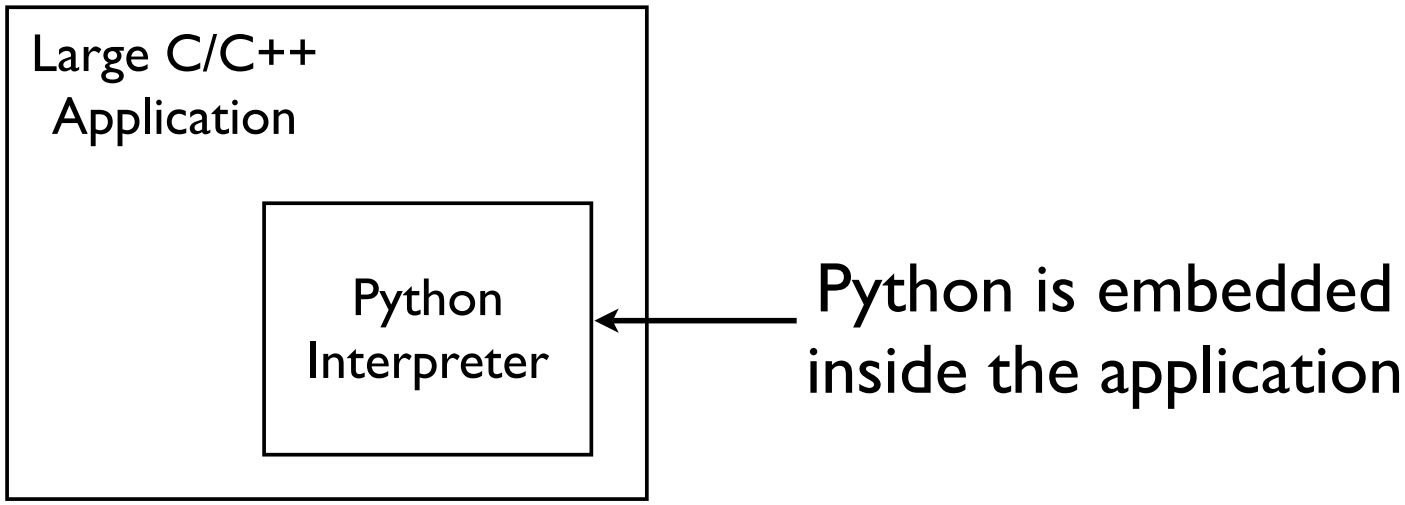
Embedding Python

Overview

- Embedding the Python interpreter into other applications
- Calling Python from C/C++
- A slightly different problem than extension building

Problem



- Typical scenario: Python is being used as a scripting environment for a special purpose application (e.g., game, scientific software, simulation, etc.)
- Python functionality as a C/C++ library

Compiling and Linking

- Python interpreter is packaged as a C library
- To embed, you merely link with the library

```
% cc foo.c bar.c -lpython2.5
```

- Finding the library might be a bit of work
- Usually found in the Python installation someplace

```
$(sys.exec_prefix)/lib/python2.5/config/libpython2.5.a
```

Compiling and Linking

- The standard disclaimers
  - Linking with Python depends on system
  - Depends on C/C++ compiler used
  - May depend on other libraries
  - May not work at all (at least at first)
- Your mileage may vary

Initializing Python

- With embedding, you are in charge
- Must explicitly initialize the interpreter

```
#include "Python.h"
int main(int argc, char **argv) {
    Py_SetProgramName(argv[0]);
    Py_Initialize();
    PySys_SetArgv(argc,argv);
    ...
    ... rest of program
    ...
}
```

Initializing Python

- With embedding, you are in charge
- Must explicitly initialize the interpreter

```
#include "Python.h"
int main(int argc, char **argv) {
    Py_SetProgramName(argv[0]);
    Py_Initialize();
    PySys_SetArgv(argc,argv);
    ...
    ... rest of program
    ...
}
```

Sets program name found in sys.argv[0]

Initializing Python

- With embedding, you are in charge
- Must explicitly initialize the interpreter

```
#include "Python.h"
int main(int argc, char **argv) {
    Py_SetProgramName(ar
    Py_Initialize();
    PySys_SetArgv(argc,a
    ...
    ... rest of program
    ...
}
```

Initializes the interpreter

Initializing Python

- With embedding, you are in charge
- Must explicitly initialize the interpreter

```
#include "Python.h"
int main(int argc, char **argv) {
    Py_SetProgramName(argv[0]);
    Py_Initialize();
    PySys_SetArgv(argc,argv);
    ...
    ... rest of program
    ...
}
```

Sets arguments in sys.argv

Finalizing Python

- Must finalize Python on program exit

```
#include "Python.h"
int main(int argc, char **argv) {
    Py_SetProgramName(argv[0]);
    Py_Initialize();
    PySys_SetArgv(argc,argv);
    ...
    ... rest of program
    ...
    /* Program about to exit */
    Py_Finalize();
}
```

Running Python Code

- Running a simple string

```
PyRun_SimpleString("print 'hello world'");
```

- Running a simple file

```
FILE *f = open("someprogram.py","r");
PyRun_SimpleFile(f,"someprogram.py");
```

- Running the interactive eval loop

```
PyRun_InteractiveLoop(stdin,"<stdin>");
```

- All of these execute code in __main__

Extracting Data

- Importing a module (from C)

```
PyObject *mod = PyImport_ImportModule("modname");
```

- Getting a symbol from a module

```
PyObject *obj = PyObject_GetAttrString(mod,"name");
```

- Printing a Python object (for debugging)

```
PyObject_Print(obj,file,0);
```

Extracting Data

- Example: Print the value of sys.argv

```
PyObject *sysmod;
PyObject *argv;

sysmod = PyImport_ImportModule("sys");
argv = PyObject_GetAttrString(mod,"argv");

PyObject_Print(argv,stdout,0);
```

Setting Values

- To set a value in a module

```
PyObject *obj;
obj = Py_BuildValue("i",37);

PyObject_SetAttrString(mod,"name",obj);
Py_DECREF(obj);
```

- Example: Set a variable in sys module

```
PyObject *obj;
PyObject *mod;
mod = PyImport_ImportModule("sys");
obj = Py_BuildValue("[sss]","foo","bar",spam");

PyObject_SetAttrString(mod,"argv",obj);
Py_DECREF(obj);
```

Calling a Function

- Getting a function object

```
PyObject *mod;
PyObject *func;

mod = PyImport_ImportModule("math");
func = PyObject_GetAttrString(mod,"pow");
```

- Making a function call

```
PyObject *args, *pyresult;
double result;

args = Py_BuildValue("(dd)",2.0,3.0);
pyresult = PyEval_CallObject(func,args);
result = PyFloat_AsDouble(pyresult);
Py_DECREF(args);
Py_DECREF(result);
```

Python as a C Library

- Problem: Write a C function that matches a C string against a regular expression using Python's regex engine
- Hide the fact that Python is involved

```
/* Return 1 if the string str matches pattern.
   Return 0, otherwise */
int match(const char *pattern, const char *str) {
  ...
}
```

Example: Pattern Matching

- Solution

```
int match(const char *pattern, const char *str) {
  static PyObject *remod = 0;
  static PyObject *match_func = 0;
  PyObject *args, *result;
  int       matched;

  if (!remod) {
     remod = PyImport_ImportModule("re");
     match_func = PyObject_GetAttrString(remod,"match");
  }
  args = Py_BuildValue("(ss)",pattern,str);
  result = PyEval_CallObject(match_func,args);
  matched = PyObject_IsTrue(result) ? 1 : 0;
  Py_DECREF(args);
  Py_DECREF(result);
  return matched;
}
```

Example: Pattern Matching

- Sample use:

```
while (1) {
   char buffer[256];
   printf("Enter a number:");
   fgets(buffer,255,stdin);
   if (!match("\\d+$",buffer)) {
       printf("That's not a number\n");
   } else {
       printf("Congratulations!\n");
       break;
   }
  }
```

- Note: Python tucked away behind scenes

Creating Objects

- Objects are created by function calls

```
PyObject *mod;
PyObject *func;
PyObject *args;
PyObject *obj;

mod = PyImport_ImportModule("StringIO");
cls = PyObject_GetAttrString(mod,"StringIO");
args = Py_BuildValue("()");
obj = PyEval_CallObject(cls,args);
```

- Same as this Python code

```
>>> import StringIO
>>> obj = StringIO.StringIO()
```

Calling a Method

- Getting a method on an object and calling it

```
PyObject *meth;
PyObject *args;
PyObject *result;

meth = PyObject_GetAttrString(obj,"write");
args = Py_BuildValue("(s)","Hello World\n");
result = PyEval_CallObject(meth,args);
```

- Same as this Python code

```
>>> r = obj.write("Hello World\n")
>>>
```

Example: C++ Proxy

- Problem: Create a C++ class that acts as a proxy for a Python object

```
// C++ class
class StringIO {
public:
    StringIO();
   ~StringIO();
    string read(int maxbytes);
    void   write(string &s);
    string getvalue();
}
```

- Will look like a C++ class, but it's really just a Python object in disguise

Example: C++ Proxy

- Constructor

```
// C++ class
class StringIO {
    PyObject *self;
    static PyObject *cls;
    static PyObject *read_mth,*write_mth,*getvalue_mth;
public:
    StringIO() {
       if (!cls) {
          PyObject *mod = PyImport_ImportModule("StringIO");
          cls = PyObject_GetAttrString(mod,"StringIO");
          read_mth = PyObject_GetAttrString(cls,"read");
          write_mth = PyObject_GetAttrString(cls,"write");
          getvalue_mth = PyObject_GetAttrString(cls,"getvalue");
       }
       PyObject *args = Py_BuildValue("()");
       self = PyEval_CallObject(cls,args);
       Py_DECREF(args);
    }
```

Example: C++ Proxy

- Set-up

```
// C++ class
class StringIO {
    PyObject *self;
    static PyObject *cls;
    static PyObject *read_mth,*write_
public:
    StringIO() {
       if (!cls) {
          PyObject *mod = PyImport_ImportModule("StringIO");
          cls = PyObject_GetAttrString(mod,"StringIO");
          read_mth = PyObject_GetAttrString(cls,"read");
          write_mth = PyObject_GetAttrString(cls,"write");
          getvalue_mth = PyObject_GetAttrString(cls,"getvalue");
       }
       PyObject *args = Py_BuildValue("()");
       self = PyEval_CallObject(cls,args);
       Py_DECREF(args);
    }
```

This one-time code gets a reference to the Python class and unbound methods associated with the class

Example: C++ Proxy

- Instance creation

```
// C++ class
class StringIO {
    PyObject *self;
    static PyObject *cls;
    static PyObject *read_mth,*write_mth,*getvalue_mth;
public:
    StringIO() {
        if (!cls) {
            PyObject *mod = PyImport_Import
            cls = PyObject_GetAttrString(mo
            read_mth = PyObject_GetAttrStri
            write_mth = PyObject_GetAttrStr
            getvalue_mth = PyObject_GetAttrString(cls,"getvalue");
        }
        PyObject *args = Py_BuildValue("()");
        self = PyEval_CallObject(cls,args);
        Py_DECREF(args);
    }
```

Here, we are creating an instance of a Python class and saving a reference

Example: C++ Proxy

- Destructor

```
// C++ class
class StringIO {
    PyObject *self;
    static PyObject *cls;
    static PyObject *read_mth,*write_mth,*getvalue_mth;
public:
    ~StringIO() {
       if (Py_IsInitialized()) {
          Py_DECREF(self);
       }
     }
```

- Simply decrement the ref-count of self
- Caveat: Only if Python is still alive

Example: C++ Proxy

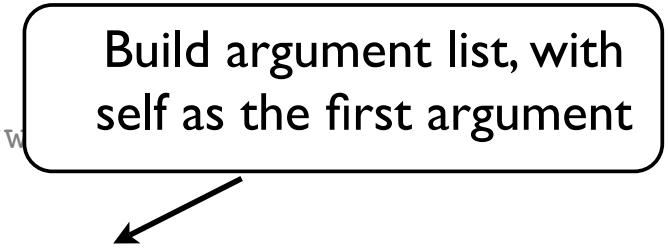
- Implementing a proxy method

```
// C++ class
class StringIO {
    PyObject *self;
    static PyObject *cls;
    static PyObject *read_mth,*write_mth,*getvalue_mth;
public:
    void write(string s) {
        PyObject *args = Py_BuildValue("(Os#)",self,
                                       s.c_str(),s.length());
        PyObject *r = PyEval_CallObject(write_mth,args);
        Py_DECREF(args);
        Py_DECREF(r);
     }
```

Example: C++ Proxy

- Implementing a proxy method

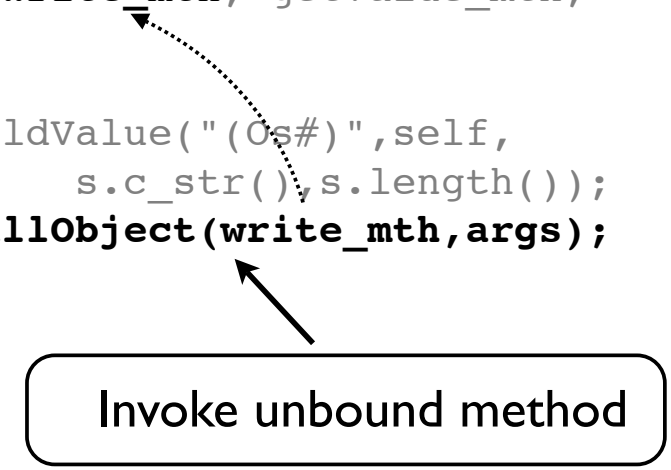
```
// C++ class
class StringIO {
    PyObject *self;
    static PyObject *cls;
    static PyObject *read_mth,*w
public:
    void write(string s) {
        PyObject *args = Py_BuildValue("(Os#)",self,
                                       s.c_str(),s.length());
        PyObject *r = PyEval_CallObject(write_mth,args);
        Py_DECREF(args);
        Py_DECREF(r);
     }
```



Example: C++ Proxy

- Implementing a proxy method

```
// C++ class
class StringIO {
    PyObject *self;
    static PyObject *cls;
    static PyObject *read_mth,*write_mth,*getvalue_mth;
public:
    void write(string s) {
        PyObject *args = Py_BuildValue("(Os#)",self,
                                       s.c_str(),s.length());
        PyObject *r = PyEval_CallObject(write_mth,args);
        Py_DECREF(args);
        Py_DECREF(r);
     }
```



Example: C++ Proxy

- Implementing another proxy method

```
// C++ class
class StringIO {
    PyObject *self;
    static PyObject *cls;
    static PyObject *read_mth,*write_mth,*getvalue_mth;
public:
    string getvalue() {
        char *s;
        int   len;
        PyObject *args = Py_BuildValue("(O)",self);
        PyObject *r = PyEval_CallObject(getvalue_mth,args);
        Py_DECREF(args);
        PyString_AsStringAndSize(r,&s,&len);
        string result = string(s,len);
        Py_DECREF(r);
        return result;
     }
```

Example: C++ Proxy

- Using the proxy class

```
int main(int argc, char **argv) {
    Py_Initialize();

    StringIO s;
    s.write(string("Hello World\n"));
    s.write(string("This is a test\n"));
    cout << s.getvalue();

    Py_Finalize();
}
```

This is using Python, but it looks like clean C++ code

- Running it:

```
% a.out
Hello World
This is a test
%
```

Advanced Example

- Directors
- Consider a C++ class with virtual methods

```
class EventHandler {
public:
    virtual void handle_keydown(int ch);
    virtual void handle_keyup(int ch);
    ...
};
```

- Normal use: Someone inherits from this class, and implements methods to handle events (e.g., part of GUI).

Advanced Example

- Direct methods to a Python object

```
class PyEventHandler : public EventHandler {
    PyObject *self;
    PyObject *mth_keydown, *mth_keyup;
public:
    PyEventHandler(PyObject *self) : self(self) {
        ...
    }
    virtual void handle_keydown(int ch) {
        ...
    }
    virtual void handle_keyup(int ch) {
        ...
    }
};
```

- This will look a lot like last example

Advanced Example

- Setup - Place wrapper about Python object

```
class PyEventHandler : public EventHandler {
    PyObject *self;
    PyObject *mth_keydown, *mth_keyup;
public:
    PyEventHandler(PyObject *self) : self(self) {
        mth_keydown = PyObject_GetAttrString(self,
                                "handle_keydown");
        mth_keyup = PyObject_GetAttrString(self,
                                "handle_keyup");
        ...
    }
    ...
};
```

Advanced Example

- Implement virtual methods that call Python

```
class PyEventHandler : public EventHandler {
    PyObject *self;
    PyObject *mth_keydown, *mth_keyup;
public:
    virtual void handle_keydown(int ch) {
        PyObject *args = Py_BuildValue("(i)",ch);
        PyObject *r = PyEval_CallObject(mth_keydown,args);
        Py_DECREF(args);
        Py_DECREF(r);
    }
    ...
};
```

Advanced Example

- If set up right, virtual functions in C++ will magically redirect to Python
- C++ won't know or care

```
void process_events(EventHandler *h) {
    ...
    h->handle_keydown(ch);
    ...
}

PyObject *pyobj = ... some object ...;
PyEventHandler *pyh = new PyEventHandler(pyobj);
process_events(pyh);
```

Advanced Example

- If mixing C++/Python, may need to have ways of registering Python objects

```
PyEventHandler *register_pyhandler(PyObject *obj) {
    PyEventHandler *pye = new PyEventHandler(obj);
    register_handler(pye);
}
```

- This, in turn, may be exposed to Python as a wrapper function/extension module

```
class MyHandler(object):
    def __init__(self):
        register_pyhandler(self)
        ...
    def handler_keydown(self,ch):
        ...
```

Summary

- Have seen basics of embedding
- Adding Python interpreter to an application
- Starting the interpreter, importing modules
- Calling Python functions
- Creating objects and invoking methods
- Implementing libraries and C++ objects using Python behind the scenes

Where to go from here?

- More details are found in the "Embedding and Extending Reference"
- Techniques described are used if Python added to other frameworks or other programming languages
- Various ways to package (e.g., "freezing").
- Devil is in the details