Python 3 への拡張モジュール移植

リリース *2.7.17*

Guido van Rossum
and the Python development team

12 月 24, 2019

Python Software Foundation
Email: docs@python.org

目 次

---

author Benjamin Peterson

概要

C-API の変更は Python 3 の目標には入っていませんでしたが、Python レベルでの変更がたくさんあったので、Python 2 の API を無傷で済ませることはできませんでした。実際、 `int()` と `long()` の統合などは C レベルのほうが目立ちます。この文書では、なくなった互換性と、その対処方法について記述しようと思います。

第1章　条件コンパイル

一部のコードを Python 3 にだけコンパイルするための一番簡単な方法は、 PY_MAJOR_VERSION が 3 以上かどうかチェックすることです。

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

存在しなくなった関数については、条件ブロックの中で同等品にエイリアスすれば良いでしょう。

第2章　オブジェクト API の変更

Python 3 では、似た機能を持つタイプのいくつかを、統合したり、きっちり分けたりしました。

2.1 str/unicode の統合

Python 3 の `str()` タイプは Python 2 の `unicode()` と同じもので、どちらも C 関数は `PyUnicode_*` と呼ばれます。昔の 8 ビット文字列タイプは `bytes()` になり、C 関数は `PyBytes_*` と呼ばれます。Python 2.6 以降には互換性ヘッダ `bytesobject.h` が用意されており、 `PyBytes` 系の名前を `PyString` 系にマップしています。Python 3 との互換性を最大限確保するには、 `PyUnicode` は文字データに、 `PyBytes` はバイナリデータにだけ使うべきです。ほかにも、Python 3 の `PyBytes` と `PyUnicode` は Python 2 の `PyString` と `PyUnicode` とは違って交換不可能だということも重要です。以下の例では `PyUnicode`, `PyString`, `PyBytes` に関するベストプラクティスを見ることができます。

```
#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);
```

(次のページに続く)

(前のページからの続き)

```
    return result;
}
```

2.2 long/int の統合

Python 3 には整数型が `int()` の一つしかありません。これは実際には Python 2 の `long()` タイプと同じもので、 Python 2 で使われていた `int()` は取り除かれました。C-API では `PyInt_*` 関数群が、同等の `PyLong_*` に置き換えられています。

第3章　モジュールの初期化と状態情報

Python 3 には、改良された拡張モジュール初期化システムがあります (**PEP 3121** 参照)。モジュールの状態はグローバル変数に持つのではなく、インタプリタ固有の構造体に持つべきだということになったのです。Pythonk 2 と Python 3 のどちらでも動くモジュールを作るのにはコツが要ります。次の簡単な例で、その方法を実演してみます。

```
#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,
```

(次のページに続く)

(前のページからの続き)

```
        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
}
```

第4章　CObject の Capsule への変更

Python 3.1 と 2.7 に `Capsule` オブジェクトが導入されました。これは `CObject` のかわりになるものです。 CObject は便利でしたが、 `CObject` API には問題がありました: 有効な CObject であればそれ以上の識別が不可能なので、合わない CObject を混ぜて使うとインタープリタをクラッシュさせかねなかったこと、そして API の一部が C の未定義な挙動に依存していたことです。 (Capsule の背景について詳しくは bpo-5630 をご覧ください。)

現時点で CObject を使っているのであれば、3.1 以降へ移行するには Capsule へ切り換える必要があります。 `CObject` は 3.1 および 2.7 で非推奨となり、Python 3.2 では完全に削除されたからです。 2.7 と 3.1 以上だけの対応でいいなら、単に `Capsule` へ切り換えるだけです。 しかし Python 3.0 や、あるいは 2.7 未満のバージョンに対応しなければならない場合は、CObject と Capsule の両方に対応する必要が出てきます。(ただし Python 3.0 はサポートが終了しているため現場利用には推奨されません。)

そうした問題は、下に例示する `capsulethunk.h` ヘッダファイルで解決できるかもしれません。 必要な作業は、 `Capsule` API でコードを書き、このヘッダファイルを `Python.h` の後でインクルードするだけです。 すると自動的に、Capsule のあるバージョンの Python では Capsule を使い、なければ CObject を使うようになります。

その場合 `capsulethunk.h` は CObject を使って Capsule をシミュレートします。 しかし `CObject` には capsule の ”name” を保持する余地がありません。 よって、 `capsulethunk.h` でシミュレートされた `Capsule` オブジェクトの動作は、本物の Capsule とは少し異なります。 具体的には:

- `PyCapsule_New()` に渡された name 引数は無視されます。
- `PyCapsule_IsValid()` および `PyCapsule_GetPointer()` に渡された name 引数は無視され、name のエラーチェックも実行されません。
- `PyCapsule_GetName()` は常に NULL を返します。
- `PyCapsule_SetName()` は常に例外を送出して、失敗を返します。(CObject に名前を格納する方法がないので、ここでは `PyCapsule_SetName()` のうるさい失敗は暗黙の失敗より好ましいと考えられました。これが不便な場合は、適切と思うやり方で自由にローカルコピーを修正してください。)

`capsulethunk.h` は Python のソースの中に Doc/includes/capsulethunk.h という名前で入っています。 読者の便宜のため、ここにも引用しておきましょう:

```
#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) \
```

(次のページに続く)

(前のページからの続き)

```
        ? (((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 *
```

(次のページに続く)

(前のページからの続き)

```
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;
}
```

(次のページに続く)

(前のページからの続き)

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

#endif /* __CAPSULETHUNK_H */
```

第5章　その他のオプション

新規に拡張モジュールを書こうと思っているのであれば、Cython を検討してみても良いでしょう。これは Python 風の言語を C に翻訳してくれるもので、出力される拡張モジュールは Python 3 と Python 2 に両方対応しています。

索引