Extending Python with C

Alex Martelli
AB Strakt

This Tutorial's Audience

- You have a good base knowledge of C (ANSI/ISO) and Python (any version)
- You want to get started writing Python extensions in C
- You want some tips on extension-writing strategies and tactics
- You know Python docs live at:

http://www.python.org/doc/current/

This Tutorial's Style

- Meant to be *interactive*
- I need feedback on your knowledge and on how well you're following
- You need to ask questions/participate (otherwise, just read *Py* zine, #1-2-...)
- So ***please*** do "interrupt" with questions & comments -- it's what we're **here** for!

STRAKT

A compact “hello world” [1]

```
#include <Python.h>

static PyObject*
tiny(PyObject* self, PyObject* args)
{
    if(!PyArg_ParseTuple(args, ""))
        return 0;
    return Py_BuildValue("s",”hll wrld");
}
```

Points to retain [1]

- always `#include <Python.h>` at the start
- `PyObject*` represents any Python value
- prototype: `static PyObject*` <name> `(PyObject* self, PyObject* args)`
  - `self` is always 0, `args` is the tuple of arguments
- `PyArg_ParseTuple` to receive arguments
- `return 0` to propagate errors
- `Py_BuildValue` to return a result

STRAKT

A compact "hello world" [2]

```
static PyMethodDef tinyFunctions[] = {
    {"tiny", tiny, METH_VARARGS,
     "A tiny but working function."},
    {0}  /* termination sentinel */
};
void
inittiny()
{
    Py_InitModule3("tiny", tinyFunctions,
        "A tiny but working module.”);
}
```

Points to retain [2]

- array of `PyMethodDef` terminated by `{0}`
- each `PyMethodDef` is a struct of 4 fields:
  - `const`† `char*` the function name shown to Python
  - C function pointer (must have right prototype)
  - `METH_VARARGS` (might also accept keywords, &c)
  - `const char*` the function's docstring
- `void init`*modulename*`()` the entry point
- `Py_InitModule3` initializes the module

† so to speak...

STRAKT

When you import an extension

- you instruct Python to `import tiny`
- Python then:
  - locates `tiny.pyd` (Windows) or `tiny.so` (Unix-y)
  - loads this dynamic library / shared object
  - locates a function named `inittiny`
  - calls said function (must be argument-less & `void`)
- `inittiny` must initialize module `tiny`
- `import` terminates, Python code reprises
- Python code can now call `tiny.tiny()`

3 keys functions of the "C API"

- `Py_InitModule3(const char* module_name, PyMethodDef* array_of_descriptors, const char* docstring)`
  - returns `PyObject*` module object (may be ignored)
- `Py_BuildValue(const char* format, ...)`
  - returns new `PyObject*` result (typically returned)
  - see sub-URL: **ext/parseTuple.html**
- `PyArg_ParseTuple(PyObject* args_tuple, const char* format, ...)`
  - returns 0 on failure, !=0 on success
  - when 0, just `return 0` yourself to propagate
  - see sub-URL: **ext/buildValue.html**

The distutils: setup.py

```
import distutils.core as dist
dist.setup(name = "tiny",
    version = "1.0",
    description = "A tiny extension",
    maintainer = "Alex Martelli",
    maintainer_email = "alex@strakt.com",
    ext_modules = [ dist.Extension(
        'tiny', sources=['tiny.c']) ]
)
```

Most likely building-bug (Win)

- “Debug” vs “Release” modes
- MSVCRT.DLL vs MSVCRTD.DLL
- Suggested approach:
  - ensure MSVCRT.DLL is always used (**/MD**)
- Alternative:
  - get Python source distribution (good idea!)
  - build a for-debug Python (PYTHON22_D.DLL &c)
  - “install” extension there in VStudio debug builds

Sum two integers

```
static PyObject*
isum(PyObject* self, PyObject* args)
{
   int a, b;
   if(!PyArg_ParseTuple(args,"ii",&a,&b))
      return 0;
   return Py_BuildValue("i", a+b);
}
```

and add to tinyMethods a descriptor line:

```
 {"isum", isum, METH_VARARGS,"Sum two integers"},
```

STRAKT

Testing isum

```
>>> import tiny              # or: reload(tiny)
>>> dir(tiny)
['__doc__', '__file__', '__name__',
  'isum', 'tiny']
>>> tiny.isum(100, 23)
123
>>> tiny.isum(4.56, 7.89)
11
```

- note truncation of arguments to int

For more generality…

- accept arguments as `PyObject*`
- operate on them with generic functions
  - **api/abstract.html**
  - **api/object.html**
  - **api/number.html**
- at the limit, you're "coding Python in C"
- (the net speedup may then be modest!)

STRAKT

Add two objects

```
static PyObject*
sum(PyObject* self, PyObject* args)
{
   PyObject *a, *b;
   if(!PyArg_ParseTuple(args,”OO",&a,&b))
      return 0;
   return PyNumber_Add(a, b);
}
```

and add to tinyMethods a descriptor line:

```
 {"sum", sum, METH_VARARGS,"Sum two objects"},
```

STRAKT

Testing sum

```
>>> import tiny          # or: reload(tiny)
>>> dir(tiny)
['__doc__', '__file__', '__name__',
  'isum', ‘sum’, 'tiny']
>>> print tiny.sum(4.56, 7.89)
12.45
>>> print tiny.sum(‘be’, ‘bop’)
bebop
```

- PyNumber_Add is not just for numbers

Reference counting

- **ext/refcounts.html** and ff
- **api/countingRefs.html** and ff
- PyObject *always* lives on the heap
- no single "owner": count of references
  - `Py_XINCREF(x)` to own a new reference
  - `Py_XDECREF(x)` to disown a reference
- Object goes away when a decref makes the reference count become 0

STRAKT

Reference counting rules

- **Borrowed** references (BRs) vs **new**
- use BRs only “briefly”, else Py_XINCREF
- a few functions return BRs (GetItem...)
- most functions transfer ownership of returned obj (including *your* functions)
- return a NULL PyObject* -> “exception”
- most arguments are BRs (exc: SetItem of tuples and lists only -- not of dicts, sequences)

STRAKT

Test-first reference counts!

- **Expect** some reference count errors!
- Thus, code test-first: Python / C
- Python: `id(x)`, `sys.getrefcount(x)`
- C: given `PyObject *x`…:
  - `x <-> id(x)`
  - `x->ob_refcount <-> sys.getrefcount(x)`
- `Py_TRACE_REFS,Py_REF_DEBUG,COUNT_ALLOCS`
- See **Include/object.h**

STRAKT

Exception handling

- “handle” (as in try/except):
  - PyErr_ExceptionMatches, PyErr_Clear
  - detect via NULL, -1, or PyErr_Occurred
- “raise”:
  - return PyErr_Format(...)
- warnings: PyErr_Warn
- **api/exceptionHandling.html**

Exception handling example

```
result = PyNumber_Add(a, b);
if(!result) {
    if(!PyErr_ExceptionMatches(PyExc_TypeError))
        return 0;
    PyErr_Clear();
    if(PyObject_IsTrue(b))
        return PyErr_Format(PyExc_RuntimeError,
            "Cannot sum arguments");
    result = a;
    Py_XINCREF(result);
}
```

Your own type objects

- Prototype in Python (with type=class)
- In C: `PyTypeObject`
- **http://www.python.org/dev/doc/devel/api/type-structs.html**
- **Include/object.h**

STRAKT

PyTypeObject contents

- standard functions (dealloc, call, str...)
- blocks: number / sequence / mapping
- docstring
- type-features flag field
  - in-place operation
  - built-in type checking (coercion not needed)
  - rich comparisons
  - support for weak references
  - ...

STRAKT

An Example PyTypeObject [1]

```
static PyTypeObject intpair_t = {
/* head */         PyObject_HEAD_INIT(0) /* for VC++ */
/* internal */     0, /* must be 0 */
/* type name */    "intpair",
/* basicsize */    sizeof(intpair),
/* itemsize */     0, /* 0 except variable-size type */
/* dealloc */      (destructor)_PyObject_Del,
/* print */        0, /* usually 0 (use str instead) */
/* getattr */      0, /* usually 0 (see getattro) */
/* setattr */      0, /* usually 0 (see setattro) */
/* compare*/       0, /* see also richcompare */
/* repr */         (reprfunc)intpair_str,
/* as_number */    0,
/* as_sequence */ 0,
/* as_mapping */  0,
   ...
```

STRAKT

An Example PyTypeObject [2]

```
...
/* hash */         0, /* 0 unless immutable */
/* call */         0, /* 0 unless callable */
/* str */          0, /* 0 -> same as repr */
/* getattro */     PyObject_GenericGetAttr,
/* setattro */     PyObject_GenericSetAttr,
/* as_buffer */    0, /* 0 unless 'buffer-type' */
/* flags */        Py_TPFLAGS_DEFAULT, /* &c... */
/* docstring */    "2 ints (first,second)",
/* traverse */     0, /* for GC only */
/* clear */        0, /* for GC only */
/* richcompare */ 0, /* block of rich-comparisons */
/* weaklistoff */ 0, /* !=0 if weakly-referenceable */
/* iter */         0, /* for iterables only */
/* iternext */     0, /* for iterators only */
...
```

An Example PyTypeObject [3]

```
...
/* methods */      0, /* if the type has methods */
/* members */      intpair_members,
/* getset */       0, /* for properties */
/* base */         0, /* 0 -> object */
/* dict */         0, /* built by PyType_Ready */
/* descr_get */    0, /* for descriptors */
/* descr_set */    0, /* for descriptors */
/* dictoffset */   0, /* if ‘expando’ type */
/* init */         intpair_init,
/* alloc */        PyType_GenericAlloc,
/* new */          intpair_new,
/* free */         _PyObject_Del,
};
```

STRAKT

Non-0 PyTypeObject fields

```
/* head */          PyObject_HEAD_INIT(0)
/* type name */     "intpair",
/* basicsize */     sizeof(intpair),
/* dealloc */       (destructor)_PyObject_Del,
/* repr */          (reprfunc)intpair_str,
/* getattro */      PyObject_GenericGetAttr,
/* setattro */      PyObject_GenericSetAttr,
/* flags */         Py_TPFLAGS_DEFAULT,
/* docstring */     "2 integers (first,second)",
/* members */       intpair_members,
/* init */          intpair_init,
/* alloc */         PyType_GenericAlloc,
/* new */           intpair_new,
/* free */          _PyObject_Del,
```

intpair, intpair_str

```
typedef struct {
    PyObject_HEAD
    long first, second;
} intpair;

/* Used for both repr() and str()...: */
static PyObject*
intpair_str(intpair *self)
{
    return PyString_FromFormat(
        “intpair(%ld,%ld)”,
        self->first, self->second);
}
```

intpair_members, intpair_new

```
static PyMemberDef intpair_members[] = {
 {"first", T_LONG, offsetof(intpair, first), },
 {"second", T_LONG, offsetof(intpair, second), },
 {0}
};

static PyObject*
intpair_new(PyTypeObject* subtype,
    PyObject* args, PyObject* kwds)
{
    return subtype->tp_alloc(subtype, 0);
}
```

intpair_init

```
static int
intpair_init(PyObject* self,
    PyObject* args, PyObject* kwds)
{
    static char *kwlist[] = {
        "first", "second", 0};
    int f, s;
    if(!PyArg_ParseTupleAndKeywords(
        args, kwds, "ii", kwlist, &f, &s))
            return -1;
    ((intpair*)self)->first = f;
    ((intpair*)self)->second = s;
    return 0;
}
```

STRAKT

includes, initintpair

```
#include "Python.h"
#include "structmember.h"
   ...
void
initintpair(void)
{
    static PyMethodDef nomet[] = { {0} };
    PyObject*
  self=Py_InitModule("intpair",nomet);
    intpair_t.ob_type = &PyType_Type; /* VC++ */
    PyType_Ready(&intpair_t);
    PyObject_SetAttrString(self, "intpair",
        (PyObject*)&intpair_t);
}
```

STRAKT