Porting Extension Modules to 3.0

Release 2.7.1

Guido van Rossum
Fred L. Drake, Jr., editor

May 29, 2011

Python Software Foundation
Email: docs@python.org

Contents

author Benjamin Peterson

Abstract

Although changing the C-API was not one of Python 3.0's objectives, the many Python level changes made leaving 2.x's API intact impossible. In fact, some changes such as `int()` and `long()` unification are more obvious on the C level. This document endeavors to document incompatibilities and how they can be worked around.

1 Conditional compilation

The easiest way to compile only some code for 3.0 is to check if `PY_MAJOR_VERSION` is greater than or equal to 3.

```
#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif
```

API functions that are not present can be aliased to their equivalents within conditional blocks.

2 Changes to Object APIs

Python 3.0 merged together some types with similar functions while cleanly separating others.

2.1 str/unicode Unification

Python 3.0's `str()` (`PyString_*` functions in C) type is equivalent to 2.x's `unicode()` (`PyUnicode_*`). The old 8-bit string type has become `bytes()`. Python 2.6 and later provide a compatibility header, `bytesobject.h`, mapping `PyBytes` names to `PyString` ones. For best compatibility with 3.0, `PyUnicode` should be used for textual data and `PyBytes` for binary data. It's also important to remember that `PyBytes` and `PyUnicode` in 3.0 are not interchangeable like `PyString` and `PyUnicode` are in 2.x. The following example shows best practices with regards to `PyUnicode`, `PyString`, and `PyBytes`.

```
#include "stdlib.h"
#include "Python.h"
#include "bytesobject.h"

/* text example */
static PyObject *
say_hello(PyObject *self, PyObject *args) {
    PyObject *name, *result;

    if (!PyArg_ParseTuple(args, "U:say_hello", &name))
        return NULL;

    result = PyUnicode_FromFormat("Hello, %S!", name);
    return result;
}

/* just a forward */
static char * do_encode(PyObject *);

/* bytes example */
static PyObject *
encode_object(PyObject *self, PyObject *args) {
    char *encoded;
    PyObject *result, *myobj;

    if (!PyArg_ParseTuple(args, "O:encode_object", &myobj))
        return NULL;

    encoded = do_encode(myobj);
    if (encoded == NULL)
        return NULL;
    result = PyBytes_FromString(encoded);
    free(encoded);
    return result;
}
```

2.2 long/int Unification

In Python 3.0, there is only one integer type. It is called `int()` on the Python level, but actually corresponds to 2.x's `long()` type. In the C-API, `PyInt_*` functions are replaced by their `PyLong_*` neighbors. The best course of action here is using the `PyInt_*` functions aliased to `PyLong_*` found in `intobject.h`. The abstract `PyNumber_*` APIs can also be used in some cases.

```
#include "Python.h"
#include "intobject.h"

static PyObject *
add_ints(PyObject *self, PyObject *args) {
    int one, two;
    PyObject *result;

    if (!PyArg_ParseTuple(args, "ii:add_ints", &one, &two))
        return NULL;

    return PyInt_FromLong(one + two);
}
```

3 Module initialization and state

Python 3.0 has a revamped extension module initialization system. (See **PEP 3121**.) Instead of storing module state in globals, they should be stored in an interpreter specific structure. Creating modules that act correctly in both 2.x and 3.0 is tricky. The following simple example demonstrates how.

```
#include "Python.h"

struct module_state {
    PyObject *error;
};

#if PY_MAJOR_VERSION >= 3
#define GETSTATE(m) ((struct module_state*)PyModule_GetState(m))
#else
#define GETSTATE(m) (&_state)
static struct module_state _state;
#endif

static PyObject *
error_out(PyObject *m) {
    struct module_state *st = GETSTATE(m);
    PyErr_SetString(st->error, "something bad happened");
    return NULL;
}

static PyMethodDef myextension_methods[] = {
    {"error_out", (PyCFunction)error_out, METH_NOARGS, NULL},
    {NULL, NULL}
};

#if PY_MAJOR_VERSION >= 3
```

```c
static int myextension_traverse(PyObject *m, visitproc visit, void *arg) {
    Py_VISIT(GETSTATE(m)->error);
    return 0;
}

static int myextension_clear(PyObject *m) {
    Py_CLEAR(GETSTATE(m)->error);
    return 0;
}


static struct PyModuleDef moduledef = {
        PyModuleDef_HEAD_INIT,
        "myextension",
        NULL,
        sizeof(struct module_state),
        myextension_methods,
        NULL,
        myextension_traverse,
        myextension_clear,
        NULL
};

#define INITERROR return NULL

PyObject *
PyInit_myextension(void)

#else
#define INITERROR return

void
initmyextension(void)
#endif
{
#if PY_MAJOR_VERSION >= 3
    PyObject *module = PyModule_Create(&moduledef);
#else
    PyObject *module = Py_InitModule("myextension", myextension_methods);
#endif

    if (module == NULL)
        INITERROR;
    struct module_state *st = GETSTATE(module);

    st->error = PyErr_NewException("myextension.Error", NULL, NULL);
    if (st->error == NULL) {
        Py_DECREF(module);
        INITERROR;
    }

#if PY_MAJOR_VERSION >= 3
    return module;
```

```
#endif
}
```

4 Other options

If you are writing a new extension module, you might consider Cython. It translates a Python-like language to C. The extension modules it creates are compatible with Python 3.x and 2.x.

Index

P