

Using Python with Other Languages

Vishesh Yadav and Siddhant Sanyam

Guidelines

- Download tutorial content from <source>
- Linux preferred
- We'll cover many topics
- Tweet #pyconindia

Tutorial Content

- Why use multiple languages?
- How to do this in Python? Overview
- Python with C
- Python with C++
- SWiG
- Python with Java

What you should know?

- Some Python experience
- Experience with atleast one of these languages
 - C
 - C++
 - Java

Why use more than one language?

Why use more than one language?

- Performance boosts
- Better productivity
- Make you application scriptable

Examples!

- Mozilla (Thunderbird/Firefox)
- Vim, GNU Emacs, Kate
- GIMP, Inkscape
- Qt Frameworks and KDE
- Weechat, XChat, irssi

and many more . . .

Why Python for this?

“Python fits your brain”

–Bruce Eckel

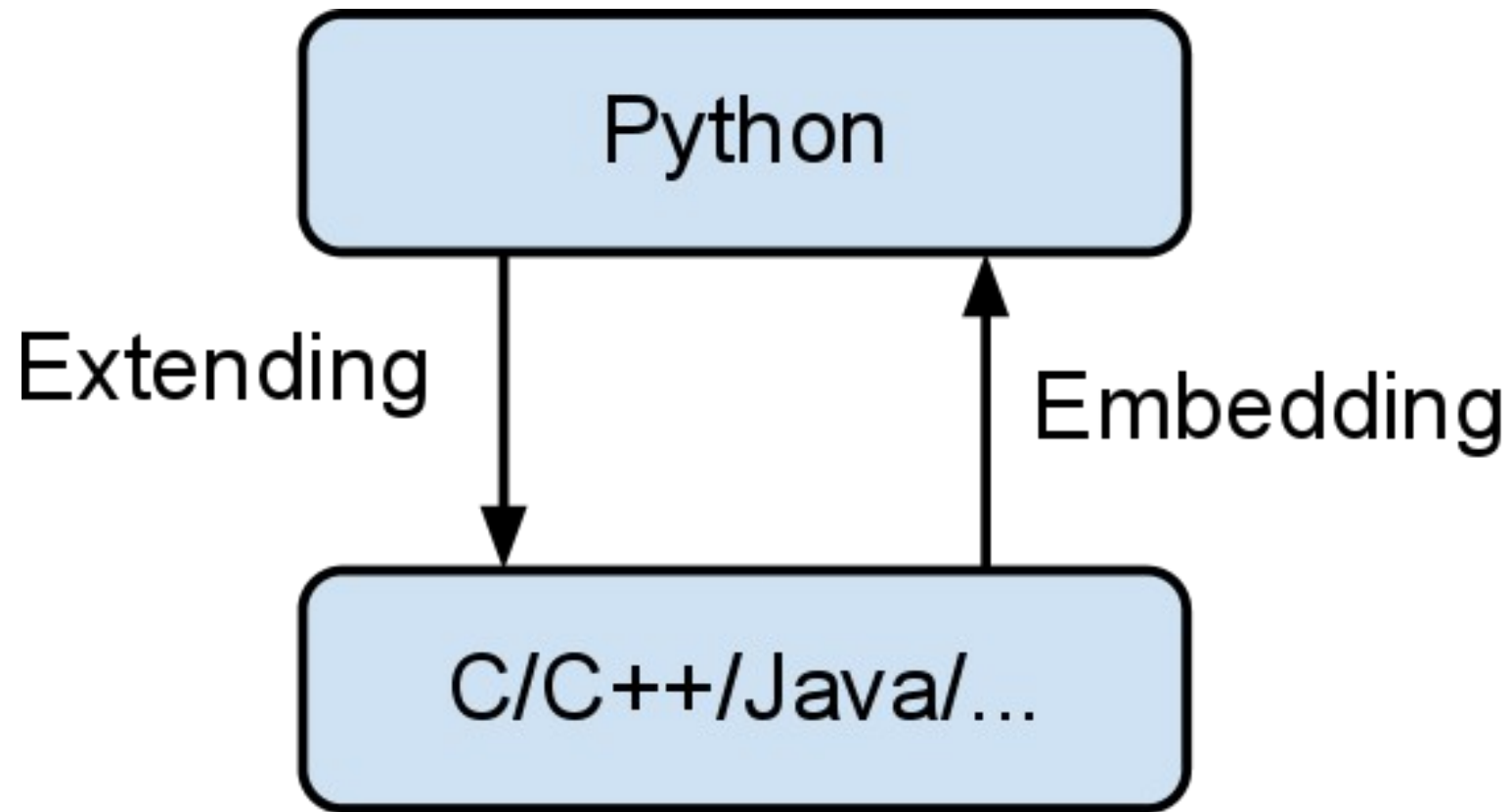
Why Python for this?

- Its Python!
- Easy to learn and get started
- Simple and robust syntax
- Targets all kinds of developers
 - Beginners to Advanced
 - Computer Programming for Everyone (CP4E)
- Powerful enough to do anything (blah almost!)
- Cross platform

How is it done?

Embedding and Extending

Embedding/Extending



Extending

- Convert data values from Python to 2nd language
- Perform a function call to 2nd language routine using converted value
- Convert data value from the call to Python

Embedding

- Convert data values from 2nd language to Python
- Perform a function call to Python interface routine using converted value
- Convert data value from the call to 2nd language

Python and C

Python and C: Approaches

- Python/C API
- Ctypes module
- SWiG
- Other approaches
 - Pyrex, Cython, elmer, etc...

Python/C API

- Python's internal API
- Part of cpython package
- Used to build Python
- Most powerful way to work with Python interpreter
- Gives access to Python interpreter at various levels

Python/C API – Different World

- Python is clean, but not its C API
- Have to be careful

Python/C API – What we'll learn?

- Extending Python using C (and C++)
- Embedding Python inside C (and C++)

Python/C API – What we'll learn?

- Extending Python using C (and C++)
- Embedding Python inside C (and C++)

Python/C API – Basics first!

- Objects – PyObject
- Reference counting
 - Py_INCREF()
 - Py_DECREF()
- Converting values
- Compiling

Python/C API – Embedding

```
#include <Python.h>
```

```
int main(int argc, char **argv)
```

```
{
```

```
    Py_Initialize();
```

```
    PyRun_SimpleString("print 'Hello World'");
```

```
    Py_Finalize();
```

```
    return 0;
```

```
}
```

Python/C API – Embedding

- Compile

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

Python/C API

Embedding Examples

- Calling Python functions from C
- Calling Python functions and pass arguments

Python/C API - Extending

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

Python/C API - Examples

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

A Stupid yet Practical Example Drawing Application

- Core application written using C++
 - C++ as we dont know how to write GTK
 - We'll use Qt for GUI (dont worry about this)
- Embedded Python
 - Provide a python interface to write Plug-ins

ctypes module

- Foreign function library for Python (\geq v2.5)
- Part of standard library
- Call functions in shared libraries or DLLs
- Provides C compatible data types
- Make new data types and arrays
 - Structures
 - Unions

ctypes: When to use?

- When you just have to call C functions
- Good to port small libraries written in C to Python
 - (even larger, eg. pygame, pyopengl)
- You don't want your code to get ugly
- Time constraints

ctypes: 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: 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: 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
```


c-types: calling functions

```
>>> printf = libc.printf
```

```
>>> printf("Hello, %s\n", "World!")
```

```
Hello, World!
```

14

Ctypes: 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: 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 type
>>> printf("%s %d %f\n", "X", 2, 3)
X 2 3.000000
13
>>>
```

ctypes: 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: 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 and C++

Python and C++: Approaches

- Python/C API
- Boost.Python
- SWiG

Python and Java

Python and Java: Approaches

- Jython
- Jython and JSR-223
- JPytype

Jython

- Python implementation written purely on Java
- Runs within JVM
- Download from:
 - <http://www.jython.org>

Jython

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

Jython - Introduction

```
>>>
```

```
>>> from java.lang import *
```

```
>>> System.out.println("Hello World")
```

```
>>>
```

Jython - Embedding

Step 1: Add jython.jar to CLASSPATH

Step 2: Import

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

Step 3:

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

Jython - Using Python classes in Java

Step 1: Create Interface in Java

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

Jython - Using Python classes 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 classes in Java

Step 3: Use the 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(PersonInterface)
    personObject.__tojava__(PersonInterface.class);

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


Jython - Using Python classes in Java

Step 3: Use the 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(PersonInterface)
    personObject.__tojava__(PersonInterface.class);

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

JPytype

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

JPytype - Basic Usage

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

Java Embedded Python (JEP)

- Uses native Python interpreter
- Provides access to Python modules
- <http://jepp.sourceforge.net/>