Using Python with Other Languages

Vishesh Yadav | Siddhant Sanyam

PyCon India 2011

15 September 2011

Guidelines

- Download binaries, packages and tutorial examples
- Linux preferred
- We'll cover many topics, so we can be fast
- Tweet @pyconindia #pyconindia

What you should know?

- Some Python experience
- Experience with atleast one language
    - C
    - C++
    - Java

Why use more than one language?

- Performance boost
- Better productivity
- Extending functionalities
- Make your application scriptable

Examples

- Mozilla Firefox, Mozilla Thunderbird
- Vim, GNU Emacs, Kate . . .
- GIMP, Inkscape, Adobe Photoshop . . .
- Qt Frameworks, KDE
- WeeChat, XChat, irssi . . .

Why Python for this?

- “Python fits your brain” - Bruce Eckel
- Easy to learn and get started
- Simple and robust syntax
- Targets all kinds of developers
  - Beginners to Advanced
  - Computer Programming for Everyone (C4PE)
- Powerful enough to do anything (blah almost!)
- Cross Platform

How is it done?

- Depends on Language and Python Implementation
- For CPython
  - Python/C API with C/C++
  - Various libraries and tools
    - ctypes module
    - Boost.Python
    - SWiG and SIP
    - Pyrex
    - and many more . . .
- For Java
  - Jython
  - JPype
  - Jepp
  - and many more (we dont know many)
- For .NET, IronPython

Extending and Embedding

- Extending
  1. Convert data value from Python to 2nd language
  2. Perform a function call to 2nd language routine using converted value
  3. Convert data from the call to Python
- Embedding
  1. Convert data values from 2nd language to Python
  2. Perform a function call to Python interface routing using converted value
  3. Convert data value from the call to 2nd language

Python with C

- Why C?
  - Very fast!
  - Python provides C API, hence gives great power
  - Other alternatives such as SWiG exist, but to understand them well you need to know Python/C API
- Why not C?
  - With great power comes great responsibility
  - Can be overkill
  - Easier to use solutions available with low overhead

Python with C: Approaches

- Python/C API
- ctypes module
- SWiG
- Others
    - Pyrex, Cython, elmer, etc . . .

Python/C API: Introduction

- Python's internal API
- Shipped with official distribution
- Used extensively in Python source code
- Most powerful way to work with Python interpretor
- Gives access to Python interpretor at various levels

Python/C API: Different World

- Python is clean, but not Python/C API
- Needs proper care and attention
- Not easy to get started

Python/C API: What we'll learn?

- Extending Python using C
- Embedding Python into C
- **Note** Python/C API can be used in C++ program without any problem

Python/C API: Basics

- Objects - Everything is PyObject
- Reference Counting
    - Py_INCREF()
    - Py_DECREF()
- Converting values
- Include <Python.h> to pull in the Python API

Python/C API: Embedding kickstart

```
#include <Python.h>

int main(int argc, char **argv)
{
    Py_Initialize();
    PyRun_SimpleString("print 'Hello World'");
    Py_Finalize();
    return 0;
}
```

- Compile using gcc

```
$ gcc -o embed-eg main.c -Wall \
          `python-config --cflags --libs`
```

Python C/API: Embedding Examples

- Calling Python functions from C
- Calling Python functions with arguments from C

Python C/API: Extending overview

- Write functions and types in C
- Describe them in array of structures
- Create Python constructs
- Compile using disutils (yay, Python!!)

Python C/API: Extending kickstart

- What we'll create?
  - A simple module written in C called spam
  - We'll interface C function <u>system()</u> in module <u>spam</u>
- How will we use it in Python?

```
import spam
spam.system("ls -l")
```

Python C/API: Extending kickstart

- **Step 1:** Implement/Interface the function to be exported in C

```c
static PyObject*
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;

    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

Python C/API: Extending kickstart

- **Step 2:** Create module method table

```
static PyMethodDef SpamMethods[] = {
    {
        "system",
        spam_system,
        METH_VARARGS,
        "Execute a shell command"
    },
    { NULL, NULL, 0, NULL }
};
```

Python C/API: Extending kickstart

- **Step 3:** Create module initialization function

```
PyMODINIT_FUNC initspam(void)
{
    PyObject *m;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;
}
```

Python C/API: Extending kickstart

- **Step 4:** Compile using distutils

```
from distutils.core import setup, Extension
spam_module = Extension('spam',
                        sources = ['spammodule.c'])

setup (name = 'spam',
        version = '1.0',
        description = 'Demo extenstion',
        ext_modules = [spam_module]
      )
```

```
$ python setup.py build
$ python setup.py install
```

Python C/API: Few points to ponder

- Functions returning PyObject*
    - **NULL** means **exception**
    - **non-NULL** a Python return value
        - including None!
- Functions returning int
    - **0**: Ok
    - -**1**: not Ok
    - (Unless its **true** or **false**)

Python C/API: Extending examples

- Implement a Python function using C
- Keyword arguments in these functions
- Build new data types

Python C/API: A stupid yet practical example

- Core application written using C++
  - C++ as we dont know how to write GTK
  - We'll use Qt for GUI
  - You dont have to worry about Qt and C++
- Embedded Python
  - Provide a Python interface to write Plug-ins
- What will our stupid application do?
  - You will provide a couple of basic functions to draw
  - You will feed Python script using those functions
  - The result will be shown on the Canvas, the drawing

ctypes module

- Foreign Function Library for Python (>= v2.5)
- Part of standard library
- Very easy to use
- Call functions in shared libraries or DLL's
- Provide C compatible data types
- Make new data types and arrays
    - Structures
    - Unions

ctypes module: When to use?

- When you just have to call C functions
- Good to port small libraries written in C
  - even larger, eg. pyglet, pyopengl
- You dont want your code to get ugly
- Time constraints

ctypes module: example

```
>>> from ctypes import *p(
>>>
>>> libc = cdll.LoadLibrary(‘‘libc.so.6’’)
>>>
>>> libc = cdll.msvcrt # for windows
>>> libc.printf(‘‘Hello World\n’’)
Hello World
11
>>>
>>> windll = cdll.kernel32 # windows
>>> systemdll = cdll.system32 # windows
```

ctypes module: Fundamental data types

- c_char, c_byte, c_wchar
- c_int, c_long, c_short, c_ulong
- c_float, c_double, c_longdouble
- c_void_p, c_char_p
- and so on!

ctypes module: Fundamental data types

```
>>> c_int()
c_long(0)
>>> c_char_p("Hello, World")
c_char_p('Hello, World')
>>> c_ushort(-3)
c_ushort(65533)
>>> i = c_int(42)
>>> print I
c_long(42)
>>> print i.value
42
>>> i.value = -99
>>> print i.value
-99
```

ctypes module: Calling functions

```
>>> printf = libc.printf
>>> printf("Hello, %s\n", "World!")
Hello, World!
14
```

ctypes module: Call function with custom data type

```
>>> class Bottles(object):
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf("%d bottles of beer\n", bottles)
42 bottles of beer
19
```

ctypes module: Specify argument type

```
>>> printf.argtypes = [c_char_p, c_char_p,
                                c_int, c_double]
>>> printf("String '%s', Int %d, Double %f\n",
                                  "Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>printf("%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ArgumentError: argument 2: exceptions.TypeError: wrong
>>> printf("%s %d %f\n", "X", 2, 3)
X 2 3.000000
13
```

ctypes module: Passing pointers

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print i.value, f.value, repr(s.value)
0 0.0 ''
>>> libc.sscanf("1 3.14 Hello", "%d %f %s",
...                 byref(i), byref(f), s)
3
>>> print i.value, f.value, repr(s.value)
1 3.1400001049 'Hello'
```

ctypes module: Structures

```
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print point.x, point.y
10 20
>>> point = POINT(y=5)
>>> print point.x, point.y
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: too many initializers
```

Python with C++

SWiG

Python with Java

- Jython
- Jython and JSR-223
- JPype

Jython

- Python implementation written purely in Java
- Runs on JVM
- Download from

  `http://www.jython.org`

Jython

- Dynamic compilation to Bytecode
- Very easy to use
- Use Java classes directly in Python code
- Embed Python in Java

Jython: kickstart

```
>>>
>>> from java.lang import *
>>> System.out.println("Hello World")
>>>
```

Jython: Embedding

- **Step 1:** Add jython.jar to CLASSPATH
- **Step 2:** Import packages and classes

```
import org.python.core.*
import org.python.util.PythonInterpretor
```

- **Step 3:**

```
PythonInterpreter interp = new PythonInterpreter();
interp.exec("import this");
interp.exec("print 'Hello World'");
```

Jython: Using Python class in Java

- **Step 1:** Create interface in Java

```
public interface Person {
    public String getName();
    public String getAge();
}
```

Jython: Using Python class in Java

- **Step 2:** Implement interface in Python

```
import PersonInterface

class Person(PersonInterface):
    def __init__(self, pname, page):
        self.pname = pname
        self.page = page

    def getName(self):
        return self.pname

    def getAge(self):
        return self.page
```

Jython: Using Python class in Java

- **Step 3:** Use Python class from Java

```
PythonInterpreter interp = new PythonInterpreter();

interp.exec("from Person import Person");
PyObject personClass = interpreter.get("Person");

PyObject personObject =
     personClass.__call__(new PyString(name),
                          new PyInteger(age));

PersonInterface person = (PersonInteface)
       personObject.__tojava__(PersonInterface.class);

System.out.println("Name: ", person.getName());
```

JPype

- For using Java from CPython
- Native Python interpreter communicates with JVM through JNI (Java Native Interface)
- Can access Java libraries, but need full package qualifiers

JPype: Basic Usage

```
>>> from jpype import *
>>>
# for linux
>>> startJVM("/opt/java/jre/lib/amd64/server/libjvm.so",
                                                    "-ea")
>>>
>>> java.lang.System.out.println("Hello World!")
>>>
>>> shutdownJVM()
```