Python so in many cases it is not necessary.
- Defining a custom conversion keyed off of Python protocol can keep from needed lots of casting and adapters in the user code.
- Add class customizers for interface or specific concrete classes to present a friendly Python API.
- If a Java class needs to be split into different pieces to match Python conventions, then build Python objects that are views which delegate to Java object.



Effective coding in Java to use JPype

Good practices in Java will result in more usable Python wrappers.

- Separate data and algorithm classes.
- Use design patterns philosophy to present common concepts.
- Define input and output classes rather then using ordering on raw arrays to pass parameters and returns.
- Access data through accessors rather than fields.
- Make generous use of interfaces
  - Expose the API as interface views rather than concrete classes.
  - Avoid forcing concrete types for parameters, but instead accept an interface or the least derived container type.
  - Move common concepts to interfaces which can be customized once in Python.
- Reuse existing Java interfaces
  - Use AutoCloseable for resources that need a defined lifespan that can be used in Python “with” statement.
  - Make use of standard collection interfaces which will automatically produce Python collection syntax.
- Don’t overload two methods which do different things. Avoid methods like java.util.List.remove() where one overload is remove by position and another is remove by search.



Credits

JPype is a community effort!

Developers

- Steve Menard
- Karl Einar Nelson

Maintainers

- Luis Nell
- Martin K. Scherer

Artwork

- Athena Nelson

Community contributors

- Bastian Bowe
- Kristi
- Thomas Calmant
- lazerscience
- Koblaid
- Michael Willis (michaelwillis)
- awesomescot
- Joe Quant (joequant)
- Mario Rodas
- David Moss
- Stepan Kolesnik
- Philip Smith
- Dongwon Shin
- rbprogrammer