3rd Python Workshop Tutorial

Extending and Embedding Python

Day Three – Embedding Python

Guido van Rossum

guido@CNRI.Reston.Va.US
http://www.python.org

Day Three – Overview

- Embedding Python in applications
- Invoking Python from C
- Loose ends, further reading

Embedding Python In Applications

- Write your own main()
- Initialize Python before using it
- Provide your own versions of certain routines
- Provide an extension to access your application
- Provide your own **config.c** file
- Link Python as a library

Minimal `main()` Program

```c
#include "Python.h"

main(int argc, char *argv)
{
    Py_Initialize(); /* initialize interpreter */

    /* Optional: */
    PySys_SetArgv(argc, argv); /* sys.argv */

    /* Invoke interpreter, e.g.: */
    PyRun_SimpleString("...python code...\n");
}
```

Python Initialization

`void Py_Initialize();`

- There isn't really much that needs to be initialized:
  - the import system (initializes sys.modules)
  - module __builtin__ (built-in functions, exceptions)
  - module sys (used by other parts)
  - module __main__ (for completeness' sake)
  - Unix signals
- All other modules are initialized when imported
- All object types are self-initializing
  - most are actually statically initialized

Setting `sys.argv`

`void PySys_SetArgv(int argc, char *argv);`

- Use this to set sys.argv
  - If not called, sys.argv does not exist
- Only Python "main programs" should need sys.argv
  - Scripts run from embedding application shouldn't need it
  - However, some library modules peek in it (e.g. Tkinter)
    - » so you may have to define it anyway
  - Doesn't have to be derived from real main()'s argc/argv

Override Some Routines

You can (but need not) override some routines:

- char *getpythonpath()
  - return default path for your application
    - » it's up to you to use $PYTHONPATH

- char *getprogramname()
  - full pathname of executable
    - » used for loading shared libraries (not for sys.argv)

Provide An App-specific Extension

- Presumably, the Python code you run will need to access functionality specific to your application
  - otherwise you could just as well run it as a subprocess

Provide Your Own `config.c`

- You may want to restrict or extend the set of modules that are available to Python code
- Example (simplified):

```
extern void initmath();
extern void initYourAppName();
struct {
    char *name;
    void (*initfunc)();
} inittab[] = {
    {"math", initmath},
    {"YourAppName", initYourAppName"},
    {NULL, NULL}
};
```

Link Python As A Library

- Link with the 4 Python libraries in this order:
  - libModules.a
  - libPython.a
  - libObjects.a
  - libParser.a
- Also link with any other libraries needed by:
  - Python
    - » normally none, but platform and configuration specific
  - any extensions you link with
  - see Python's Makefile, or Setup

Invoking Python From C

- What do you want?
  - run a script for its side effects
  - run a script for its output *(capturing the output)*
  - evaluate an expression for its value
  - call a function written in Python *(special case)*
- Where does the code come from?
  - read from a file
  - read from a string buffer (or literal)
  - read from a tty device
- In which namespace should the code run?
  - in module **`__main__`**
  - in a special dictionary

Run A Script For Its Side Effects

To run code in module **`__main__`**:

- From file or string buffer:
  - int PyRun_SimpleFile(FILE *, char *); /* script */
  - int PyRun_SimpleString (char *); /* command */
- From file, which may be a tty device:
  - int PyRun_AnyFile(FILE *, char *);/* script or interactive */
  - int PyRun_InteractiveOne(FILE *, char *); /* one cmd */
  - int PyRun_InteractiveLoop(FILE *, char *); /* until EOF */

the (second char * argument is the filename, for use in error messages)

To Capture A Script's Output

(Without running it in a subprocess via a pipe)

- Set sys.stdout to something that captures the output
  - Could be a Python file object
  - Could be anything else with a "write()" method
    - For instance StringIO.StringIO() instance
    - Or something that appends the string to a text widget

To Run In Another Namespace

Interfaces:

- PyObject *PyRun_File(FILE *file, char *filename, int start,
  PyObject *globals, PyObject *locals);
- PyObject *PyRun_String(char *buffer, int start,
  PyObject *globals, PyObject *locals);

- Create a dictionary object to serve as namespace
  - PyObject *ns = PyDict_New();
  - pass it as global as well as for local namespace
- Include "graminit.h"
  - use **`file_input`** as start symbol
- N.B. Return value is *owned* ref to None
  - or NULL if an exception occurred

To Evaluate An Expression

- Use PyRun_File() or PyRun_String() with start symbol: **`eval_input`**
- Return value is the expression's value (owned)
  - or NULL (you know when)

To Call A Function

- PyObject *PyEval_CallObject(PyObject *, PyObject *);
  - takes function object, argument tuple
  - always runs in the function's own module
  - returns owned ref to function return value, or NULL
- PyObject *PyEval_CallObjectWithKeywords(
  PyObject *, PyObject *, PyObject *);
  - third argument is dictionary containing keyword arguments

How To Access Variables

- Use your own namespace dictionary
- Use PyDict_* interface to access variables

- If you *have* to use __main__, extract its dictionary:

```
PyObject *modules = PySys_GetObject ("modules");
PyObject *main = PyDict_GetItemString(modules,
                                      "__main__")
PyObject *dict = PyModule_GetDict( main);
```

How To Handle Exceptions

- PyErr_Print()
  - picks up traceback from current exception
  - prints traceback on **sys.stderr**
  - clears exception condition when done
- PyErr_Fetch(&exception, &value, &traceback)
  - roll your own error reporting mechanism
    - » see print_error() in pythonrun.c for example
  - you *own* the three object references
  - value and traceback may be NULL
  - exception is NULL if no exception occurred

Loose Ends

Some things that didn't get enough attention earlier:

- The most important reference count anomalies
- Useful include files

The Most Important Reference Count Anomalies

- Borrowed references returned:
  - PyTuple_GetItem()
  - PyList_GetItem()
  - PyDict_GetItem (), PyDict_GetItemString ()
    - » also doesn't set exception when NULL returned
  - PyModule_GetDict()
  - PySys_GetObject()
- References "stolen":
  - PyTuple_SetItem()
  - PyList_SetItem()
  - ***But N O T*** PyDict_SetItemString() – it is well-behaved

Useful Include Files

- rename2.h (nearly all names in the Python API)
- abstract.h (abstract object interface)
- object.h, objimpl.h (object framework)
- listobject.h, tupleobject.h, mappingobject.h
- pythonrun.h, ceval.h (calling the interpreter)

Any Questions?

- This is your last chance!

- ...Or send email to python-help@python.org