Python and Java: The Best of Both Worlds

Jim Hugunin
Corporation for National Research Initiatives
hugunin@cnri.reston.va.us

Abstract

This paper describes a new working implementation of the Python language; built on top of the Java language and run-time environment. This is in contrast to the existing implementation of Python, which has been built on top of the C language and run-time environment. Implementing Python in Java has a number of limitations when compared to the current implementation of Python in C. These include about 1.7X slower performance, portability limited by Java VM availability, and lack of compatibility with existing C extension modules. The advantages of Java over C as an implementation language include portability of binary executables, object-orientation in the implementation language to match object-orientation in Python, true garbage collection, run-time exceptions instead of segmentation faults, and the ability to automatically generate wrapper code for arbitrary Java libraries.

1. Why Java?

The current implementation of Python in C is a fairly large and complicated software system. Does Java provide enough advantages when compared to the current implementation in C for it to be worth this significant reimplementation effort? The simplest answer to this question is that the tidal wave of popularity Java has been enjoying recently is a good enough reason to be interested in building a Python system that is 100% pure Java and can capitalize on Java's rise to fame. In addition, Java has a number of significant technical advantages over C, Python's current implementation language. These include portability, object-orientation, robustness, and scripting language friendliness.

1.1. Portable

Java programs compile down to portable executable bytecodes that can run on any computer that supports a Java virtual machine. Unlike ANSI C, which achieves portability across platforms through recompilation of source files, Java's portability is available at the level of binary executables. This means that Java programs can be distributed as binary files that will run on any platform. While code written purely in Python currently enjoys this advantage and will run on any machine with a Python interpreter installed; the wealth of C-based Python extension modules, as well as the central Python interpreter itself, are only portable to multiple platforms after a (sometimes painful) recompilation of the C source.

Another important aspect of Java's portability is the rich set of portable APIs that SUN is defining for the language. They provide standard interfaces for everything from database access to 3d graphics; and work uniformly on any platform that supports Java. More detailed information on these current and planned APIs can be found at JavaSoft's web pages [1].

1.2. Object-Oriented

Java is an object-oriented language with the basic features of encapsulation, polymorphism, and inheritance. This makes it a much better fit to Python's object oriented nature than C. The existing implementations of Python's built-in types in C all show the contortions required to represent classes in a non-object oriented language. Built-in types can be much more cleanly represented in Java by taking advantages of the language's own object oriented features.

One of the standard advantages cited for object-oriented languages is the ability to easily achieve code reuse. I take advantage of this in my implementation by having all Python objects inherit from a single PyObject base class. Furthermore, all of the sequence types (list, tuple, string) inherit from a single PySequence base class.

The current implementation of Python in C has an unfortunate asymmetry between built-in types and Python-defined classes. One aspect of this asymmetry is that users cannot write classes that inherit from built-in types such as lists and dictionaries. Another aspect of the current asymmetry is that instances of built-in types don't have __dict__'s that can be inspected to determine their attributes and methods. A large part of the reason for this asymmetry comes from the implementation difficulties of creating "real" classes in a non-object-oriented language like C. In my implementation of Python in Java, it is relatively easy to make this asymme-

try go away by taking advantages of the object-oriented design of Java for implementing the built-in types.

1.3. Robust

No Java program will ever segmentation fault. Instead it will throw a catchable runtime exception. This is a significant improvement over C where uncatchable, destructive errors occur all too frequently at runtime. It needs to be said that bugs in the Java VM can still lead to segmentation faults, but it's a lot easier to debug a single VM than all of the libraries and applications you'll want to run on top of it. Python programmers are used to this level of robustness in their Python code. However, the frequent inclusion of C-based extension modules in Python leads to these runtime segmentation faults appearing nonetheless.

A key part of Java's robustness is its use of true garbage collection. My Python implementation on top of Java is able to take advantage of this capability in the underlying language and eliminates the need to maintain reference counts on Python objects. This makes writing extension modules significantly easier as there is no longer a need to keep track of the reference counts on Python objects. It also provides true garbage collection to Python code so that circular references no longer lead to memory leaks.

1.4. Glue Language Friendly

One of Python's primary uses is as a glue language to work with existing libraries written in C. The ease with which users can generate Python wrappers around these existing C-based libraries is a frequently cited strength of Python. Nonetheless, these wrappers require a significant amount of hand-coding to make the C libraries available to Python.

Java's design is friendlier to glue languages than C. Both Java's type safety and the existence of a reflection API make it reasonably easy to automatically generate wrappers that allow Python programmers to access Java packages. The ability to do this without any hand-coding of wrapper modules (as is currently the case in C) makes it trivial for Python programmers to use any Java package.

As an example of how Java's type safety makes it possible to automatically generate wrappers for a given function call, consider the following C function declaration:

```
void foo(int *a, int n);
```

This function signature has at least three interpretations, each of which is expressed uniquely in Java. The lack of pointers in Java, and the fact that all arrays have their length implicitly included are what make the Java specification of these signatures unique.

1) A function of a single array of length n (the length is encoded in the array in Java).

```
void foo(int a[]);
```

2) A function of a single int 'n' returning a new int 'a'.

```
int foo(int n);
```

3) A function of two ints 'n' and 'a', returning a new int in 'a'.

```
int foo(int a, int n);
```

Beyond this clear model of what any Java type represents, Java also provides a reflection API to allow glue languages to discover all of the methods and fields implemented by a given class and to dynamically access them at runtime.

2. Implementing Python in/on Java

My implementation of Python in Java consists of three key components. The first is a parser and lexer for the Python grammar. These are pure Java code generated by the JavaCC parser generator [2] freely available from Sun. They are similar to Python's C-code parser and lexer. When run on a Python source file the parser generates a collection of Java objects representing the parser tree.

The second component of the system is a compiler written entirely in Python that traverses this parse tree to output actual Java bytecodes. (Notice that I'm already taking advantage of the power of this system here as I use a compiler written in Python to traverse a parse tree consisting of Java objects produced by a Java-based parser). This is different from the current Python implementation, which outputs its own private bytecode format. By producing Java bytecodes, I'm coming closer to what in the C-world would be accomplished by directly generating machine code. But this is a portable machine code that will run on any Java platform.

The third component of my implementation of Python in Java is a collection of Java-based support classes. These classes provide the implementations for the basic Python objects (lists, integers, instances, classes, …) as well as implementations of handy functions like "print" and "import". The functionality of Python's current

virtual machine is split between these support classes and Java's own virtual machine.

2.1. Compiling Python to Java Bytecodes

Both the Python and Java languages compile to a portable bytecode format. The virtual machines for both of these languages are stack-based. Stack-based architectures seem to be the norm for virtual machines, while most native microprocessors are instead register-based machines. Unlike the Python virtual machine which was designed purely for running Python code, the Java VM was built to be able to run code from a wide variety of languages. I am aware of compiler projects (in various stages of completion) for Scheme, TCL, BASIC, Rexx, Python (described in this paper), ADA and C that target the Java VM.

Let's take a look at how a very simple expression is currently compiled to Python bytecodes by the Python compiler. To keep things as simple as possible, we'll consider the expression "2+2".

Python Bytecodes for 2+2

```
0 LOAD_CONST          0 (2)
3 LOAD_CONST          0 (2)
6 BINARY_ADD
```

This simple expression generates three Python bytecodes. The first two load the Python integer object "2" on to the operand stack. The final opcode adds the top two elements on the stack together. It does this using Python's conventions for dynamic method invocation.

The Java bytecodes for this same operation look surprisingly similar:

Java Bytecodes for 2+2

```
0 iconst_2
1 iconst_2
2 iadd
```

The first two opcodes push the integer "2" on to the operand stack, and the third opcode adds the top two operands together.

Despite the apparent similarities between these two bits of assembly code, the underlying mechanics are in fact quite different. The Java machine is working on literal integer values. It actually pushes the 32-bit binary value 0x00000002 on to the stack to represent the integer 2. The Python code, on the other hand, is manipulating Python integer objects. What it pushes on to the stack is not a raw integer literal, but a pointer to a corresponding integer object.

Furthermore, Python's BINARY_ADD opcode is very different from Java's iadd. The Python opcode dynamically invokes the appropriate code to add the top two objects on the stack together based on their run-time types. The Java opcode on the other hand requires that its two operands are literal 32-bit integers. This difference captures much of the inherent differences between Python's highly interactive/dynamic nature, and Java's static but high-performance design.

Despite these deep underlying differences, the Java VM has facilities to implement Python's dynamic model with minimal overhead. While Java's integers are inappropriate for representing Python's integers, Java objects can readily represent Python integers (and all other Python objects). And while Java's built-in add methods can not implement Python's dynamic method lookup, Java method invocation can be used to achieve the same effects. Making these changes, we see how the Python semantics for "2+2" can be compiled to Java bytecodes.

Java Bytecodes for 2+2 with Python's Semantics

```
     0 getstatic #1 // static field _i2
     3 getstatic #1 // static field _i2
     5 invokevirtual #2
  // PyObject.add(LPyObject;)LPyObject;
```

This implementation will push a Java object corresponding to the Python integer "2" on to the top of the stack, and then do that again. The third opcode will invoke the add method on the top object on the stack, and it's the responsibility of this method to implement Python's "add" semantics.

2.2. Java Class Hierarchy for Python Types

In order to make the above bytecodes work, all of the built-in Python types must be implemented as Java classes. The Java class hierarchy to implement the core Python types is shown below.

- java.lang.Object
  - PyObject
    - PyInteger
    - PyString
    - ...
    - PyDictionary
    - PySequence
      - PyList
      - PyTuple
    - PyClass
    - PyInstance
    - PyFrame

The names of the classes should be familiar to anybody who has worked with Python's existing internals. It's

unfortunate that these names use the "Py" prefix that's so necessary in Python's C implementation. Java has very nice namespace management (similar to the package namespaces provided in Python 1.5). Unfortunately, every Java program automatically imports the "java.lang" namespace. This means that the names "Object", "Class", and "Integer" are already taken in any Java program.

The PyObject class implements all of the basic functionality of Python's dynamic method invocations. This is very similar to Python's existing semantics, though it is simplified because of increased similarities between built-in types and Python defined classes.

2.3. A More Complicated Example

The actual implementation of my Python to Java bytecode translator performs a direct translation from Python to Java bytecodes. Nonetheless, almost all of what it does can be viewed conceptually as a translation of Python to Java. Since it tends to be easier to read Java source code than Java VM assembly code, I'll present the next few examples that way. The translation from Java source code to Java bytecodes is generally fairly straightforward. Remember that the following examples of Java source code are for illustrative purposes only. My Python in Java system never generates Java source files, but only executable Java bytecodes.

I'm going to show how a very simple Python module is conceptually translated to a Java module. Table 1 shows both the code for the Python module and the Java source that corresponds to the Java bytecodes my compiler actually generates.

Line 1 declares a new Java class. Every Python module is implemented as exactly one Java class. Each Python module must also implement the interface PyRunnable which allows the module to be initialized (and requires the module to implement the "run" method as shown here).

Lines 3-6 define the constant pool for this Python module as static fields on the Java class. Each of these static fields holds a Java object which corresponds to some primitive Python constant.

Lines 7-18 define the single method of this Java class. This is the method that is invoked when the module "foo" is imported under Python. This method receives a PyFrame object corresponding to the Python stack frame in which the module should execute. This object is used to hold all Python local and global variables (no effort has been made to take advantage of Java's local variables as they have sufficiently different semantics than Python's dynamic counterparts as to make this difficult).

Line 9 shows how this frame object is used to execute simple variable assignment. The local variable whose name corresponds to the python string "x" is assigned to the Python integer constant 2.

Line 11 shows a slightly more interesting example. Here, the lookup of the local variable x should be obvious. It is added to the Python constant integer 2 using its add method. This "add" method is implemented in PyObject and handles Python's dynamic method invocation semantics. Notice also that the Java VM's execution stack is being used to keep track of the operands in this add operation; and furthermore, the Java VM's garbage collection is being held responsible for handling memory management so that no reference count management is needed.

Line 13 is a simple import statement. All of the interesting work is hidden here in the implementation of Py.importModule(). I don't want to go into the details of this here, I am just using it as a convenient device to get an object into my name space for which I can get and set attributes.

Line 15 is an example of setting an attribute on an object. Those familiar with Python's "__" special methods should recognize the "__setattr__" method immediately. This method works just like the Python version except that it operates on Java objects (some of which might represent PyInstance objects).

Finally, line 17 prints out the attribute that has just been set. Once again, "__getattr__" works just like the corresponding Python special method.

2.4. An Example With Function Definition

As one final example, I will show how a simple Python module that defines a new Python function can be converted to Java. This example is somewhat complicated because I have tried to make it complete. If you're not really interested in the low level details of how Python in Java is implemented, I'd highly recommend skiping this example. The original Python source for this module, as well as its theoretical translation to Java source are shown in Table 2.

Line 1 declares the "doubleit" Java class corresponding to the Python module just as in the previous example. In addition to the PyRunnable interface which was de-

Table 1. Translation of a simple Python module to Java

Python code for module: “foo.py”

```
x = 2
print x+2

import spam
spam.eggs = 2
print spam.eggs
```

Java source corresponding to Java bytecodes which my compiler actually generates: “foo.java”

```
1  public class foo implements PyRunnable {
2     //Declare fields for constants
3     static PyInteger _i2 = new PyInteger(2);
4     static PyString _sx = new PyString(“x”);
5     static PyString _sspam = new PyString(“spam”);
6     static PyString _seggs = new PyString(“eggs”);

7     public void run(PyFrame frame) {
8           //x = 2
9           frame.setlocal(_sx, _i2);

10          //print x+2
11          Py.print(frame.getlocal(_sx).add(_i2), true);

12          //import spam
13          frame.setlocal(_sspam, Py.importModule(_sspam));

14          //spam.eggs = 2
15          frame.getlocal(_sspam).__setattr__(_seggs, _i2);

16          //print spam.eggs
17          Py.print(frame.getlocal(_sspam).__getattr__(_seggs));
18    }
19 }
```

scribed previously, this class also supports the PyFunctionTable interface which is used to support Python defined functions. More details on how this is accomplished are described below.

Lines 3-6 define the constant pool for this module just as in the previous example.

Lines 7-11 are used to add a Python code object to the constant pool. The arguments passed to this new code object are (in order): number of arguments, an array of variable names (including arguments), the filename of the module which defines this code, the name of the code, whether the code supports *args, whether it supports **keywords, an object that can be used to lookup the actual code to be invoked, and an integer index to be passed to that object to indicate which function to actually call. The details of these last two parameters will be explained later.

Lines 12-15 define the actual Java code to be invoked for the “double” Python function. This function will be called when the _xdouble code object is called. Line 14 shows a small optimization in this code that is equivalent to Python’s FAST_LOCALS optimization. The local variable “x” in this function is looked up by a numeric index into the frame variable table rather than by name. This optimization offers a significant speedup for local variable references in both Python in C and my own Python in Java.

Table 2. Translation of a Python module containing a function definition to Java

Python code for module: “doubleit.py”

```
def double(x):
      return x*2

print double(2)
```

Java source corresponding to Java bytecodes which compiler actually generates: “doubleit.java”

```
1  public class doubleit implements PyRunnable, PyFunctionTable {
2     //Declare fields for constants
3     static PyInteger _i2 = new PyInteger(2);
4     static PyString filename = new PyString(“doubleit.py”);
5     static PyString _sx = new PyString(“x”);
6     static PyString _sdouble = new PyString(“double”);

7     static PyFunctionTable table = new doubleit();
8     static PyCode xdouble = new PyTableCode(1, new PyString[] {_x},
9                                             false, false,
10                                            filename, _sdouble,
11                                            table, 0);


12    public PyObject _fdouble(PyFrame frame) {
13          // return x*2
14          return frame.getlocal(0).multiply(_i2);
15    }

16    public PyObject call_function(int index, PyFrame frame) {
17          switch (index) {
18          case 0:
19                return _fdouble(frame);
20          default:
21                throw new PyInternalError(“Illegal function referenced”);
22          }
23    }

24    public void run(PyFrame frame) {
25        //def double(x):
26        frame.setlocal(_sdouble, new PyFunction(frame.f_globals,
27                                                Py.EmptyObjects,
28                                                _xdouble));

29        //print double(2)
30        Py.print(frame.getlocal(_sdouble).__call__(new PyObject[]{_i2}));
31    }
32 }
```

Table 3. A simple “Java” applet implemented in Python

```
1  import java

2  class HelloApplet(java.applet.Applet):
3     def paint(self, g):
4           g.setColor(java.awt.Color.black)
5           g.fill3DRect(5,5,590,100,0)
6           g.setFont(java.awt.Font("Arial", 0, 80))
7           g.setColor(java.awt.Color.magenta)
8           g.drawString("Hello World", 90, 80)
```

Lines 16-23 define the required method for the PyFunctionTable interface. The goal of this method (and this interface) is to fake the effects that are typically realized in C using pointers to functions. The PyRunnable interface is the simplest solution to the problem that Java does not allow references to function to be passed around. Any class that implements this interface can be passed around and used as a function pointer to the “run” method contained by that class. Instead of using the PyFunctionTable interface, I could instead generate a new Java class file for every function pointer that I wanted, but this is very wasteful of space. Using the PyFunctionTable interface, each module contains methods for every code object that it contains. It also contains this “call_function” method that will lookup the appropriate function to call based on an integer index. This allows a single Java class to support as many Python code objects as desired.

Finally, lines 24-31 define the run method that is used to invoke the top-level code in this module. This is equivalent to the same method as in the previous example.

Line 26 creates a new Python function object which is initialized with the current globals dictionary, no default arguments, and the appropriate Python code object from the constant pool.

Finally, line 30 prints out the result of calling this new function object with the single Python integer 2. The arguments to the function call are passed as a Java array of PyObjects. This allows the function to be called with an arbitrary number of arguments despite the fact that Java doesn’t support anything like Python’s “*args”.

3. Integration of Python and Java

Compiling Python to Java bytecodes is only a part of the picture for making Python and Java work together. What is also needed are seamless mechanisms to allow Python code to use Java libraries (see related work by Kevin Butler [5] and Douglas Cunningham [4]) and to allow Python code to be called from Java (I think this is going to be really hard to achieve with any generality in the other two approaches to embedding Java in Python).

Applets provide a good example of where this sort of seamless bi-directional integration is required. In Table 3 I show the Python source code for a simple wokring “Hello World” applet. It will run in any web browser that supports JDK1.1. At the moment this includes SUN’s HotJava browser, and patched versions of Internet Explorer 3.1 and Netscape Navigator 4.0. Before the end of 1997 all major web browsers should support JDK 1.1 without patches.

This example shows how Python classes can create instances of Java classes, and invoke methods on these instances. It also shows how a Python class can subclass from a Java class and override specific methods.

Line 1 imports the “java” package. This package is the root of the standard Java class hierarchy. If I didn’t want to use fully qualified names, I could have written things like “from java.applet import Applet” instead. The new support for packages in Python 1.5 makes it so that this syntax and semantics for importing Java packages is virtually equivalent to that used for Python packages.

Line 2 creates a new Python class “HelloApplet” which subclasses the Java class, “java.applet.Applet”. The ability of Python classes to subclass Java classes is a key part of the seamless integration of the two languages. This also means that Python in Java code can subclass from any built-in class including Python lists and dictionaries (no more UserList.py).

Line 3 implements the "paint" method for this applet object. This overrides the standard implementation of this method in the Java superclass. It accepts a single argument (other than itself) which is a Java graphics object.

Lines 4-8 invoke various methods on this graphic object in order to draw a huge garish magenta on black "Hello World" within the browser where it is running. All of the methods called here are Java methods, implemented in the java.awt package. In line 5, one is called with a Java object corresponding to the color black. In line 6, the "fill3DRect" method is called with five Python integers. These Python Objects are coerced to the appropriate Java primitive integers when making the call.

Line 6 shows the creation of a new instance of the java.awt.Font class. The syntax for instance creation is exactly like that used when creating a new instance of a Python class. The Python objects which are the arguments are appropriately coerced to Java objects or primitives when the class is actually instantiated.

4. Separation of Python and Java VM's

By choosing to implement Python in Java by running Java bytecodes directly on top of the Java virtual machine, I am able to utilize the machinery of the Java VM to handle many tasks that the current Python VM in C must deal with itself.

The Java VM takes care of memory management using a true garbage collection scheme. This means that there is no need to deal with reference counting at run-time for code that is implemented in Python or Java.

The Java VM handles the immediate operation stack. This means that I don't need to worry about managing a stack of intermediate results. It also handles multi-threading.

Finally, the Java VM deals with much of the problem of exception handling. Every try/except clause in Python is implemented as a try/catch clause at the level of the Java VM. The one great limitation is that the Java machinery must trap all exceptions and then subsequent support code is used to determine the actual type of the Python exception thrown and invoke the appropriate except clause, or reraise the exception to the next stack frame.

There are a number of additional function performed by the standard Python VM that do not map nicely onto the Java VM. These functions are implemented by Java support code that is a key part of my Python in Java system.

As mentioned above, part of this support code is needed for handling specific Python exceptions. It also includes support code for dynamic method invocation.

Finally, this support code must manage a separate Python call frame stack. A key part of managing the call frame stack is handling local and global variable access. This is also done in the support code. It is unfortunate that the Java VM's ability to manipulate local variables can't be used here instead. There seem to be a small number of unresolved issues related to the very dynamic nature of namespaces in Python that make it much easier to just implement my own local variables.

5. Current Status of Implementation

I can compile reasonably complicated Python modules directly to Java bytecodes. The resulting code runs about half as fast as under the standard Python 1.4 in C distribution. In addition, the Java ".class" file is about twice as large as the corresponding Python ".pyc" file. On the pystone benchmark found in the standard Python distribution, I found that my Python in Java implementation obtains 941 PyStones running on a 200MHz PentiumPro under Windows NT 4.0. The standard Python 1.4 implementation in C obtains 1595 PyStones running on the same machine under the same operating system. The Python in Java implementation is about 1.7 times slower than the standard Python in C. I timed the Java implementation running under Microsoft's second beta release of their JDK1.1 compliant virtual machine with JIT compilation enabled. This is the fastest Java VM that I could find at the time of writing this paper.

6. Conclusions

6.1. Python in Java's Disadvantages

My new implementation of Python in Java has a number of disadvantages when compared to the existing Python in C system. Some of these are the frequently cited disadvantages of Java vs. C. It is not compatible with existing Python modules that have been written in C. Java's portability is great, but there are still a number of platforms where it is either not supported at all, or only supported poorly. Finally, this system is slower than the current C-based implementation (by a factor of 1.7).

The portability and speed problems are likely to go away as people continue to work on improving the Java runtime systems. As anecdotal evidence, the first Java VM that I used for my experiments in April 1997 is

about 2.5 times slower than the Java VM I'm using today (September 1997). As far as portability is concerned, the Kaffe project [5] provides a free portable implementation of the Java VM that is relatively easy to get running on a wide variety of platforms.

The problem of backwards compatibility is more difficult to address. Personally, I feel that Python in Java has enough advantages for module writers (garbage collection, true exceptions, object-orientation, portable binaries, …) that they will be happy to reimplement their systems. Whether this is true or not remains to be seen.

6.2. Python in Java's Advantages

Using Java as the underlying systems language for Python has a number of advantages over the current implementation of Python in C. First and foremost of these in my mind is the opportunity to ride the Java popularity wave and let Python code run everywhere there's a Java VM. It also makes the rich set of portable Java API's available from within Python.

There is also a nice collection of technical reasons why Java is a superior implementation language for Python than C. These include Java's binary portability, thread-safety, object-orientation, true exceptions, garbage collection, and friendliness to glue languages. More questions need to be answered before I can make a convincing argument that Python 2.0 should be implemented in Java rather than C. Nonetheless, I think that Java offers many advantages for Python as both an implementation language and a widely available run-time platform.

7. References

[1] Java API Overview and Schedule; JavaSoft; http://www.javasoft.com/products/api-overview.

[2] Java Compiler Compiler (JavaCC); Sriram Sankar, Sreenivasa Viswanadha, Rob Duncan; http://www.suntest.com/JavaCC/.

[3] PyJava: Java Embedded in Python; Kevin Butler; http://students.cs.byu.edu/~butler/jni/PyJava.html.

[4] Java Python Interface (JPI); Douglas Cunningham; http://www.ndim.edrc.cmu.edu/dougc/jpi/.

[5] Kaffe: A free virtual machine to run Java code; Tim Wilkinson; http://www.kaffe.org.