Porting Extension Modules to Python 3

Release 3.6.3rc1

Guido van Rossum
and the Python development team

September 30, 2017

Python Software Foundation
Email: docs@python.org

Contents

1 Conditional compilation 1

2 Changes to Object APIs 2
2.1 str/unicode Unification 2
2.2 long/int Unification 3

3 Module initialization and state 3

4 CObject replaced with Capsule 4

5 Other options 7

Index 8

author Benjamin Peterson

Abstract

Although changing the C-API was not one of Python 3's objectives, the many Python-level changes made leaving Python 2'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 Python 3 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 merged together some types with similar functions while cleanly separating others.

2.1 str/unicode Unification

Python 3's str() type is equivalent to Python 2's unicode(); the C functions are called PyUnicode_* for both. The old 8-bit string type has become bytes(), with C functions called PyBytes_*. Python 2.6 and later provide a compatibility header, bytesobject.h, mapping PyBytes names to PyString ones. For best compatibility with Python 3, PyUnicode should be used for textual data and PyBytes for binary data. It's also important to remember that PyBytes and PyUnicode in Python 3 are not interchangeable like PyString and PyUnicode are in Python 2. 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

Python 3 has only one integer type, int(). But it actually corresponds to Python 2's long() type—the int() type used in Python 2 was removed. In the C-API, PyInt_* functions are replaced by their PyLong_* equivalents.

3 Module initialization and state

Python 3 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 Python 2 and Python 3 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

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

PyMODINIT_FUNC
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 CObject replaced with Capsule

The Capsule object was introduced in Python 3.1 and 2.7 to replace CObject. CObjects were useful, but the CObject API was problematic: it didn't permit distinguishing between valid CObjects, which allowed mismatched CObjects to crash the interpreter, and some of its APIs relied on undefined behavior in C. (For further reading on the rationale behind Capsules, please see bpo-5630.)

If you're currently using CObjects, and you want to migrate to 3.1 or newer, you'll need to switch to Capsules. CObject was deprecated in 3.1 and 2.7 and completely removed in Python 3.2. If you only support 2.7, or 3.1 and above, you can simply switch to Capsule. If you need to support Python 3.0, or versions of Python earlier than 2.7, you'll have to support both CObjects and Capsules. (Note that Python 3.0 is no longer

supported, and it is not recommended for production use.)

The following example header file capsulethunk.h may solve the problem for you. Simply write your code against the Capsule API and include this header file after Python.h. Your code will automatically use Capsules in versions of Python with Capsules, and switch to CObjects when Capsules are unavailable.

capsulethunk.h simulates Capsules using CObjects. However, CObject provides no place to store the capsule's "name". As a result the simulated Capsule objects created by capsulethunk.h behave slightly differently from real Capsules. Specifically:

- The name parameter passed in to PyCapsule_New() is ignored.
- The name parameter passed in to PyCapsule_IsValid() and PyCapsule_GetPointer() is ignored, and no error checking of the name is performed.
- PyCapsule_GetName() always returns NULL.
- PyCapsule_SetName() always raises an exception and returns failure. (Since there's no way to store a name in a CObject, noisy failure of PyCapsule_SetName() was deemed preferable to silent failure here. If this is inconvenient, feel free to modify your local copy as you see fit.)

You can find capsulethunk.h in the Python source distribution as Doc/includes/capsulethunk.h. We also include it here for your convenience:

```
#ifndef __CAPSULETHUNK_H
#define __CAPSULETHUNK_H

#if (    (PY_VERSION_HEX <  0x02070000) \
    || ((PY_VERSION_HEX >= 0x03000000) \
     && (PY_VERSION_HEX <  0x03010000)) )

#define __PyCapsule_GetField(capsule, field, default_value) \
    ( PyCapsule_CheckExact(capsule) \
        ? (((PyCObject *)capsule)->field) \
        : (default_value) \
    ) \

#define __PyCapsule_SetField(capsule, field, value) \
    ( PyCapsule_CheckExact(capsule) \
        ? (((PyCObject *)capsule)->field = value), 1 \
        : 0 \
    ) \


#define PyCapsule_Type PyCObject_Type

#define PyCapsule_CheckExact(capsule) (PyCObject_Check(capsule))
#define PyCapsule_IsValid(capsule, name) (PyCObject_Check(capsule))


#define PyCapsule_New(pointer, name, destructor) \
    (PyCObject_FromVoidPtr(pointer, destructor))


#define PyCapsule_GetPointer(capsule, name) \
    (PyCObject_AsVoidPtr(capsule))

/* Don't call PyCObject_SetPointer here, it fails if there's a destructor */
#define PyCapsule_SetPointer(capsule, pointer) \
    __PyCapsule_SetField(capsule, cobject, pointer)
```

```
#define PyCapsule_GetDestructor(capsule) \
    __PyCapsule_GetField(capsule, destructor)

#define PyCapsule_SetDestructor(capsule, dtor) \
    __PyCapsule_SetField(capsule, destructor, dtor)


/*
 * Sorry, there's simply no place
 * to store a Capsule "name" in a CObject.
 */
#define PyCapsule_GetName(capsule) NULL

static int
PyCapsule_SetName(PyObject *capsule, const char *unused)
{
    unused = unused;
    PyErr_SetString(PyExc_NotImplementedError,
        "can't use PyCapsule_SetName with CObjects");
    return 1;
}



#define PyCapsule_GetContext(capsule) \
    __PyCapsule_GetField(capsule, descr)

#define PyCapsule_SetContext(capsule, context) \
    __PyCapsule_SetField(capsule, descr, context)


static void *
PyCapsule_Import(const char *name, int no_block)
{
    PyObject *object = NULL;
    void *return_value = NULL;
    char *trace;
    size_t name_length = (strlen(name) + 1) * sizeof(char);
    char *name_dup = (char *)PyMem_MALLOC(name_length);

    if (!name_dup) {
        return NULL;
    }

    memcpy(name_dup, name, name_length);

    trace = name_dup;
    while (trace) {
        char *dot = strchr(trace, '.');
        if (dot) {
            *dot++ = '\0';
        }

        if (object == NULL) {
            if (no_block) {
                object = PyImport_ImportModuleNoBlock(trace);
```

```
            } else {
                object = PyImport_ImportModule(trace);
                if (!object) {
                    PyErr_Format(PyExc_ImportError,
                        "PyCapsule_Import could not "
                        "import module \"%s\"", trace);
                }
            }
        } else {
            PyObject *object2 = PyObject_GetAttrString(object, trace);
            Py_DECREF(object);
            object = object2;
        }
        if (!object) {
            goto EXIT;
        }

        trace = dot;
    }

    if (PyCObject_Check(object)) {
        PyCObject *cobject = (PyCObject *)object;
        return_value = cobject->cobject;
    } else {
        PyErr_Format(PyExc_AttributeError,
            "PyCapsule_Import \"%s\" is not valid",
            name);
    }

EXIT:
    Py_XDECREF(object);
    if (name_dup) {
        PyMem_FREE(name_dup);
    }
    return return_value;
}

#endif /* #if PY_VERSION_HEX < 0x02070000 */

#endif /* __CAPSULETHUNK_H */
```

5 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 and Python 2.

Index

P

Python Enhancement Proposals
    PEP 3121, 3