
Portage des modules d'extension vers Python 3

Version 3.6.11

Guido van Rossum
and the Python development team

juin 29, 2020

Python Software Foundation
Email : docs@python.org

Table des matières

1	Compilation conditionnelle	2
2	Modifications apportées aux API des objets	2
2.1	Unification de <i>str</i> et <i>unicode</i>	2
2.2	Unification de <i>long</i> et <i>int</i>	3
3	Initialisation et état du module	3
4	CObject remplacé par Capsule	5
5	Autres options	8
	Index	9

auteur Benjamin Peterson

Résumé

Changer l'API C n'était pas l'un des objectifs de Python 3, cependant les nombreux changements au niveau Python ont rendu impossible de garder l'API de Python 2 comme elle était. Certains changements tels que l'unification de `int()` et `long()` sont plus apparents au niveau C. Ce document s'efforce de documenter les incompatibilités et la façon dont elles peuvent être contournées.

1 Compilation conditionnelle

La façon la plus simple de compiler seulement une section de code pour Python 3 est de vérifier si `PY_MAJOR_VERSION` est supérieur ou égal à 3. :

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

Les fonctions manquantes dans l'API peuvent être remplacées par des alias à leurs équivalents dans des blocs conditionnels.

2 Modifications apportées aux API des objets

Python 3 a fusionné certains types avec des fonctions identiques tout en séparant de façon propre, d'autres.

2.1 Unification de *str* et *unicode*

Le type `str()` de Python 3 est l'équivalent de `unicode()` sous Python 2; Les fonctions C sont appelées `PyUnicode_*` pour les deux versions. L'ancien type de chaîne de caractères de 8 bits est devenue `bytes()`, avec des fonctions C nommées `PyBytes_*`. Python 2.6 et toutes les versions supérieures fournissent un en-tête de compatibilité, `bytesobject.h`, faisant correspondre les noms `PyBytes` aux `PyString`. Pour une meilleure compatibilité avec Python 3, `PyUnicode` doit être utilisé seulement pour des données textuelles et `PyBytes` pour des données binaires. Il est important de noter que `PyBytes` et `PyUnicode` en Python 3 ne sont pas remplaçables contrairement à `PyString` et `PyUnicode` dans Python 2. L'exemple suivant montre l'utilisation optimale de `PyUnicode`, `PyString`, et `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;
```

(suite sur la page suivante)

```

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

```

2.2 Unification de *long* et *int*

Python 3 n'a qu'un type d'entier, `int()`. Mais il correspond au type `long()` de Python 2 — le type `int()` utilisé dans Python 2 a été supprimé. Dans l'API C, les fonctions `PyInt_*` sont remplacées par leurs équivalents `PyLong_*`.

3 Initialisation et état du module

Python 3 a remanié son système d'initialisation des modules d'extension (Voir [PEP 3121](#)). Au lieu de stocker les états de module dans les variables globales, les états doivent être stockés dans une structure spécifique à l'interpréteur. Créer des modules qui ont un fonctionnement correct en Python 2 et Python 3 est délicat. L'exemple suivant montre comment.

```

#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 remplacé par Capsule

L'objet Capsule a été introduit dans Python 3.1 et 2.7 pour remplacer CObject. Le type CObject était utile, mais son API posait des soucis : elle ne permettait pas la distinction entre les objets C valides, ce qui permettait aux objets C assortis incorrectement de planter l'interpréteur, et certaines des API s'appuyaient sur un comportement indéfini en C. (Pour plus de détails sur la logique de Capsules, veuillez consulter [bpo-5630](#)).

Si vous utilisez actuellement CObjects et que vous voulez migrer vers la version 3.1 ou plus récente, vous devrez passer à Capsules. CObject est déprécié dans 3.1 et 2.7 et est supprimé dans Python 3.2. Si vous ne gérez que les versions 2.7, ou 3.1 et supérieures, vous pouvez simplement passer à Capsule. Si vous avez besoin de gérer Python 3.0, ou des versions de Python antérieures à 2.7, vous devez gérer CObjects et Capsules. (Notez que Python 3.0 n'est plus maintenu, et qu'il n'est pas recommandé pour une utilisation en production).

L'exemple suivant d'en-tête de fichier capsulethunk.h peut résoudre le problème. Il suffit d'écrire votre code dans l'API Capsule et d'inclure ce fichier d'en-tête après Python.h. Votre code utilisera automatiquement Capsules dans les versions de Python avec Capsules, et passera à CObjects lorsque les Capsules ne sont pas disponibles.

capsulethunk.h reproduit le fonctionnement de Capsules en utilisant CObjects. Cependant, CObject ne permet pas de stocker le « nom » de la capsule. Les objets simulés Capsule créés par capsulethunk.h se comportent légèrement différemment des véritables Capsules. Ainsi :

- Le paramètre *name* passé à PyCapsule_New() est ignoré.
- Le paramètre *name* passé à PyCapsule_IsValid() et PyCapsule_GetPointer() est ignoré et il n'y a pas de vérification d'erreur du nom.
- PyCapsule_GetName() renvoie toujours un NULL.
- PyCapsule_SetName() lève toujours une exception et renvoie un échec. Note : Puisqu'il n'y a aucun moyen de stocker un nom dans un CObject, l'échec verbeux de PyCapsule_SetName() a été jugé préférable à un échec non-verbeux dans ce cas. Si cela ne vous convenait pas, vous pouvez modifier votre copie locale selon vos besoins.

Vous pouvez trouver capsulethunk.h dans la distribution source de Python comme [Doc/includes/capsulethunk.h](#). Nous l'incluons ici pour votre confort :

```
#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))
```

(suite sur la page suivante)

```

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

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

/* Don't call PyObject_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_MALLLOC(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 Autres options

Si vous écrivez un nouveau module d'extension, vous pouvez envisager d'utiliser [Cython](#). Il traduit un langage de type Python en C. Les modules d'extension qu'il crée sont compatibles avec Python 3 et Python 2.

Index

P

Python Enhancement Proposals

PEP 3121, [3](#)