

Combining Python with C++: why?

Lecture 19

Extending Python

We want to write everything in Python: it is easier because it is "higher-level". But at many times we want the speed of C++, and we want to get close to the computer.

Then we need to combine the benefits of C++ and Python.

References:

- Python manual: <http://www.python.org/doc/1.5.2p2/>, especially sections on "Extending and Embedding".

PMOOP(0396A)

PMOOP(0396A)-19.1

We want to write most of our code in a scripting language like Python.

But **Python alone is not good enough** for all computing needs:

- Some **libraries** are only available in C++.
- You cannot touch the **internals** of the machines, e.g., deal with I/O ports, make system calls, etc., in Python.
- Some needed **data types** are unavailable in Python.
- Sometimes you need **efficiency** that is not provided by Python.

In these cases, we still want most of our code to be in a scripting language, but **we must write part of our code to produce machine code**.

That is, we want **both** the flexibility and efficiency of C++, and the ease of use and higher-level constructs in Python.

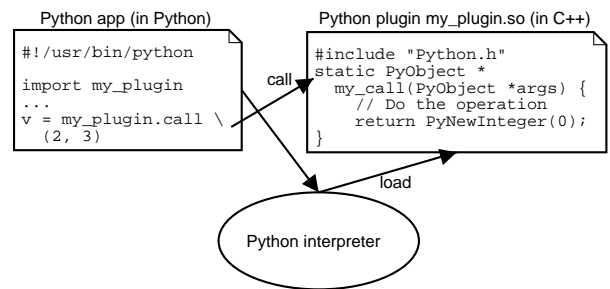
What can we do

There are two completely different approach to combine two languages.

| Extending Python | Embedding Python |
|---|--|
| Write the application as a Python script, which import some C++ plugins to perform functions unavailable in standard Python. | Write the application in C++, but runs a Python interpreter in it. The C++ program defines some functions that can be called from the Python interpreter. |
| Works best if Python is already "nearly good enough" for the application. The plugin provides the missing pieces. | Works best if Python is not good enough, but we want to allow users to configure or write scripts for our program using Python. |
| Examples: Tkinter, PIL, numeric library, etc. | Examples: Apache python-plugin, Gimp python-plugin, etc. |

PMOOP(0396A)-19.2

Extending Python: The concepts



When the **import** Python statement is executed, the Python interpreter tries to find a module or plugin with the appropriate name.

The **plugin** then registers some functions to Python so that the script can call them later.

PMOOP(0396A)-19.3

Writing a C++ function to be called from Python

As an example, we write a function that returns the *i*-th Fibonacci number. We should really do this in Python, so this is just an example.

```
#include "Python.h"
static PyObject *get_fib(PyObject *self, PyObject *args) {
    int arg, ret = 0;
    // Interesting place 1
    if (!PyArg_ParseTuple(args, "i", &arg))
        return 0; // Error! Propagate by returning NULL.
    if (arg >= 0) {
        int last = 1;
        for (int i = 0; i < arg; ++i)
            { int temp = ret; ret += last; last = temp; }
    }
    // Interesting place 2
    return PyInt_FromLong(ret);
}
```

PMOOP(0396A)-19.4

Reading the arguments

There are two interesting things in the code, which communicate values between our Python plugin to the Python script.

- A Python-callable function gets 2 or 3 Python objects as arguments.
- The first argument is useful only for class functions. Ignore it now.
- The second argument is a **tuple** containing some *position arguments*.
- The third argument is a **dictionary** containing *keyword arguments*.
- In most cases, we can **parse** the tuple by using *PyArg_ParseTuple*.
- The function is called with the **tuple**, a **format string** telling what type of arguments are expected, and then some pointers to variables that are to hold the parsed values.
- You can concatenate format chars, e.g. "ii" = two integers expected.

PMOOP(0396A)-19.5

Making out the result

The function must return a Python Object to the Python script. If we don't need to pass anything, at least we need to pass `Py_None`.

To create a value, we have two ways:

- Call `Py_BuildValue`. It is very similar to `Py_ParseTuple`: a **format string** tells what to make, followed by some arguments. E.g.:

```
Py_BuildValue("i", 20) // Create a Python integer of 20
```

Multiple format chars results in tuples. E.g., `Py_BuildValue("ii", 5, 10)`.

- Call a Python function that create a **specific type** of object. E.g.:

```
PyInt_FromLong(20) // The same
```

Usually we can use the first method, but if we need to build a sequence or dictionary of unknown number of elements, we need the second.

PMOOP(0396A)-19.6

Registering with Python interpreter

When Python loads a plugin, it calls a C function in the plugin called `initXXX`, where XXX is the plugin's name. If it's "fibonacci", we write

```
extern "C" void initfibonacci() {
    Py_InitModule("fibonacci", FibonacciMethods);
}
```

Here `FibonacciMethods` is an array that tells Python what methods we want to be available to Python scripts:

```
static PyMethodDef FibonacciMethods[] = { // Do this before initfibonacci()!
    {"get_fibonacci", get_fibonacci, METH_VARARGS }, // No keyword arguments
    { NULL, NULL } // No more Python functions
};
```

This tells that we want to expose the C++ `get_fibonacci` function to Python scripts, using the name "fibonacci.get_fibonacci" in Python.

PMOOP(0396A)-19.7

Setup.in and compiling

Now that we have the plugin source, we can build it. We can write a Makefile to do it, but actually Python provide an easier way.

From the Python installation directory `"/usr/lib/python1.5/conf"`, copy the file `"Makefile.pre.in"` to the current directory. Create a file called `"Setup.in"` like this:

```
*shared*
EC=fibonacci
CCC=g++
fibonacci fibonacci.cc
```

which is a partial Makefile. Then type `make -f Makefile.pre.in bootstrap` and then `make`.

This free us from remembering whether to use which compiler options like `-G`, `-Xlinker`, ... to build the shared object.

PMOOP(0396A)-19.8

Using the plugin

Using the plugin is easy. A simple script:

```
import fibonacci
print fibonacci.get_fibonacci(10)
```

If you pass it some wrong arguments (e.g., `fibonacci.get_fibonacci('a')`), you get error from the `PyArg_ParseTuple` function.

In general, returning a NULL value means that there is an error.

PMOOP(0396A)-19.9

Reference counting

Once you do a bit more than just creating new Python values and return, you start needing to take care about Python's garbage collector.

- Python use a **reference counting** mechanism to collect garbage.
- A new object (e.g., using `Py_BuildValue`) has a reference count of 1.
- We increment/decrement the count by `Py_INCREF/Py_DECREF`. **When the reference count of an object becomes 0, it is deleted.**
- When you don't need a Python object, call `Py_DECREF`, not delete.
- If you get an object without creating it, and then want to return it, increment the reference count.
- Example: to return `Py_None`, you need `Py_INCREF(Py_None)`; before return `Py_None`.

PMOOP(0396A)-19.10

Conclusion and exercise

- We can combine the benefits of C++ and Python by either extending Python by plugin or embedding Python in a C++ application.
- To create a plugin, we need to define a C++ function taking python objects, define an array of exposed functions, and make an init function.
- Python provide an easy way to build the plugin using `make`.
- We load a plugin from Python just like we import a Python module.
- Python uses reference counting to clean-up objects that are no longer available.

Exercise: The source of the lecture contains a directory `fibonacci`, which in turn contains the source code in this lecture `fibonacci.cc` and `Setup.in`. Try to build the plugin and use it.

PMOOP(0396A)-19.11