Notes for "Writing Python extensions in C"

Noufal Ibrahim

28 February 2011

Contents

1 Introduction

1.1 Aims

This are the notes accompanying the tutorial titled "Writing Python extensions in C" presented at PyCon on March 10, 2011 by Noufal Ibrahim. The presentation is linked to at

http://us.pycon.org/2011/schedule/presentations/44/.

This tutorial aims to teach the student how to wrap an existing C library using the Python C-API so that they can use it from within Python.

We will be using the libcsv [1] library as a learning vehicle and will write wrappers for it.

The tutorial is structured around getting a "working solution" rather than something perfect which covers all aspects of the API. The focus is on making the module work as a well behaved python library.

1.2 Preparation

The files and other materials needed for this tutorial are available on github at https://github.com/nibrahim/C-extension-tutorial.

This will contain all the files necessary to go through the tutorial.

As for things which you should already know before starting on this, you should be comfortable with Python, with C and with generally writing and compiling C programs. We will be creating an isolated environment to do all of our work using `virtualenv` so you should spend some time familiarising yourself with it. You should also be comfortable with `git`. We will be using it to to look through the each stage of the tutorial. You can probably look through the code on github directly but it's useful to have it locally to tweak and try things out with.

You will require the following things installed on your machine.

A Gnu/Linux machine While the tutorial could also be done on Windows, it's not been tested there.

Build tools You will require the standard tools usually used to build C programs. The important ones are `gcc`, `make` and `ld`. On a Debian based distribution, you can simply run the `sudo apt-get install build-essentials` command to install the necessary packages.

[1] http://sourceforge.net/projects/libcsv/libcsv

git Git is the versin control system we will be using to walk through the stages of the tutorial. You'll need it installed. On Debian based distributions, you can do a `sudo apt-get install git-core` to get it.

Python 3.1.3 Download the files appropriate for your platform from

http://www.python.org/download/releases/3.1.3/

and install the interpreter. You will require the Python header files to compile the modules.

virtualenv Download the program and use it to create a virtualenv for the new interpreter. http://pypi.python.org/pypi/virtualenv

py.test We will be using this program to write and run test cases for our module. `http://pytest.org/`

After you obtain the above packages and the sources, you will have to edit the `Makefile` in the `pycsv` directory and change the `CINC` line to point to the include directory inside your python source distribution.

1.3 Tutorial structure

This tutorial is structured as a TDD [2] style walkthrough of a series of commits in a repository.

Each pair of commits will first introduce a test case tries out a certain operation (e.g. importing the module) and then will introduce the code that will implement the feature and make the test pass.

The commits are tagged by the stage name and by red/green. So, the commit with tag 3red means that it has a failing test to implement the 3rd feature in the list below.

Shifting from commit to commit will show the various stages of project. These notes describe each commit, why we did it and what we did.

2 Libcsv

libcsv is a simple C libary that is used to parse CSV files. The documentation for the project (available in the `extras/libcsv-3.0.0/csv.pdf` directory) contains details on how it can be used to parse files.

[2] http://en.wikipedia.org/wiki/Test-driven_development

A simple program is included in the `C` called `trial0.c` which uses the library to parse a provided sample file. To try it out,

1. Compile `libcsv.o` in the `extras/libcsv-3.0.0/` directory using `make libcsv.o`
2. Copy that over to the `C` directory.
3. Then run `make` in the `C` directory
4. You will get the `csv_test` executable.

Running it will show you some output from the file which was parsed. The basic idea is quite simple,

1. You declare a `csv_parser` and then `csv_init` it.
2. You call `csv_parse` with the above parser and provide two callback functions.
3. The first callback will be called everytime a cell read is completed.
4. The second callback will be called everytime a row is completely read.
5. `csv_parse` will return the number of bytes that it has parsed. If this is less than the requested number, we have hit an error which we can get using the `csv_error` function.
6. Use the `csv_fini` function to finalise the parsing process (leftover data).
7. Free the data using `csv_free`

Our aim is to construct a python module which will work in a similar way and which will accept Python cell and row callbacks.

We're not doing steps 6 and 7 since they don't add anything substantial to out demonstration.

3 The stages

Each subsection here is describes a test and a piece of the implementaion.

3.1 Skeleton module

3.1.1 The test

The first thing is to create a module. Commit `1red` implements a test that will try to import the module. If it can import the module, it will pass the test. The test will, of course, fail since we have no module created at all. Switch to the `pycsv` directory and run the tests using `py.test tests`. You'll see it fail.

```
def test_import(setup):
    "Try importing the module"
    import pycsv
```

3.1.2 The code

Commit `1green` implements an empty module that can simply be imported. Switching to this commit and typing `make` will compile the module. You can try running the test now and it will pass.

```
#include <Python.h>

static PyMethodDef PyCSVMethods[] = {
  {NULL, NULL, 0, NULL}
};

static struct PyModuleDef pycsvmodule = {
   PyModuleDef_HEAD_INIT,
   "pycsv",   /* name of module */
   "",        /* docstring */
   -1,
   PyCSVMethods
};

PyMODINIT_FUNC /* The only non static function in file */
PyInit_pycsv(void)
{
    return PyModule_Create(&pycsvmodule);
}
```

First of all, you need to include the `Python.h` file. It includes a bunch of standard library headers as well (e.g. `string.h`).

The `PyCSVMethods` array in an an array of all the methods this module will have (i.e. top level functions). For now, we leave it empty. The value it contains right now is a sentinel indicating the end of the array.

The `pycsvmodule` is a structure of type `PyModuleDef`. **This** is the module definition. It contains the following fields

- The module base. This always has to be initialised to `PyModuleDef_HEAD_INIT`.
- The name of the module
- The docstring for the module
- Additional memory needed for this module. We don't need any so we set it to `-1`.
- List of methods.

Finally, The initialisation function which sets up the module and makes it **importable**. The return type should be `PyMODINIT_FUNC` and the name of the function should be `PyInit_modname`. `modname` should be the name of the module which is imported. Our module is called `pycsv` and so the function is called `PyInit_pycsv`. The `PyModule_Create` function will "create" the based on the module structure passed to it and then return it.

You should note that the `PyInit_pycsv` function is the only non static function in the whole module that can be called from outside.

This will create an empty module which can simply be imported.

3.2 Instantiate a CSVFile object

3.2.1 The test

The next step is to create an empty object inside the currently empty module. The `CSVFile` object which will later be the object through which we parse the file. The commit is `2red`. Run the tests using the same command as before and you'll see the test fail telling you that the module doesn't have a `CSVFile` attribute.

3.2.2 The code

We're not showing the entire code here. You'll have to look at the repository to see it.

First, we create a `pycsv_CSVFileObject` object which for now simply contains the `PyObject_HEAD` macro. This represents the "object". The `PyObject_HEAD` macro initialises the references counts some other variables that are used to track the object.

The `pycsv_PyCSVFileType` is the type object (of type `PyTypeObject`) which contains the type of the object. As an example, in regular python, "test" would be the **object** and `str` the **type**. All the methods (e.g. `replace` for strings) would be part of the type rather than the object. The `PyVarObject_HEAD_INIT` macro initialises a few fields like the base class.

We then have the full name of the object and the size of the object. We'll ignore a few of the next fields. The `Py_TPFLAGS_DEFAULT` is bunch of flags that set certain bits in the object. We set it to the defaults for now.

Finally, we have a docstring.

In the init function, the `tp_new` field is set to `PyType_GenericNew` which is a function used to create the object. After this, we call `PyType_Ready` which adds inherited slots and completes initialisation. We then create the module as before and add the the newly created type as an attribute to the module with the name `CSVFile` using `PyModule_AddObject`.

We now have a module with a single type inside it that can be instantiated. It doesn't do anything

3.2.3 Reference counts

Python uses manual reference counting. It is the programmers responsibility to increment and decrement the reference counts of the objects her code uses. Once an objects count reaches zero, it will be reclaimed by the runtime.

The two macros used to increment and decrement counts are `Py_INCREF` and `Py_DECREF` respectively. When the latter is called, if the count reaches zero, the object is freed by invoking the free function in the corresponding type object.

You can "own" a reference which means you call `Py_INCREF` on the object in question. You are now responsible for freeing decrementing the count when your business is done. This is often done when you create an

instance of an object that is long living. One example in our code is where we increment the reference count of the `pycsv_PyCSVFileType` object. Our module "owns" a reference to this object now. Many functions that create objects (e.g. `PyLong_FromLong`) return the ownership back to the caller. This means that the caller is responsible for freeing it.

The other option is that you can "borrow" a reference. This means that some other function has incremented the reference count and you simply use the object. In this situation, neither increment nor decrement the counts for the object but just hold onto and use it. You can think of the actual owner and you being fused into one for the duration of the the ownership of the object. Usually, if we pass an object into a function, it has borrowed the ownership from you for its duration. It will not decrement or increment the counts (unless of course, it needs ownership for itself).

Usually, in C extensions, references are borrowed from the Python callers. Sometimes (like we will do with the file argument below), we need to maintain a reference by ourselves till our business is completed and so we increment the count.

In the case of C functions returning objects to the Python namespace, the ownership is usually passed to the calling function and it is its job to decrement the count when it's business is over.

3.3 Instantiate CSVFile only with arguments

3.3.1 The test

The next step is to handle arguments. We'll bundle all of this into a test that expects our code to raise a `TypeError` when `CSVFile` is instantiated without any arguments. The commit is `3red`. Run the test and watch it fail saying that `TypeError` is not raised.

3.3.2 The code

We cleanup a few names in this version of the file. Things like `pycsv_CSVFileObject` of the previous commit are renamed to `PyCSVFile`.

This change is accomplished by adding an "init" function. This corresponds to the `__init__` function of a regular python object.

Our `PyCSVFile_init` function is quite simple. It receives three arguments. The first argument is similar to the regular python `self`. The `args`

argument denotes positional arguments and the `kwds` denotes keyword arguments.

You'll notice the `PyObject` declaration we've made. `PyObject` is sort of like the `void *` of the Python C API. It refers to any python object. There are more specific types of objects like `PyFileObject` for files. All of these are `PyObject` objects with a some extras - a superclass of sorts.

We then call the `PyArg_ParseTuple` with the `args` parameter, a format string which tells us how to parse the arguments we're passing to it. For details of the format specifiers, you should refer to the Python C-API documentation. The `O` in this case indicates that one argument should be handled and that it should simply be assigned without any conversion to the first argument after the format string given to `PyArg_ParseTuple`. In other words, if this is successful, `file` will contain the file which we've passed to the initialisation function. If there's a mismatch in the number of arguments, the function will return -1 which signals an error condition and the interpreter will raise an appropriate exception.

We then attach this function into the `tp_init` slot of the `PyCSVFileType` structure and try our test again. It passes.

3.4 Adding attributes

3.4.1 The test

First, we initialise the `CSVFile` object with an open file and then we check the `_file` attribute to see if it has been stored properly. `e12a7b2575` implements the test. It will fail.

3.4.2 The code

`a7028c8da7` implements the feature.

First we keep a new pointer inside the `PyCSVFile` structure. This is what we will be assigning to. The C name of the variable has nothing to do with the name which will be visible to python calling functions but we'll come to that later.

We create a few new functions in this commit. The first one is a deallocator `PyCSVFile_dealloc`. When this object is to be garbage collected, it has to release any holds it has on the file object and free it.

Next we create an allocator function `PyCSVFile_new`. This is distinct from the initialiser. This will be called at **all** times when we create a new `PyCSVFile` object. The initialiser on the other hand might not be called in some circumstances (e.g. unpickling). Inside, we create call the `tp_alloc` function of the corresponding `type`. If the allocation was successful, we return `None`.

The `PyCSVFile_init` function is also enhanced. Now, we actually take the argument which we receive, increment it's reference count (since now we have one more hold on it via. the `_file` attribute) and assign it to `file`. We decrement the refcount of what `file` used to point to but do it only after we assign it to the new value. This is erring on the side of paranoia. It might be possible that if we decrement the count first, destructor code gets called that actually tries to access `file` member which will mess things up.

Now, we add a `PyCSVFile_members` variable that holds the `_file` attribute. The `offsetoff` function used to tell the interpreter the offset of the attribute is declared in the `structmember.h` header file so we include that.

We fill in the extra slots in the type definition and give it a shot. It passes.

3.5 Adding methods

3.5.1 The test

`04856fb29` introduces a new test that tries to call the `parse` method on our `CSVFile` object. Our task is to create a stub function that can be called.

3.5.2 The code

Since we're not adding any functionality into this function, this change is quite simple. It's done in `3634a5826`.

We first create a `PyCSV_parse` function that simply returns `None`.

We then create a `PyCSVFile_methods` array (similar to the `PyCSVFile_members`) which a reference to the newly created function under the name `parse`. We use `METH_NOARGS` to tell python that the function doesn't receive any arguments from the caller (as it's being invoked in the test).

We add the `PyCSVFile_methods` to the `tp_methods` slot of the type object and we're done.

3.6 Doing the parsing

3.6.1 The test

Commit `6red` introduces a test that actually tries to parse a csv file using a pair of callbacks to see if we read back the values correctly.

3.6.2 The code

`6green` implements the feature. It fleshes out the `parse` method.

We use `PyArg_ParseTuple` to handle default arguments this time. We set our placeholder `PyObject` variables to `Py_None` (the python `None`) and then do the parsing. The `|` character indicates optional arguments. We do it this way so that the previous test doesn't break.

We then use the `PyObject_CallMethod` to read the entire contents of the file into a variable. This is similar so saying `data  f.read()`= in python. We interpret what we read as a ASCII encoded Unicode string and then extract the internal character buffer into the `data` variable so that the low level C library can access it.

We then instantiate a structure we declare on top of the file to hold both the callbacks and the user provided data.

We initialise the parser and then call it using two custom callback functions that we next describe. And then we return `None`

The callback functions defined at the top of the file are quite simple.

The `cell_callback` is what's called when we finish reading a cell. We get the python callbacks and convert the cell data into a python unicode object. We then use the `PyObject_CallFunctionObjArgs` function to call the provided callback with the cell contents and the user provided data as an argument.

The row callback is similar.

As we mentioned earlier, exceptions have to explicitly tested in the C version of functions. If the passed callback has an improper number of arguments, `PyObject_CallFunctionObjArgs` will fail but we're not testing it here. The **right** way to do it would be to somewhow abort the parsing process if this happens and to raise the appropriate exception. We've left it out for simplicity but be aware that this is a flawed implementation.

This is also the time we need to actually link to the `libcsv.o` file so the Makefile is altered accordingly. There is a precompiled version in the

repository but if your platform is different, you can compile it in the in `extras/libcsv-3.0.0/` and copy it over here.

3.7 Custom exceptions

3.7.1 The test

Commit `7red` introduces a test that checks to see if a certain exception is raised if the file we're parsing is corrupt.

3.7.2 The code

`e500c8b` implements this feature. First, we use the `CSV_STRICT` flag while parsing to make sure that the `csv_parse` function aborts on broken files. We then keep a count of length of the file we're parsing and how much actually gets parsed. If they're not the same, we get the error message and use the `PyErr_SetString` to set the error message. As mentioned earlier, returning a NULL will tell the interpreter than an error has occurred.

We also create a new exception object in the module. We declare a `PyObject` to hold the exception and in the module init function, we use `PyErr_NewException` to create a new exception (derived from `Exception` by default) and add it to the module's attributes. The error is called a `pycsv.ParseError`.

3.7.3 Exceptions

As a rule, python functions implemented in C always return python objects.

Pythons exception system works in a fashion similar to the UNIX errno variable. When an error is encoutered, the function that encountered it is responsible for setting 3 global variables that hold the type of the exception, the instance of the exception and the traceback. To signal the error condition, the function will return an error value (usually a `NULL` pointer).

The three globals variables are controlled using one of a number of functions. The most common one (and the one we use) is `PyErr_SetString`. It expectes an exception object and a C string which contains the string that the exception is instantiated with. `raise KeyError("foo")` would become `PyErr_SetString(PyExc_KeyError,"foo")`.

There are more functions that can check for exceptions, store and retrieve them, clear them etc. which are mentioned in the documentation.

3.8 Packaging

This is done in commit `7green` and doesn't have any tests.

We create a `pycsv` directory and move all the files to inside it. We then create a top level `distutils` `setup.py` file. We remove things like the Makefile etc. and specify the include directories, libraries, library directories and the source file as an `Extension`.

We then make the entire package along with description and other things and include the extension in it.

Running `python setup.py build` will create the module. To use it, you will have to provide the path to the `libcsv.so` file using your `LD_LIBRARY_PATH`.

4 Conclusions

This completes our implementation of the `libcsv` wrapper.

The Python documentation includes a page entitled "Extending and Embedding the Python Interpreter" which is a tutorial style introduction to creating extension modules.

The "Python/C API Reference Manual" contains the nuts and bolts of the API. It serves as a handy reference while actually doing this kind of thing.

The source distribution contains a few boilerplate modules prefixed with `xx` (e.g. `Modules/xxmodule.c`) which can be used as a starting point to implement your own modules.