3rd Python Workshop Tutorial

Extending and Embedding Python

Day One – Writing Extensions

Guido van Rossum

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

Three Day Tutorial Overview

Day One

- Extending vs. Embedding
- Writing extension modules in C
- Compile/link time issues

Day Two

- Defining new Python objects in C
- Generating extensions (semi-)automatically
  - Modulator; BGEN

Day Three

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

Day One – Overview

- Extending versus Embedding
- Writing Extension Modules in C
  - Lots of Details on what parts to write
  - Exceptions
  - Reference Counting
  - Using Python Objects from C
- Compile Time Issues
  - Compiling extensions
  - Dynamic loading of extensions
  - Using other languages than C
  - Non-Unix platforms

Extending versus Embedding

- Extending Python:
  - write a new "built-in" module in C/C++
  - used by ordinary Python code through "import", for either:
    - » performance
    - » use of libraries only accessible from C/C++
- Embedding Python:
  - incorporate Python in another application
  - usually comes with application-specific extension!
  - Python interpreter invoked by application for subtasks, e.g.:
    - » running scripts written by end users
    - » localization, customization

Writing Extension Modules in C

- How extensions are used
- How they are initialized
- How methods are defined
  - method implementation functions
  - method table and method templates
  - parsing arguments
  - return value
- Raising and using exceptions
- Reference counting
- Using Python objects from C

How Extensions Are Used

- Just like other modules. For example:

```
import posix
...
posix.unlink("some_file")
...
file = posix.popen("ls -l", "r")
lines = file.readlines()
file.close()
```

- The only way to tell is by looking at the semantics...

What Extensions Can Do That Ordinary Modules Can't

- Define new *opaque, built-in* types
  - representation using C data types
- Call any function that's in a library, e.g.:
  - low-level I/O functions
  - platform-specific libraries
  - numerical routines
- Execute bit/byte level operations *efficiently*
  - especially loops over large amounts of data
- Make horrible mistakes

What Python Modules Can Do That Extensions Can't

- Define classes usable as base class for subclassing
- (Easily) import and access other modules

- Be careless with error checking and allocation

...But Extensions can run scripts from strings...

What Happens During Import

- The first time a module is imported, it is *loaded*
  - On following imports, the already loaded module is used
- To load an extension, its *init function* is called, which:
  - creates an empty module object
  - installs it in sys.modules
  - fills the module object's dictionary with objects
- This is accomplished by calling Py_InitModule():
  - Py_InitModule("posix", posixmethods);
  - posixmethods is an array of *method object templates* (see later)
  - return value is the module object
    - » used for adding other objects

Module Init Function (Basic)

- Must be called **`init`*`modulename`***:

```
#include "Python.h"

static PyMethodDef posixmethods[] = {...};

void initposix() {
    Py_InitModule("posix", posixmethods);
}
```

Module Init Function (2)

```c
#include "Python.h"

static PyObject *error;

static PyMethodDef posixmethods[] = {...};

void initposix() {
    PyObject *m, *d;
    m = Py_InitModule("posix", posixmethods);
    d = PyModule_GetDict(m);
    error = PyString_FromString("posix.error");
    PyDict_SetItemString (d, "error", error);
}
```

Method Implementation Functions

- Really "function implementation"; referred to as methods for reasons that will become clear later

- Declared static, e.g.

```
static PyObject *
posix_unlink(PyObject *self, PyObject *args)
{
    ....
}
```

- We'll fill in the blanks later

Method Table

- Contains list of method templates, e.g.:

```
static posixmethods[] = {
  ...
  {"unlink", posix_unlink, METH_VARARGS, NULL},
  ...
  {NULL} /* Sentinel */
};
```

- Passed to Py_InitModule() by module init function
- Must have static lifetime

Method Templates

- Describes one method, e.g.:

  ```
  {"unlink", posix_unlink, METH_VARARGS, NULL},
  ```

- Components:
  - method name
  - C function
  - flags (METH_VARARGS, METH_KEYWORDS)
  - optional doc string
- Must have static lifetime
  - Incorporated by reference in method object

Method Implementation (Revisited)

```
static PyObject *
posix_unlink(PyObject *self, PyObject *args) {
  char *file;
  if (!PyArg_ParseTuple(args, "s", &file))
      return NULL;
  unlink(file);  /* error check omitted */
  Py_INCREF(Py_None);
  return Py_None;
}
```

- Return NULL if failure (with exception set)
- Return a "new" object if success

PyArg_ParseTuple()

- Parse "varargs" tuple according to template string
- return 1 if success; 0 if failure (and sets exception)
- store C values through pointers (a la sscanf())
- Sample format codes (many more exist):
  - **`s`** = string, **`s#`** = string plus size (2 C variables)
  - **`i`** = int, **`l`** = long int
  - **`O`** = arbitrary object
  - **`|`** = following arguments are optional, e.g. **`"s|si"`**
    - » C variables must be initialized to default values
- Must set **`METH_VARARGS`** flag in method template!!!!

Return Value

- Must be one of:
  - a new reference to an existing object, e.g.

    ```
    Py_INCREF(Py_None);
    return Py_None;
    ```
  - a new object, e.g.

    ```
    return Py_BuildValue("s", "hello world");
    ```
  - NULL (to raise an exception, see later)
- Use **`Py_BuildValue()`** to construct Python values

Py_BuildValue()

- Roughly the inverse of PyArg_ParseTuple()

- Create new Python object from template string, e.g.

  `Py_BuildValue("s", "spam")`
  - returns string object: **`"spam"`**

  `Py_BuildValue("iis", 12, 34, "eggs")`
  - returns 3-tuple: **`(12, 34, "eggs")`**

Raising Exceptions

- Common usage:

```
PyErr_SetString(exception, "message");
return NULL;
```

- Common exceptions:

  `PyExc_TypeError, PyExc_IOError`, etc.

- Modules can define their own exception objects
  - must be string objects
    (in the future, exceptions will become classes)
- To specify an arbitrary object as "message":

```
PyErr_SetValue(exception, object);
```

Using Exceptions

- If a function you call raises an exception, *don't set the exception again yourself!*
  - just **`return NULL;`**
- To test for an exception, use **`PyErr_Occurred()`**
  - usually, Python run-time functions return NULL or –1
- To clear an exception, use **`PyErr_Clear()`**
- To catch an exception, use **`PyErr_Fetch(&exc, &msg, &traceback)`**
  - this clears the exception condition
  - you must decref exc, msg, traceback when you're done!

Reference Counting

- Py_INCREF(x)
  - increment reference count of x; x must not be NULL
- Py_XINCREF(x)
  - increment reference count of x if it is not NULL
- Py_DECREF(x)
  - decrement reference count of x; x must not be NULL
- Py_XDECREF(x)
  - decrement reference count of x if it is not NULL

Reference Counting (2)

- When **`Py_DECREF(x)`** sets reference count to zero:
  - x is deallocated by calling the dealloc() function of its type
    - » (see later)

- Regardless of the resulting reference count,
  variable x should not be used following **`Py_DECREF(x)`**

Reference Counting (3)

- You either "borrow" or "own" references to objects
- Borrowed references:
  - become invalid when current call returns
  - should never be argument to **`Py_DECREF()`**
  - can be changed to owned by **`Py_INCREF()`**
- Owned references:
  - must eventually be disposed of by **`Py_DECREF()`**
  - are "cloned" by **`Py_INCREF()`** (-> two owned references)
- Rule of thumb:
  - arguments are borrowed by callee
  - return values are owned by caller

Using Python Objects From C

- Old interface:
  - separate functions per object type, e.g.:
    - » **`PyList_GetItem(list, index)`**
    - » **`PyTuple_GetItem(tuple, index)`**
  - only documented in source files
- New "abstract" interface:
  - matches polymorphic Python language operations, e.g.:
    - » **`PySequence_GetItem(sequence, index)`**
  - documented in **`"abstract.h"`** header file
  - replaces old interface except for certain odd cases
    - » e.g. constructing tuples/lists efficiently

Example: Sum Item Lengths

- Python equivalent:

```
def count(seq):
    sum = 0
    for item in seq:
        sum = sum + len(item)
    return sum
```

Sum Item Lengths: C Code

```
static Count(PyObject *self, PyObject *args) {
    PyObject *seq, *item; int i, n, len; int sum = 0;
    if (!PyArg_ParseTuple("O", &seq)) return NULL;
    if ((n = PyObject_Length(seq)) < 0) return NULL;
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(seq, i);
        if (item == NULL) return NULL;
        len = PySequence_GetLenth(item);
        Py_DECREF(item);
        if (len < 0) return NULL;
        sum = sum + len;
    }
    return Py_BuildValue("i", sum);
}
```

Compile/Link Time Issues

- Compiling extensions
- Dynamic loading of extensions
- Using other languages than C
- Non-Unix platforms

Compiling Extension Modules

- Simplest solution: in the Python source tree
  - drop in **`Modules`** directory
  - add a line to the **`Setup`** file, e.g.:

```
posix posixmodule.c -I/usr/include -L/lib -lc
```

- Independent: in your own directory
  - Copy **`Makefile.pre.in`** from **`Extensions`** directory
  - add the same line to the **`Setup`** file
  - make sure first to run

    ```
    make inclinstall; make libainstall
    ```

    in the Python source tree (then you can throw it away)

Dynamic Loading of Extensions

- To build a shared library:
  - insert your module after **`*noconfig*`** in the **`Setup`** file
- Unfortunately this is extremely platform specific:
  - different compiler and loader options in all phases
    - » sometimes *all other code* must be compiled specially
  - configure script switches on platform name
    - » additions welcome
  - sometimes manual intervention is easiest

Using Other Languages

- Using C++ is easy: all Python headers start with

```
#ifdef __cplusplus
extern "C" {
#endif
```
(and end with similar code)

- Must edit the Makefile to compile with C++

- To use code written in other languages
(e.g. numerical Fortran code)
if you can call those languages from C/C++
  - you have to write a C++ wrapper

Non-Unix Platforms

- Usually have integrated development environment
- Adding your extension to the project is not enough
  - you also need to edit **`config.c`**
    to add the new module to the table of modules
    - » on Unix this is done automatically by **`makesetup`**
- Building a shared library (DLL) is also possible
  - except on 68K Mac (CFM68K support isn't stable yet)
  - consult platform-specific documentation

Any Questions?