Migrating C extensions

Python 3 has many changes to the C API, including in the API for defining classes and initializing modules. This means that every single C extension has to be modified to run under Python 3. Some of the changes are simple and some are not, but you will have to do them all by hand as `2to3` only handles Python code. That also means that you can't support Python 2 and Python 3 by `2to3`conversion. Luckily the the C pre-processor make it fairly easy to support both Python 2 and Python 3 from the same code. This is a standard way of supporting different versions of API's in C it will be standard fare for C-programmers. So there are no ugly hacks involved, just some less than pretty `#if` and `#ifndef`statements.

Before you start

There are some things you can do before you start the actual work of addnig Python 3 support. The first one is to remove any usage of some old aliases you don't need any more. For example the `RO` macro has been removed. It was only a shorthand for `READONLY`, so if you used `RO` in your code you can replace it with `READONLY` instead.

Other common redefinitions are `statichere` and `staticforward`. They are workarounds for compatibility with certain compilers. For well behaving compilers they are just redefinitions of `static` so in Python 3 they are now gone since there are now have well behaving compilers on all platforms that CPython supports. If you use them in your code, you can replace them with `static`.

Another change you can do as preparation is to move away from `PyClassObject`. That's the long deprecated "old-style classes" which are removed in Python 3. You should be able to move over to `PyTypeObject` with no big problem.

Object initialization

One of the less obvious errors encountered when compiling C extensions under Python 3 is the error `"missing braces around initializer"`, which you will get when you initialize a Python class. You can indeed get around that by placing a couple of braces in the correct places, but a better solution is to replace the`PyObject_HEAD_INIT` macro to the `PyVarObject_HEAD_INIT` macro. The change was done to conform with C's strict aliasing rules, you can find more information in PEP 3123[1] if you are interested.

Code that previously looked like this:

```
static PyTypeObject mytype = {
    PyObject_HEAD_INIT(NULL)
    0,
    ...
};
```

Should now look like this:

```
static PyTypeObject mytype = {
    PyVarObject_HEAD_INIT(NULL, 0)
    ...
};
```

This will work in Python 2.6 and 2.7 as well. If you need to support Python 2.5 or earlier, you can simply define the `PyVarObject_HEAD_INIT` macro if it's missing:

```
#ifndef PyVarObject_HEAD_INIT
    #define PyVarObject_HEAD_INIT(type, size) \
        PyObject_HEAD_INIT(type) size,
#endif
```

Another change in the object header is that the `PyObject_HEAD` macro has changed so that `ob_type` is now in a nested structure. This means you no longer can pick the `ob_type` directly from the struct, so code like `ob->ob_type` stops working. You should replace this with `Py_TYPE(ob)`. The `Py_TYPE` macro doesn't appear until Python 2.6, so to support earlier versions we make another `#ifndef`:

```
#ifndef Py_TYPE
    #define Py_TYPE(ob) (((PyObject*)(ob))->ob_type)
#endif
```

In both cases the definitions above are taken directly from the Python 2.6 headers, where they are defined for forward compatibility purposes with Python 3. They work well in earlier Python versions as well, so this is a trick you can use as a general rule; if you need to use a macro that is defined in Python 3 and Python 2.6, just steal the Python 2.6 or Python 2.7 definition and put it inside an `#ifndef`.

Module initialization

Another big change is in the module initialization. There are many changes here, and as a result the module initialization usually ends up as part of the C extension with the most pre-processor commands or macros.

The family of functions to initialize modules, such as `Py_InitModule3` are gone. Instead, you should use `PyModule_Create`. Where `Py_InitModule3` took a couple of parameters `PyModule_Create` needs a `PyModuleDef` struct. If you want to support Python 2 you need to wrap this code with an `#if PY_MAJOR_VERSION >=3`, both when you define the struct and when you use it.

```
#if PY_MAJOR_VERSION >= 3
    static struct PyModuleDef moduledef = {
        PyModuleDef_HEAD_INIT,
        "themodulename",     /* m_name */
        "This is a module",  /* m_doc */
        -1,                  /* m_size */
        module_functions,    /* m_methods */
        NULL,                /* m_reload */
        NULL,                /* m_traverse */
        NULL,                /* m_clear */
        NULL,                /* m_free */
    };
#endif

...

#if PY_MAJOR_VERSION >= 3
    m = PyModule_Create(&moduledef);
#else
    m = Py_InitModule3("themodulename",
```

```
        module_functions, "This is a module");
#endif
```

If you want to separate the `#if` statements from the code you can make a macro definition. I've used this one, although it doesn't support the extra functions like reload and traverse:

```
#if PY_MAJOR_VERSION >= 3
    #define MOD_DEF(ob, name, doc, methods) \
        static struct PyModuleDef moduledef = { \
            PyModuleDef_HEAD_INIT, name, doc, -1, methods, }; \
        ob = PyModule_Create(&moduledef);
#else
    #define MOD_DEF(ob, name, doc, methods) \
        ob = Py_InitModule3(name, methods, doc);
#endif
```

The definition of the module initialization function has also changed. In Python 2 you declared a function to initialize the module like this:

```
PyMODINIT_FUNC init<yourmodulename>(void)
```

In Python 3 this has changed to:

```
PyMODINIT_FUNC PyInit_<yourmodulename>(void)
```

It's not just the name that has changed; it's also the value of `PyMODINIT_FUNC`. In Python 2 it's typically `void` while in Python 3 it now returns a `PyObject*`. You have to return `NULL` if an error happened and you need to return the module object if initialization succeeded. There are various ways of dealing with this if you need both Python 3 and Python 2 support, starting with using multiple `#ifPY_MAJOR_VERSION >= 3` in the function. However, that gets ugly, especially in the function definition:

```
PyMODINIT_FUNC
#if PY_MAJOR_VERSION >= 3
PyInit_<yourmodulename>(void)
#else
init<yourmodulename>(void)
#endif
{
...
```

It works, but it is not very readable code. It gets slightly better by using a macro:g

```
#if PY_MAJOR_VERSION >= 3
    #define MOD_INIT(name) PyMODINIT_FUNC PyInit_##name(void)
#else
    #define MOD_INIT(name) PyMODINIT_FUNC init##name(void)
#endif

MODINIT(themodulename)
{
...
}
```

But you still have to either have `#if` statements in the function to determine if you should return a value or not, or make yet another macro for that.

Another option is to define three functions. Firstly the actual module initialization function, returning a `PyObject*` and then two wrappers. One for Python 3 that calls the first and returns the value and one for Python 2 that calls the module initizaliation without returning a value:

```
// Python 3 module initialization
static PyObject *
moduleinit(void)
{
    MOD_DEF(m, "themodulename",
            "This is the module docstring",
    module_methods)

    if (m == NULL)
        return NULL;

    if (PyModule_AddObject(m, "hookable",
           (PyObject *)&hookabletype) < 0)
        return NULL;

    return m;
}

#if PY_MAJOR_VERSION < 3
    PyMODINIT_FUNC initthemodulename(void)
    {
        moduleinit();
    }
#else
    PyMODINIT_FUNC PyInit_themodulename(void)
    {
        return moduleinit();
    }
#endif
```

As you see the module initialization will in any case end up with a lot of `#ifPY_MAJOR_VERSION >= 3`. A complete example of all these `#if` statements is this, taken from `zope.proxy`:

```
#if PY_MAJOR_VERSION >= 3
  static struct PyModuleDef moduledef = {
    PyModuleDef_HEAD_INIT,
    "_zope_proxy_proxy", /* m_name */
    module___doc__,      /* m_doc */
    -1,                  /* m_size */
    module_functions,    /* m_methods */
    NULL,                /* m_reload */
    NULL,                /* m_traverse */
    NULL,                /* m_clear */
    NULL,                /* m_free */
  };
#endif

static PyObject *
moduleinit(void)
{
    PyObject *m;

#if PY_MAJOR_VERSION >= 3
    m = PyModule_Create(&moduledef);
#else
    m = Py_InitModule3("_zope_proxy_proxy",
                        module_functions, module___doc__);
```

```
#endif

    if (m == NULL)
        return NULL;

    if (empty_tuple == NULL)
        empty_tuple = PyTuple_New(0);

    ProxyType.tp_free = _PyObject_GC_Del;

    if (PyType_Ready(&ProxyType) < 0)
        return NULL;

    Py_INCREF(&ProxyType);
    PyModule_AddObject(m, "ProxyBase", (PyObject *)&ProxyType);

    if (api_object == NULL) {
        api_object = PyCObject_FromVoidPtr(&wrapper_capi, NULL);
        if (api_object == NULL)
        return NULL;
    }
    Py_INCREF(api_object);
    PyModule_AddObject(m, "_CAPI", api_object);

  return m;
}

#if PY_MAJOR_VERSION < 3
    PyMODINIT_FUNC
    init_zope_proxy_proxy(void)
    {
        moduleinit();
    }
#else
    PyMODINIT_FUNC
    PyInit__zope_proxy_proxy(void)
    {
        return moduleinit();
    }
#endif
```

If you don’t like all the version tests, you can put all of these together before the function definition and use macros for anything that differs. Here is the same `zope.proxy` module, after I replaced all the `#if` tests with one block of definitions in the beginning:

```
#if PY_MAJOR_VERSION >= 3
  #define MOD_ERROR_VAL NULL
  #define MOD_SUCCESS_VAL(val) val
  #define MOD_INIT(name) PyMODINIT_FUNC PyInit_##name(void)
  #define MOD_DEF(ob, name, doc, methods) \
          static struct PyModuleDef moduledef = { \
            PyModuleDef_HEAD_INIT, name, doc, -1, methods, }; \
          ob = PyModule_Create(&moduledef);
#else
  #define MOD_ERROR_VAL
  #define MOD_SUCCESS_VAL(val)
  #define MOD_INIT(name) void init##name(void)
  #define MOD_DEF(ob, name, doc, methods) \
          ob = Py_InitModule3(name, methods, doc);
#endif
```

```
MOD_INIT(_zope_proxy_proxy)
{
    PyObject *m;

    MOD_DEF(m, "_zope_proxy_proxy", module___doc__,
            module_functions)

    if (m == NULL)
        return MOD_ERROR_VAL;

    if (empty_tuple == NULL)
        empty_tuple = PyTuple_New(0);

    ProxyType.tp_free = _PyObject_GC_Del;

    if (PyType_Ready(&ProxyType) < 0)
        return MOD_ERROR_VAL;

    Py_INCREF(&ProxyType);
    PyModule_AddObject(m, "ProxyBase", (PyObject *)&ProxyType);

    if (api_object == NULL) {
        api_object = PyCObject_FromVoidPtr(&wrapper_capi, NULL);
        if (api_object == NULL)
        return MOD_ERROR_VAL;
    }
    Py_INCREF(api_object);
    PyModule_AddObject(m, "_CAPI", api_object);

    return MOD_SUCCESS_VAL(m);

}
```

This is by far my preferred version, for stylistic reasons, but ultimately it's a matter of taste and coding style if you prefer the in-line `#if` statements or if you like to use many `#define` macros. So you choose what fits best with your coding style.

Changes in Python

The changes in Python are of course reflected in the C API. These are usually easy to handle. A typical example here is the unification of `int` and `long` types. Although in Python it behaves like the `long` type is gone, it's actually the `int` type that has been removed and the `long` type renamed. However, in the C API it hasn't been renamed. That means that all the functions that returned Python `int` objects are now gone and you need to replace them with the functions that returns Python `long` objects. This means that `PyInt_FromLong` must be replaced with `PyLong_FromLong`, `PyInt_FromString` with `PyLong_FromString` etc. If you need to keep Python 2 compatibility you have to replace it conditionally:

```
#if PY_MAJOR_VERSION >= 3
    PyModule_AddObject(m, "val", PyLong_FromLong(2));
#else
    PyModule_AddObject(m, "val", PyInt_FromLong(2));
#endif
```

Also in this case a `#define` makes for cleaner code if you need to do it more than once:

```
#if PY_MAJOR_VERSION >= 3
    #define PyInt_FromLong PyLong_FromLong
#endif

PyModule_AddObject(m, "val", PyInt_FromLong(2));
```

In Python 3.2 the CObject API was removed. It was replaced with the Capsule API, which is also available for Python 2.7 and 3.1. For the simple usecase of just wrapping a C value the change is simple:

```
#if PY_MAJOR_VERSION < 3
    c_api = PyCObject_FromVoidPtr ((void *)pointer_to_value, NULL);
#else
    c_api = PyCapsule_New((void *)pointer_to_value, NULL, NULL);
#endif
```

Other things changed in Python that you are likely to encounter is the removal of the support for `__cmp__()` methods. The `_typeobject` structure used for defining a new type includes a place for a `__cmp__()` method definition. It's still there in Python 3 for compatibility but it's now ignored. The `cmpfunc` type definition and the `PyObject_Compare` functions have been removed as well. The only way to get full Python 3 compatibility here is to implement rich comparison functionality. There is support for that back to Python 2.1, so there is no problem with backwards compatibility.

Strings and Unicode

The changes in strings, Unicode and bytes are of course one of the biggest changes also when writing C extensions. In the C API, as with integers, there has been no renaming amongst the strings and the `unicode` type is still called `unicode`. The `str` type and all accompanying support functions are gone and the new `bytes` type has replaced it.

This means that if your extension returns or handles binary data you will in Python 2 get and return `PyString` objects, while you in Python 3 will handle `PyBytes` objects. Where you handle text data you should in Python 2 accept both `PyString` and `PyUnicode` while in Python 3 only `PyUnicode` is relevant. This can be handled with the same techniques as for `int` and `long` above, you can either make two versions of the code and choose between them with an `#if`, or you can redefine the missing `PyString` functions in Python 3 as either `PyBytes` or `PyUnicode` depending on what you need.

Footnotes

[1] http://www.python.org/dev/peps/pep-3123/