

---

# **Extending and Embedding Python**

***Version 3.9.5***

**Guido van Rossum  
and the Python development team**

**juin 27, 2021**

**Python Software Foundation  
Email : [docs@python.org](mailto:docs@python.org)**



---

## Table des matières

---

<b>1</b>	<b>Les outils tiers recommandés</b>	<b>3</b>
<b>2</b>	<b>Création d'extensions sans outils tiers</b>	<b>5</b>
2.1	Étendre Python en C ou C++	5
2.1.1	Un exemple simple	6
2.1.2	Intermezzo : Les erreurs et les exceptions	7
2.1.3	Retour vers l'exemple	9
2.1.4	La fonction d'initialisation et le tableau des méthodes du module	10
2.1.5	Compilation et liaison	12
2.1.6	Appeler des fonctions Python en C	12
2.1.7	Extraire des paramètres dans des fonctions d'extension	14
2.1.8	Paramètres nommés pour des fonctions d'extension	15
2.1.9	Créer des valeurs arbitraires	17
2.1.10	Compteurs de références	17
2.1.11	Écrire des extensions en C++	21
2.1.12	Fournir une API en langage C pour un module d'extension	21
2.2	Defining Extension Types : Tutorial	25
2.2.1	The Basics	25
2.2.2	Adding data and methods to the Basic example	29
2.2.3	Providing finer control over data attributes	36
2.2.4	Supporting cyclic garbage collection	41
2.2.5	Subclassing other types	46
2.3	Définir les types d'extension : divers sujets	49
2.3.1	Finalisation et de-allocation	51
2.3.2	Présentation de l'objet	52
2.3.3	Gestion des attributs	53
2.3.4	Comparaison des objets	55
2.3.5	Support pour le protocole abstrait	56
2.3.6	Prise en charge de la référence faible	58
2.3.7	Plus de suggestions	59
2.4	Construire des extensions C et C++	59
2.4.1	Construire les extensions C et C++ avec <i>distutils</i>	60
2.4.2	Distribuer vos modules d'extension	61
2.5	Construire des extensions C et C++ sur Windows	61
2.5.1	Une approche "recette de cuisine"	62
2.5.2	Différences entre Unix et Windows	62

2.5.3	Utiliser les DLL en pratique . . . . .	63
<b>3</b>	<b>Intégrer l'interpréteur CPython dans une plus grande application</b>	<b>65</b>
3.1	Intégrer Python dans une autre application . . . . .	65
3.1.1	Intégration de très haut niveau . . . . .	66
3.1.2	Au-delà de l'intégration de haut niveau : survol . . . . .	66
3.1.3	Intégration pure . . . . .	67
3.1.4	Étendre un Python intégré . . . . .	69
3.1.5	Intégrer Python dans du C++ . . . . .	70
3.1.6	Compiler et Lier en environnement Unix ou similaire . . . . .	70
<b>A</b>	<b>Glossaire</b>	<b>71</b>
<b>B</b>	<b>À propos de ces documents</b>	<b>85</b>
B.1	Contributeurs de la documentation Python . . . . .	85
<b>C</b>	<b>Histoire et licence</b>	<b>87</b>
C.1	Histoire du logiciel . . . . .	87
C.2	Conditions générales pour accéder à, ou utiliser, Python . . . . .	88
C.2.1	PSF LICENSE AGREEMENT FOR PYTHON 3.9.5 . . . . .	88
C.2.2	LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0 . . . . .	89
C.2.3	LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1 . . . . .	90
C.2.4	LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2 . . . . .	91
C.2.5	LICENCE BSD ZERO-CLAUSE POUR LE CODE DANS LA DOCUMENTATION DE PYTHON 3.9.5 . . . . .	91
C.3	Licences et remerciements pour les logiciels tiers . . . . .	92
C.3.1	Mersenne twister . . . . .	92
C.3.2	Interfaces de connexion ( <i>sockets</i> ) . . . . .	93
C.3.3	Interfaces de connexion asynchrones . . . . .	93
C.3.4	Gestion de témoin ( <i>cookie</i> ) . . . . .	94
C.3.5	Traçage d'exécution . . . . .	94
C.3.6	Les fonctions UUencode et UUdecode . . . . .	95
C.3.7	Appel de procédures distantes en XML ( <i>RPC</i> , pour <i>Remote Procedure Call</i> ) . . . . .	95
C.3.8	test_epoll . . . . .	96
C.3.9	Select kqueue . . . . .	96
C.3.10	SipHash24 . . . . .	97
C.3.11	<i>strtod</i> et <i>dtoa</i> . . . . .	97
C.3.12	OpenSSL . . . . .	98
C.3.13	expat . . . . .	100
C.3.14	libffi . . . . .	101
C.3.15	zlib . . . . .	101
C.3.16	cfuhash . . . . .	102
C.3.17	libmpdec . . . . .	102
C.3.18	Ensemble de tests C14N du W3C . . . . .	103
<b>D</b>	<b>Copyright</b>	<b>105</b>
	<b>Index</b>	<b>107</b>

Ce document décrit comment écrire des modules en C ou C++ pour étendre l'interpréteur Python à de nouveaux modules. En plus de définir de nouvelles fonctions, ces modules peuvent définir de nouveaux types d'objets ainsi que leur méthodes. Ce document explique aussi comment intégrer l'interpréteur Python dans une autre application, pour être utilisé comme langage d'extension. Enfin, ce document montre comment compiler et lier les modules d'extension pour qu'ils puissent être chargés dynamiquement (à l'exécution) dans l'interpréteur, si le système d'exploitation sous-jacent supporte cette fonctionnalité.

Ce document présuppose que vous avez des connaissances de base sur Python. Pour une introduction informelle du langage, voyez [tutorial-index](#). [reference-index](#) donne une définition plus formelle du langage. [library-index](#) documente les objets types, fonctions et modules existants (tous intégrés et écrits en Python) qui donnent au langage sa large gamme d'applications.

Pour une description dans sa totalité de l'API Python/C, voir [c-api-index](#).



# CHAPITRE 1

---

## Les outils tiers recommandés

---

Ce guide ne couvre que les outils basiques permettant de créer des extensions fournies dans cette version de CPython. Les outils tiers tels que [Cython](#), [cffi](#), [SWIG](#) et [Numba](#) offrent des approches plus simples et plus élaborées pour créer des extensions C et C++ pour Python.

**Voir aussi :**

**Guide d'utilisation de l'empaquetage Python : Extensions binaires** Le guide d'utilisation de l'empaquetage Python ne couvre pas uniquement quelques outils disponibles qui simplifient la création d'extensions binaires, mais aborde aussi les différentes raisons pour lesquelles créer un module d'extension peut être souhaitable d'entrée.





---

## Création d'extensions sans outils tiers

---

Cette section du guide couvre la création d'extensions C et C++ sans l'utilisation d'outils tiers. Cette section est destinée aux créateurs de ces outils, plus que d'être une méthode recommandée pour créer votre propre extension C.

### 2.1 Étendre Python en C ou C++

Il est relativement facile d'ajouter de nouveaux modules à Python, si vous savez programmer en C. Ces *<modules d'extension> extension modules* permettent deux choses qui ne sont pas possibles directement en Python : Elles peuvent définir de nouveaux types natifs, et peuvent appeler des fonctions de bibliothèques C ou faire des appels systèmes.

Pour gérer les extensions, l'API Python (*Application Programmer Interface*) définit un ensemble de fonctions, macros et variables qui donnent accès à la plupart des aspects du système d'exécution de Python. L'API Python est incorporée dans un fichier source C en incluant l'en-tête `"Python.h"`.

La compilation d'un module d'extension dépend de l'usage prévu et de la configuration du système, plus de détails peuvent être trouvés dans les chapitres suivants.

---

**Note :** L'interface d'extension C est spécifique à *CPython*, et les modules d'extension ne fonctionnent pas sur les autres implémentations de Python. Dans de nombreux cas, il est possible d'éviter la rédaction des extensions en C et ainsi préserver la portabilité vers d'autres implémentations. Par exemple, si vous devez appeler une fonction de la bibliothèque C ou faire un appel système, vous devriez envisager d'utiliser le module `ctypes` ou d'utiliser la bibliothèque *\*cffi\** plutôt que d'écrire du code C sur mesure. Ces modules vous permettent d'écrire du code Python s'interfaçant avec le code C et sont plus portables entre les implémentations de Python que l'écriture et la compilation d'une d'extension C.

---

## 2.1.1 Un exemple simple

Créons un module d'extension appelé `spam` (la nourriture préférée de fans des *Monty Python* ...) et disons que nous voulons créer une interface Python à la fonction de la bibliothèque C `system()`<sup>1</sup>. Cette fonction prend une chaîne de caractères à terminaison nulle comme argument et renvoie un entier. Nous voulons que cette fonction soit callable à partir de Python comme suit :

```
>>> import spam
>>> status = spam.system("ls -l")
```

Commencez par créer un fichier `spammodule.c`. (Historiquement, si un module se nomme `spam`, le fichier C contenant son implémentation est appelé `spammodule.c`. Si le nom du module est très long, comme `spammify`, le nom du module peut être juste `spammify.c`.)

Les deux premières lignes de notre fichier peuvent être :

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

qui récupère l'API Python (vous pouvez ajouter un commentaire décrivant le but du module et un avis de droit d'auteur si vous le souhaitez).

**Note :** Python pouvant définir certaines définitions pré-processeur qui affectent les têtes standard sur certains systèmes, vous devez inclure `Python.h` avant les en-têtes standards.

Il est recommandé de toujours définir `PY_SSIZE_T_CLEAN` avant d'inclure `Python.h`. Lisez [Extraire des paramètres dans des fonctions d'extension](#) pour avoir une description de cette macro.

Tous les symboles exposés par `Python.h` sont préfixés de `Py` ou `PY`, sauf ceux qui sont définis dans les en-têtes standard. Pour le confort, et comme ils sont largement utilisés par l'interpréteur Python, "`Python.h`" inclut lui-même quelques d'en-têtes standard : `<stdio.h>`, `<string.h>`, `<errno.h>` et `<stdlib.h>`. Si ce dernier n'existe pas sur votre système, il déclare les fonctions `malloc()`, `free()` et `realloc()` directement.

La prochaine chose que nous ajoutons à notre fichier de module est la fonction C qui sera appelée lorsque l'expression Python `spam.system(chaîne)` sera évaluée (nous verrons bientôt comment elle finit par être appelée) :

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

Il y a une correspondance directe de la liste des arguments en Python (par exemple, l'expression `"ls -l"`) aux arguments passés à la fonction C. La fonction C a toujours deux arguments, appelés par convention *self* et *args*.

Pour les fonctions au niveau du module, l'argument *self* pointe sur l'objet module, pour une méthode, il pointe sur l'instance de l'objet.

L'argument *args* sera un pointeur vers un *n*-uplet Python contenant les arguments. Chaque élément du *n*-uplet correspond à un argument dans la liste des arguments de l'appel. Les arguments sont des objets Python, afin d'en faire quelque chose

1. Une interface pour cette fonction existe déjà dans le module standard `os`, elle a été choisie comme un exemple simple et direct.

dans notre fonction C, nous devons les convertir en valeurs C. La fonction `PyArg_ParseTuple()` de l'API Python vérifie les types des arguments et les convertit en valeurs C. Elle utilise un modèle sous forme de chaîne pour déterminer les types requis des arguments ainsi que les types de variables C dans lequel stocker les valeurs converties. Nous en verront plus, plus tard.

`PyArg_ParseTuple()` renvoie vrai (pas zéro) si tous les arguments ont le bon type et que ses composants ont été stockés dans les variables dont les adresses ont été données en entrée. Il renvoie faux (zéro) si une liste d'arguments invalide a été passée. Dans ce dernier cas, elle lève également une exception appropriée de sorte que la fonction d'appel puisse renvoyer `NULL` immédiatement (comme nous l'avons vu dans l'exemple).

## 2.1.2 Intermezzo : Les erreurs et les exceptions

Une convention primordiale imprégnant tout l'interpréteur Python est : quand une fonction échoue, elle devrait laisser une exception et renvoyer une valeur d'erreur (typiquement un pointeur `NULL`). Dans l'interpréteur, les exceptions sont stockées dans une variable globale statique, si cette variable est `NULL`, aucune exception n'a eu lieu. Une seconde variable globale stocke la "valeur associée" à l'exception (le deuxième argument de `raise`). Une troisième variable contient la trace de la pile dans le cas où l'erreur soit survenue dans du code Python. Ces trois variables sont les équivalents C du résultat de `sys.exc_info()` en Python (voir la section sur le module `sys` dans *The Python Library Reference*). Il est important de les connaître pour comprendre comment les erreurs sont propagées.

L'API Python définit un certain nombre de fonctions pour créer différents types d'exceptions.

La plus courante est `PyErr_SetString()`. Ses arguments sont un objet exception et une chaîne C. L'objet exception est généralement un objet prédéfini comme `PyExc_ZeroDivisionError`. La chaîne C indique la cause de l'erreur et est convertie en une chaîne Python puis stockée en tant que "valeur associée" à l'exception.

Une autre fonction utile est `PyErr_SetFromErrno()`, qui construit une exception à partir de la valeur de la variable globale `errno`. La fonction la plus générale est `PyErr_SetObject()`, qui prend deux arguments : l'exception et sa valeur associée. Vous ne devez pas appliquer `Py_INCREF()` aux objets transmis à ces fonctions.

Vous pouvez tester de manière non destructive si une exception a été levée avec `PyErr_Occurred()`. Cela renvoie l'objet exception actuel, ou `NULL` si aucune exception n'a eu lieu. Cependant, vous ne devriez pas avoir besoin d'appeler `PyErr_Occurred()` pour voir si une erreur est survenue durant l'appel d'une fonction, puisque vous devriez être en mesure de le déterminer à partir de la valeur renvoyée.

When a function *f* that calls another function *g* detects that the latter fails, *f* should itself return an error value (usually `NULL` or `-1`). It should *not* call one of the `PyErr_*` functions --- one has already been called by *g*. *f*'s caller is then supposed to also return an error indication to *its* caller, again *without* calling `PyErr_*`, and so on --- the most detailed cause of the error was already reported by the function that first detected it. Once the error reaches the Python interpreter's main loop, this aborts the currently executing Python code and tries to find an exception handler specified by the Python programmer.

(Il y a des situations où un module peut effectivement donner un message d'erreur plus détaillé en appelant une autre fonction `PyErr_*`, dans de tels cas, il est tout à fait possible de le faire. Cependant, ce n'est généralement pas nécessaire, et peut amener à perdre des informations sur la cause de l'erreur : la plupart des opérations peuvent échouer pour tout un tas de raisons.)

Pour ignorer une exception qui aurait été émise lors d'un appel de fonction qui aurait échoué, l'exception doit être retirée explicitement en appelant `PyErr_Clear()`. Le seul cas pour lequel du code C devrait appeler `PyErr_Clear()` est lorsqu'il ne veut pas passer l'erreur à l'interpréteur, mais souhaite la gérer lui-même (peut-être en essayant quelque chose d'autre, ou en prétendant que rien n'a mal tourné).

Chaque échec de `malloc()` doit être transformé en une exception, l'appelant direct de `malloc()` (ou `realloc()`) doit appeler `PyErr_NoMemory()` et prendre l'initiative de renvoyer une valeur d'erreur. Toutes les fonctions construisant des objets (tels que `PyLong_FromLong()`) le font déjà, donc cette note ne concerne que ceux qui appellent `malloc()` directement.

Notez également que, à l'exception notable de `PyArg_ParseTuple()` et compagnie, les fonctions qui renvoient leur statut sous forme d'entier donnent généralement une valeur positive ou zéro en cas de succès et `-1` en cas d'échec, comme les appels du système Unix.

Enfin, lorsque vous renvoyez un code d'erreur, n'oubliez pas faire un brin de nettoyage (en appelant `Py_XDECREF()` ou `Py_DECREF()` avec les objets que vous auriez déjà créés) !

Le choix de l'exception à lever vous incombe. Il existe des objets C correspondant à chaque exception Python, tel que `PyExc_ZeroDivisionError`, que vous pouvez utiliser directement. Choisissez judicieusement vos exceptions, typiquement n'utilisez pas `PyExc_TypeError` pour indiquer qu'un fichier n'a pas pu être ouvert (qui devrait probablement être `PyExc_IOError`). Si quelque chose ne va pas avec la liste des arguments, la fonction `PyArg_ParseTuple()` lève habituellement une exception `PyExc_TypeError`. Mais si vous avez un argument dont la valeur doit être dans un intervalle particulier ou qui doit satisfaire d'autres conditions, `PyExc_ValueError` sera plus appropriée.

Vous pouvez également créer une exception spécifique à votre module. Pour cela, déclarez simplement une variable statique au début de votre fichier :

```
static PyObject *SpamError;
```

et initialisez-la dans la fonction d'initialisation de votre module (`PyInit_spam()`) avec un objet exception :

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_XINCREF(SpamError);
    if (PyModule_AddObject(m, "error", SpamError) < 0) {
        Py_XDECREF(SpamError);
        Py_CLEAR(SpamError);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

Notez que le nom de l'exception en Python est `spam.error`. La fonction `PyErr_NewException()` peut créer une classe héritant de `Exception` (à moins qu'une autre classe ne lui soit fournie à la place de `NULL`), voir `bltin-exceptions`.

Notez également que la variable `SpamError` contient une référence à la nouvelle classe créée; ceci est intentionnel ! Comme l'exception peut être retirée du module par un code externe, une référence à la classe est nécessaire pour assurer qu'il ne sera pas rejeté, causant `SpamError` et devenir un pointeur défaillant. Si cela se produirait, le C code qui lève cette exception peut engendrer un *core dump* ou des effets secondaires inattendus.

Nous traiterons de l'utilisation de `PyMODINIT_FUNC` comme un type de renvoi de fonction plus tard dans cette section.

L'exception `spam.error` peut être levée dans votre module d'extension en appelant `PyErr_SetString()` comme montré ci-dessous :

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
```

(suite sur la page suivante)

(suite de la page précédente)

```

const char *command;
int sts;

if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
sts = system(command);
if (sts < 0) {
    PyErr_SetString(SpamError, "System command failed");
    return NULL;
}
return PyLong_FromLong(sts);
}
    
```

### 2.1.3 Retour vers l'exemple

En revenant vers notre fonction exemple, vous devriez maintenant être capable de comprendre cette affirmation :

```

if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
    
```

Elle renvoie NULL (l'indicateur d'erreur pour les fonctions renvoyant des pointeurs d'objet) si une erreur est détectée dans la liste des arguments, se fiant à l'exception définie par `PyArg_ParseTuple()`. Autrement, la valeur chaîne de l'argument a été copiée dans la variable locale `command`. Il s'agit d'une attribution de pointeur et vous n'êtes pas supposés modifier la chaîne vers laquelle il pointe (donc en C Standard, la variable `command` doit être clairement déclarée comme `const char *command`).

La prochaine instruction est un appel à la fonction Unix `system()`, en lui passant la chaîne que nous venons d'obtenir à partir de `PyArg_ParseTuple()` :

```

sts = system(command);
    
```

Notre fonction `spam.system()` doit renvoyer la valeur de `sts` comme un objet Python. Cela est effectué par l'utilisation de la fonction `PyLong_FromLong()`.

```

return PyLong_FromLong(sts);
    
```

Dans ce cas, elle renverra un objet entier. (Oui, même les entiers sont des objets dans le tas en Python !)

Si vous avez une fonction C qui ne renvoie aucun argument utile (une fonction renvoyant `void`), la fonction Python correspondante doit renvoyer `None`. Vous aurez besoin de cette locution pour cela (qui est implémentée par la macro `Py_RETURN_NONE`) :

```

Py_INCREF(Py_None);
return Py_None;
    
```

`Py_None` est la dénomination en C pour l'objet spécial Python `None`. C'est un authentique objet Python plutôt qu'un pointeur NULL, qui signifie qu'une erreur est survenue, dans la plupart des situations, comme nous l'avons vu.

## 2.1.4 La fonction d'initialisation et le tableau des méthodes du module

Nous avons promis de montrer comment `spam_system()` est appelée depuis les programmes Python. D'abord, nous avons besoin d'avoir son nom et son adresse dans un « tableau des méthodes » :

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system", spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}          /* Sentinel */
};
```

Notez la troisième entrée (`METH_VARARGS`). C'est un indicateur du type de convention à utiliser pour la fonction C, à destination de l'interpréteur. Il doit valoir normalement `METH_VARARGS` ou `METH_VARARGS | METH_KEYWORDS`; la valeur 0 indique qu'une variante obsolète de `PyArg_ParseTuple()` est utilisée.

Si seulement `METH_VARARGS` est utilisé, la fonction s'attend à ce que les paramètres Python soient passés comme un *n*-uplet que l'on peut analyser *via* `PyArg_ParseTuple()`; des informations supplémentaires sont fournies plus bas.

Le bit `METH_KEYWORDS` peut être mis à un dans le troisième champ si des arguments par mot-clés doivent être passés à la fonction. Dans ce cas, la fonction C doit accepter un troisième paramètre `PyObject *` qui est un dictionnaire des mots-clés. Utilisez `PyArg_ParseTupleAndKeywords()` pour analyser les arguments d'une telle fonction.

Le tableau des méthodes doit être référencé dans la structure de définition du module :

```
static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam",          /* name of module */
    spam_doc,        /* module documentation, may be NULL */
    -1,              /* size of per-interpreter state of the module,
                     or -1 if the module keeps state in global variables. */
    SpamMethods
};
```

Cette structure, à son tour, doit être transmise à l'interpréteur dans la fonction d'initialisation du module. La fonction d'initialisation doit être nommée `PyInit_name()`, où *nom* est le nom du module, et doit être le seul élément non `static` défini dans le fichier du module :

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModule_Create(&spammodule);
}
```

Notez que `PyMODINIT_FUNC` déclare la fonction comme renvoyant un objet de type `PyObject *`, et déclare également toute déclaration de liaison spéciale requise par la plate-forme, et pour le C++ déclare la fonction comme un `C extern`.

Lorsque le programme Python importe le module `spam` pour la première fois, `PyInit_spam()` est appelé. (Voir ci-dessous pour les commentaires sur l'intégration en Python.) Il appelle `PyModule_Create()`, qui renvoie un objet module, et insère des objets fonction intégrés dans le module nouvellement créé en se basant sur la table (un tableau de structures `PyMethodDef`) trouvée dans la définition du module. `PyModule_Create()` renvoie un pointeur vers l'objet module qu'il crée. Il peut s'interrompre avec une erreur fatale pour certaines erreurs, ou renvoyer `NULL` si le module n'a pas pu être initialisé de manière satisfaisante. La fonction *\*init\** doit renvoyer l'objet module à son appelant, afin qu'il soit ensuite inséré dans `sys.modules`.

Lors de l'intégration de Python, la fonction `PyInit_spam()` n'est pas appelée automatiquement, sauf s'il y a une entrée dans la table `PyImport_Inittab`. Pour ajouter le module à la table d'initialisation, utilisez `PyImport_AppendInittab()`, suivi éventuellement d'une importation du module :

```

int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }

    /* Add a built-in module, before Py_Initialize */
    if (PyImport_AppendInittab("spam", PyInit_spam) == -1) {
        fprintf(stderr, "Error: could not extend in-built modules table\n");
        exit(1);
    }

    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(program);

    /* Initialize the Python interpreter. Required.
       If this step fails, it will be a fatal error. */
    Py_Initialize();

    /* Optionally import the module; alternatively,
       import can be deferred until the embedded script
       imports it. */
    PyObject *pmodule = PyImport_ImportModule("spam");
    if (!pmodule) {
        PyErr_Print();
        fprintf(stderr, "Error: could not import module 'spam'\n");
    }

    ...

    PyMem_RawFree(program);
    return 0;
}

```

---

**Note :** Supprimer des entrées de `sys.modules` ou importer des modules compilés dans plusieurs interpréteurs au sein d'un processus (ou suivre un `fork()` sans l'intervention d'un `exec()`) peut créer des problèmes pour certains modules d'extension. Les auteurs de modules d'extension doivent faire preuve de prudence lorsqu'ils initialisent des structures de données internes.

---

Un exemple de module plus substantiel est inclus dans la distribution des sources Python sous le nom `Modules/xxmodule.c`. Ce fichier peut être utilisé comme modèle ou simplement lu comme exemple.

---

**Note :** Contrairement à notre exemple de `spam`, `xxmodule` utilise une *initialisation multi-phase* (nouveau en Python 3.5), où une structure `PyModuleDef` est renvoyée à partir de `PyInit_spam`, et la création du module est laissée au mécanisme d'importation. Pour plus de détails sur l'initialisation multi-phase, voir [PEP 489](#).

---

## 2.1.5 Compilation et liaison

There are two more things to do before you can use your new extension : compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses ; see the chapters about building extension modules (chapter *Construire des extensions C et C++*) and additional information that pertains only to building on Windows (chapter *Construire des extensions C et C++ sur Windows*) for more information about this.

If you can't use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on Unix : just place your file (`spammodule.c` for example) in the `Modules/` directory of an unpacked source distribution, add a line to the file `Modules/Setup.local` describing your file :

```
spam spammodule.o
```

and rebuild the interpreter by running **make** in the toplevel directory. You can also run **make** in the `Modules/` subdirectory, but then you must first rebuild `Makefile` there by running **'make Makefile'**. (This is necessary each time you change the `Setup` file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance :

```
spam spammodule.o -lX11
```

## 2.1.6 Appeler des fonctions Python en C

So far we have concentrated on making C functions callable from Python. The reverse is also useful : calling Python functions from C. This is especially the case for libraries that support so-called "callback" functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer ; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won't dwell on how to call the Python parser with a particular string as input --- if you're interested, have a look at the implementation of the `-c` command line option in `Modules/main.c` from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to `Py_INCREF()` it!) in a global variable --- or wherever you see fit. For example, the following function might be part of a module definition :

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);          /* Add a reference to new callback */
        Py_XDECREF(my_callback); /* Dispose of previous callback */
        my_callback = temp;       /* Remember new callback */
        /* Boilerplate to return "None" */
    }
}
```

(suite sur la page suivante)



(suite de la page précédente)

```

        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
    
```

Cette fonction doit être déclarée en utilisant le drapeau `METH_VARARGS` ; ceci est décrit dans la section *La fonction d'initialisation et le tableau des méthodes du module*. La fonction `PyArg_ParseTuple()` et ses arguments sont documentés dans la section *Extraire des paramètres dans des fonctions d'extension*.

Les macros `Py_XINCRREF()` et `Py_XDECREF()` incrémentent/décrémentent le compteur des références d'un objet et sont sûres quant à la présence de pointeurs `NULL` (mais notez que *temp* ne sera pas `NULL` dans ce contexte). Plus d'informations à ce sujet dans la section *Compteurs de références*.

Later, when it is time to call the function, you call the C function `PyObject_CallObject()`. This function has two arguments, both pointers to arbitrary Python objects : the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass in `NULL`, or an empty tuple; to call it with one argument, pass a singleton tuple. `Py_BuildValue()` returns a tuple when its format string consists of zero or more format codes between parentheses. For example :

```

int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
    
```

`PyObject_CallObject()` returns a Python object pointer : this is the return value of the Python function. `PyObject_CallObject()` is "reference-count-neutral" with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is `Py_DECREF()`-ed immediately after the `PyObject_CallObject()` call.

The return value of `PyObject_CallObject()` is "new" : either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow `Py_DECREF()` the result, even (especially !) if you are not interested in its value.

Mais avant de le faire, il est important de vérifier que la valeur renvoyée n'est pas `NULL`. Si c'est le cas, la fonction Python s'est terminée par la levée d'une exception. Si le code C qui a appelé `PyObject_CallObject()` est appelé depuis Python, il devrait maintenant renvoyer une indication d'erreur à son appelant Python, afin que l'interpréteur puisse afficher la pile d'appels, ou que le code Python appelant puisse gérer l'exception. Si cela n'est pas possible ou souhaitable, l'exception doit être effacée en appelant `PyErr_Clear()`. Par exemple :

```

if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
    
```

Selon l'interface souhaitée pour la fonction de rappel Python, vous devrez peut-être aussi fournir une liste d'arguments à `PyObject_CallObject()`. Dans certains cas, la liste d'arguments est également fournie par le programme Python, par l'intermédiaire de la même interface qui a spécifié la fonction de rappel. Elle peut alors être sauvegardée et utilisée de la même manière que l'objet fonction. Dans d'autres cas, vous pouvez avoir à construire un nouveau n-uplet à passer comme liste d'arguments. La façon la plus simple de faire cela est d'appeler `Py_BuildValue()`. Par exemple, si vous voulez passer un code d'événement intégral, vous pouvez utiliser le code suivant :

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

Note the placement of `Py_DECREF(arglist)` immediately after the call, before the error check! Also note that strictly speaking this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.

Vous pouvez également appeler une fonction avec des arguments nommés en utilisant `PyObject_Call()`, qui accepte les arguments et les arguments nommés. Comme dans l'exemple ci-dessus, nous utilisons `Py_BuildValue()` pour construire le dictionnaire. :

```
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

### 2.1.7 Extraire des paramètres dans des fonctions d'extension

La fonction `PyArg_ParseTuple()` est déclarée ainsi :

```
int PyArg_ParseTuple(PyObject *arg, const char *format, ...);
```

The *arg* argument must be a tuple object containing an argument list passed from Python to a C function. The *format* argument must be a format string, whose syntax is explained in arg-parsing in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while `PyArg_ParseTuple()` checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call : if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

Notez que n'importe quelles références sur un objet Python qui sont données à l'appelant sont des références *empruntées*; ne décrémente pas leur compteur de références !

Quelques exemples d'appels :

```
#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>
```

```
int ok;
int i, j;
long k, l;
const char *s;
Py_ssize_t size;
```

(suite sur la page suivante)

(suite de la page précédente)

```
ok = PyArg_ParseTuple(args, ""); /* No arguments */
/* Python call: f() */
```

```
ok = PyArg_ParseTuple(args, "s", &s); /* A string */
/* Possible Python call: f('whoops!') */
```

```
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
/* Possible Python call: f(1, 2, 'three') */
```

```
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
/* A pair of ints and a string, whose size is also returned */
/* Possible Python call: f((1, 2), 'three') */
```

```
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* A string, and optionally another string and an integer */
    /* Possible Python calls:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}
```

```
{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
        &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
       f(((0, 0), (400, 300)), (10, 10)) */
}
```

```
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}
```

## 2.1.8 Paramètres nommés pour des fonctions d'extension

La fonction `PyArg_ParseTupleAndKeywords()` est déclarée ainsi :

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                const char *format, char *kwlist[], ...);
```

Les paramètres *arg* et *format* sont identiques à ceux de la fonction `PyArg_ParseTuple()`. Le paramètre *kwdict* est le dictionnaire de mots-clés reçu comme troisième paramètre du *runtime* Python. Le paramètre *kwlist* est une liste de chaînes de caractères terminée par `NULL` qui identifie les paramètres; les noms sont mis en correspondance, de gauche à droite, avec les informations de type de *format*. En cas de succès du processus, `PyArg_ParseTupleAndKeywords()` renvoie vrai, sinon il renvoie faux et lève une exception appropriée.

**Note :** Les n-uplets imbriqués ne peuvent pas être traités lorsqu'on utilise des arguments de type mot-clé ! Ceux-ci doivent apparaître dans *kwlist*, dans le cas contraire une exception `TypeError` est levée.

Voici un exemple de module qui utilise des mots-clés, basé sur un exemple de *Geoff Philbrick* ([philbrick@hks.com](mailto:philbrick@hks.com)) :

```
#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    const char *state = "a stiff";
    const char *action = "voom";
    const char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                     &voltage, &state, &action, &type))
        return NULL;

    printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

    Py_RETURN_NONE;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)(void(*) (void))keywdarg_parrot, METH_VARARGS | METH_
↪KEYWORDS,
    "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

static struct PyModuleDef keywdargmodule = {
    PyModuleDef_HEAD_INIT,
    "keywdarg",
    NULL,
    -1,
    keywdarg_methods
};

PyMODINIT_FUNC
PyInit_keywdarg(void)
{
    return PyModule_Create(&keywdargmodule);
}
```

## 2.1.9 Créer des valeurs arbitraires

Cette fonction est le complément de `PyArg_ParseTuple()`. Elle est déclarée comme suit :

```
PyObject *Py_BuildValue(const char *format, ...);
```

Il reconnaît un ensemble d'unités de format similaires à celles reconnues par `PyArg_ParseTuple()`, mais les arguments (qui sont les données en entrée de fonction, et non de la sortie) ne doivent pas être des pointeurs, mais juste des valeurs. Il renvoie un nouvel objet Python, adapté pour être renvoyé par une fonction C appelée depuis Python.

One difference with `PyArg_ParseTuple()` : while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), `Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

Exemples (à gauche l'appel, à droite la valeur résultante, en Python) :

<code>Py_BuildValue("")</code>	<code>None</code>
<code>Py_BuildValue("i", 123)</code>	<code>123</code>
<code>Py_BuildValue("iii", 123, 456, 789)</code>	<code>(123, 456, 789)</code>
<code>Py_BuildValue("s", "hello")</code>	<code>'hello'</code>
<code>Py_BuildValue("y", "hello")</code>	<code>b'hello'</code>
<code>Py_BuildValue("ss", "hello", "world")</code>	<code>('hello', 'world')</code>
<code>Py_BuildValue("s#", "hello", 4)</code>	<code>'hell'</code>
<code>Py_BuildValue("y#", "hello", 4)</code>	<code>b'hell'</code>
<code>Py_BuildValue("()")</code>	<code>()</code>
<code>Py_BuildValue("(i)", 123)</code>	<code>(123,)</code>
<code>Py_BuildValue("(ii)", 123, 456)</code>	<code>(123, 456)</code>
<code>Py_BuildValue("(i,i)", 123, 456)</code>	<code>(123, 456)</code>
<code>Py_BuildValue("[i,i]", 123, 456)</code>	<code>[123, 456]</code>
<code>Py_BuildValue("{s:i,s:i}",</code>	
<code>    "abc", 123, "def", 456)</code>	<code>{'abc': 123, 'def': 456}</code>
<code>Py_BuildValue("((ii)(ii))(ii)",</code>	
<code>    1, 2, 3, 4, 5, 6)</code>	<code>(( (1, 2), (3, 4) ), (5, 6))</code>

## 2.1.10 Compteurs de références

Dans les langages comme le C ou le C++, le développeur est responsable de l'allocation dynamique et de la dés-allocation de la mémoire sur le tas. En C, cela se fait à l'aide des fonctions `malloc()` et `free()`. En C++, les opérateurs `new` et `delete` sont utilisés avec essentiellement la même signification et nous limiterons la discussion suivante au cas du C.

Every block of memory allocated with `malloc()` should eventually be returned to the pool of available memory by exactly one call to `free()`. It is important to call `free()` at the right time. If a block's address is forgotten but `free()` is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a *memory leak*. On the other hand, if a program calls `free()` for a block and then continues to use the block, it creates a conflict with re-use of the block through another `malloc()` call. This is called *using freed memory*. It has the same bad consequences as referencing uninitialized data --- core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It's easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time : the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses

the leaking function frequently. Therefore, it's important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Comme Python fait un usage intensif de `malloc()` et de `free()`, il a besoin d'une stratégie pour éviter les fuites de mémoire ainsi que l'utilisation de la mémoire libérée. La méthode choisie est appelée *reference counting*. Le principe est simple : chaque objet contient un compteur, qui est incrémenté lorsqu'une référence à l'objet est stockée quelque part, et qui est décrémenté lorsqu'une référence à celui-ci est supprimée. Lorsque le compteur atteint zéro, la dernière référence à l'objet a été supprimée et l'objet est libéré.

Une stratégie alternative est appelée *automatic garbage collection* (ramasse-miettes). Parfois, le comptage des références est également appelé stratégie de ramasse-miettes, d'où l'utilisation du terme "automatique" pour distinguer les deux. Le grand avantage du ramasse-miettes est que l'utilisateur n'a pas besoin d'appeler `free()` explicitement. (Un autre avantage important est l'amélioration de la vitesse ou de l'utilisation de la mémoire, ce n'est cependant pas un fait avéré). L'inconvénient est que pour C, il n'y a pas de ramasse-miettes portable proprement-dit, alors que le comptage des références peut être implémenté de façon portable (tant que les fonctions `malloc()` et `free()` soient disponibles, ce que la norme C garantit). Peut-être qu'un jour un ramasse-miettes suffisamment portable sera disponible pour C. D'ici là, nous devons utiliser les compteurs des références.

Bien que Python utilise l'implémentation traditionnelle de comptage de référence, il contient également un détecteur de cycles qui fonctionne pour détecter les cycles de référence. Cela permet aux applications d'empêcher la création de références circulaires directes ou indirectes ; ceci sont les faiblesses du ramasse-miettes mis en œuvre en utilisant uniquement le comptage de référence. Les cycles de référence sont constitués d'objets qui contiennent des références (éventuellement indirectes) à eux-mêmes, de sorte que chaque objet du cycle a un comptage de référence qui n'est pas nul. Les implémentations typiques de comptage de référence ne sont pas capables de récupérer la mémoire appartenant à des objets dans un cycle de référence, ou référencés à partir des objets dans le cycle, même s'il n'y a pas d'autres références au cycle lui-même.

The cycle detector is able to detect garbage cycles and can reclaim them. The `gc` module exposes a way to run the detector (the `collect()` function), as well as configuration interfaces and the ability to disable the detector at runtime. The cycle detector is considered an optional component ; though it is included by default, it can be disabled at build time using the `--without-cycle-gc` option to the **configure** script on Unix platforms (including Mac OS X). If the cycle detector is disabled in this way, the `gc` module will not be available.

### Comptage de références en Python

Il existe deux macros, `Py_INCREF(x)` et `Py_DECREF(x)`, qui gèrent l'incrémement et la décrémement du comptage de référence. `Py_DECREF()` libère également l'objet lorsque le comptage atteint zéro. Pour plus de flexibilité, il n'appelle pas `free()` directement — plutôt, il fait un appel à travers un pointeur de fonction dans l'objet *type objet* de l'objet. À cette fin (et pour d'autres), chaque objet contient également un pointeur vers son objet type.

La grande question demeure maintenant : quand utiliser `Py_INCREF(x)` et `Py_DECREF(x)` ? Commençons par définir quelques termes. Personne ne possède un objet, mais vous pouvez en *avoir une référence*. Le comptage de références d'un objet est maintenant défini comme étant le nombre de références à cet objet. Le propriétaire d'une référence est responsable d'appeler `Py_DECREF()` lorsque la référence n'est plus nécessaire. La propriété d'une référence peut être transférée. Il y a trois façons de disposer d'une référence : la transmettre, la stocker, ou appeler `Py_DECREF()`. Oublier de se débarrasser d'une référence crée une fuite de mémoire.

It is also possible to *borrow*<sup>2</sup> a reference to an object. The borrower of a reference should not call `Py_DECREF()`. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely<sup>3</sup>.

L'avantage d'emprunter, plutôt qu'être propriétaire d'une référence est que vous n'avez pas à vous soucier de disposer de la référence sur tous les chemins possibles dans le code — en d'autres termes, avec une référence empruntée, vous ne courez pas le risque de fuites lors d'une sortie prématurée. L'inconvénient de l'emprunt par rapport à la possession est

---

2. L'expression « emprunter une référence » n'est pas tout à fait correcte, car le propriétaire a toujours une copie de la référence.

3. Vérifier que le comptage de référence est d'au moins 1 **ne fonctionne pas**, le compte de référence lui-même pourrait être en mémoire libérée et peut donc être réutilisé pour un autre objet !

qu'il existe certaines situations subtiles où, dans un code apparemment correct, une référence empruntée peut être utilisée après que le propriétaire auquel elle a été empruntée l'a en fait éliminée.

A borrowed reference can be changed into an owned reference by calling `Py_INCREF()`. This does not affect the status of the owner from which the reference was borrowed --- it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).

## Règles concernant la propriété de références

Chaque fois qu'une référence d'objet est passée à l'intérieur ou à l'extérieur d'une fonction, elle fait partie de la spécification de l'interface de la fonction, peu importe que la propriété soit transférée avec la référence ou non.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, such as `PyLong_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, `PyLong_FromLong()` maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance `PyObject_GetAttrString()`. The picture is less clear, here, however, since a few common routines are exceptions: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, and `PyDict_GetItemString()` all return references that you borrow from the tuple, list or dictionary.

The function `PyImport_AddModule()` also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in `sys.modules`.

When you pass an object reference into another function, in general, the function borrows the reference from you --- if it needs to store it, it will use `Py_INCREF()` to become an independent owner. There are exactly two important exceptions to this rule: `PyTuple_SetItem()` and `PyList_SetItem()`. These functions take over ownership of the item passed to them --- even if they fail! (Note that `PyDict_SetItem()` and friends don't take over ownership --- they are "normal.")

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference's lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling `Py_INCREF()`.

The object reference returned from a C function that is called from Python must be an owned reference --- ownership is transferred from the function to its caller.

## Terrain dangereux

Il existe quelques situations où l'utilisation apparemment inoffensive d'une référence empruntée peut entraîner des problèmes. Tous ces problèmes sont en lien avec des invocations implicites de l'interpréteur, et peuvent amener le propriétaire d'une référence à s'en défaire.

Le premier cas, et le plus important à connaître, est celui de l'application de `Py_DECREF()` à un objet non relié, tout en empruntant une référence à un élément de liste. Par exemple :

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

Cette fonction emprunte d'abord une référence à `list[0]`, puis remplace `list[1]` par la valeur 0, et enfin affiche la référence empruntée. Ça a l'air inoffensif, n'est-ce pas ? Mais ce n'est pas le cas !

Suivons le flux de contrôle dans `PyList_SetItem()`. La liste possède des références à tous ses éléments, donc quand l'élément 1 est remplacé, elle doit se débarrasser de l'élément 1 original. Supposons maintenant que l'élément 1 original était une instance d'une classe définie par l'utilisateur, et supposons en outre que la classe définisse une méthode `__del__()`. Si l'instance de cette classe a un nombre des références de 1, sa destruction appellera sa méthode `__del__()`.

Comme elle est écrite en Python, la méthode `__del__()` peut exécuter du code Python arbitraire. Pourrait-elle faire quelque chose pour invalider la référence à `item` dans `bug()` ? Bien sûr ! En supposant que la liste passée dans `bug()` est accessible à la méthode `__del__()`, elle pourrait exécuter une instruction à l'effet de `del list[0]`, et en supposant que ce soit la dernière référence à cet objet, elle libérerait la mémoire qui lui est associée, invalidant ainsi `item`.

The solution, once you know the source of the problem, is easy : temporarily increment the reference count. The correct version of the function reads :

```
void
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his `__del__()` methods would fail...

Le deuxième cas de problèmes liés à une référence empruntée est une variante impliquant des fils de discussion. Normalement, plusieurs threads dans l'interpréteur Python ne peuvent pas se gêner mutuellement, car il existe un verrou global protégeant tout l'espace objet de Python. Cependant, il est possible de libérer temporairement ce verrou en utilisant la macro `Py_BEGIN_ALLOW_THREADS`, et de le ré-acquérir en utilisant `Py_END_ALLOW_THREADS`. Ceci est un procédé courant pour bloquer les appels d'entrées/sorties, afin de permettre aux autres threads d'utiliser le processeur en attendant que les E/S soient terminées. Évidemment, la fonction suivante a le même problème que la précédente :

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```



## Pointeurs NULL

En général, les fonctions qui prennent des références d'objets comme arguments ne sont pas conçues pour recevoir des pointeurs NULL, et si vous en donnez comme arguments, elles causeront une erreur de segmentation (ou provoqueront des *core dump* ultérieurs). Les fonctions qui renvoient des références d'objets renvoient généralement NULL uniquement pour indiquer qu'une exception s'est produite. La raison pour laquelle les arguments NULL ne sont pas testés est que les fonctions passent souvent les objets qu'elles reçoivent à d'autres fonctions, si chaque fonction devait tester pour NULL, il y aurait beaucoup de tests redondants et le code s'exécuterait plus lentement.

Il est préférable de tester la présence de NULL uniquement au début : lorsqu'un pointeur qui peut être NULL est reçu, par exemple, de `malloc()` ou d'une fonction qui peut lever une exception.

Les macros `Py_INCREF()` et `Py_DECREF()` ne vérifient pas les pointeurs NULL. Cependant, leurs variantes `Py_XINCREF()` et `Py_XDECREF()` le font.

The macros for checking for a particular object type (`Pytype_Check()`) don't check for NULL pointers --- again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with NULL checking.

Le mécanisme d'appel de fonctions C garantit que la liste d'arguments passée aux fonctions C (`args` dans les exemples) n'est jamais NULL. En fait, il garantit qu'il s'agit toujours d'un n-uplet<sup>4</sup>.

C'est une grave erreur de laisser un pointeur NULL "échapper" à l'utilisateur Python.

### 2.1.11 Écrire des extensions en C++

C'est possible d'écrire des modules d'extension en C++, mais sous certaines conditions. Si le programme principal (l'interpréteur Python) est compilé et lié par le compilateur C, les objets globaux ou statiques avec les constructeurs ne peuvent pas être utilisés. Ceci n'est pas un problème si le programme principal est relié par le compilateur C++. Les fonctions qui seront appelées par l'interpréteur Python (en particulier, les fonctions d'initialisation des modules) doivent être déclarées en utilisant `extern "C"`. Il n'est pas nécessaire d'inclure les fichiers d'en-tête Python dans le `extern "C" {...}`, car ils utilisent déjà ce format si le symbole `__cplusplus` est défini (tous les compilateurs C++ récents définissent ce symbole).

### 2.1.12 Fournir une API en langage C pour un module d'extension

De nombreux modules d'extension fournissent simplement de nouvelles fonctions et de nouveaux types à utiliser à partir de Python, mais parfois le code d'un module d'extension peut être utile pour d'autres modules d'extension. Par exemple, un module d'extension peut mettre en œuvre un type "collection" qui fonctionne comme des listes sans ordre. Tout comme le type de liste Python standard possède une API C qui permet aux modules d'extension de créer et de manipuler des listes, ce nouveau type de collection devrait posséder un ensemble de fonctions C pour une manipulation directe à partir d'autres modules d'extension.

À première vue, cela semble facile : il suffit d'écrire les fonctions (sans les déclarer "statiques", bien sûr), de fournir un fichier d'en-tête approprié et de documenter l'API C. Et en fait, cela fonctionnerait si tous les modules d'extension étaient toujours liés statiquement avec l'interpréteur Python. Cependant, lorsque les modules sont utilisés comme des bibliothèques partagées, les symboles définis dans un module peuvent ne pas être visibles par un autre module. Les détails de la visibilité dépendent du système d'exploitation ; certains systèmes utilisent un espace de noms global pour l'interpréteur Python et tous les modules d'extension (Windows, par exemple), tandis que d'autres exigent une liste explicite des symboles importés au moment de la liaison des modules (AIX en est un exemple), ou offrent un choix de stratégies différentes (la plupart des *Unix*). Et même si les symboles sont globalement visibles, le module dont on souhaite appeler les fonctions n'est peut-être pas encore chargé !

4. Ces garanties ne sont pas valables lorsqu'on emploie les conventions de nommage anciennes, qu'on retrouve encore assez souvent dans beaucoup de code existant.

La portabilité exige donc de ne faire aucune supposition sur la visibilité des symboles. Cela signifie que tous les symboles des modules d'extension doivent être déclarés `static`, à l'exception de la fonction d'initialisation du module, afin d'éviter les conflits de noms avec les autres modules d'extension (comme discuté dans la section *La fonction d'initialisation et le tableau des méthodes du module*). Et cela signifie que les symboles qui *devraient* être accessibles à partir d'autres modules d'extension doivent être exportés d'une manière différente.

Python provides a special mechanism to pass C-level information (pointers) from one extension module to another one : Capsules. A Capsule is a Python data type which stores a pointer (`void *`). Capsules can only be created and accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module's namespace. Other extension modules can then import this module, retrieve the value of this name, and then retrieve the pointer from the Capsule.

Il existe de nombreuses façons d'utiliser les Capsules pour exporter l'API C d'un module d'extension. Chaque fonction peut obtenir sa propre Capsule, ou tous les pointeurs de l'API C peuvent être stockés dans un tableau dont l'adresse est inscrite dans une Capsule. Et les différentes tâches de stockage et de récupération des pointeurs peuvent être réparties de différentes manières entre le module fournissant le code et les modules clients.

Whichever method you choose, it's important to name your Capsules properly. The function `PyCapsule_New()` takes a name parameter (`const char *`); you're permitted to pass in a `NULL` name, but we strongly encourage you to specify a name. Properly named Capsules provide a degree of runtime type-safety; there is no feasible way to tell one unnamed Capsule from another.

In particular, Capsules used to expose C APIs should be given a name following this convention :

```
modulename.attributename
```

The convenience function `PyCapsule_Import()` makes it easy to load a C API provided via a Capsule, but only if the Capsule's name matches this convention. This behavior gives C API users a high degree of certainty that the Capsule they load contains the correct C API.

L'exemple suivant montre une approche qui fait peser la plus grande partie de la charge sur le rédacteur du module d'exportation, ce qui est approprié pour les modules de bibliothèque couramment utilisés. Il stocke tous les pointeurs de l'API C (un seul dans l'exemple !) dans un tableau de pointeurs `void` qui devient la valeur d'une Capsule. Le fichier d'en-tête correspondant au module fournit une macro qui se charge d'importer le module et de récupérer ses pointeurs d'API C. Les modules clients n'ont qu'à appeler cette macro avant d'accéder à l'API C.

Le module d'exportation est une modification du module `spam` de la section *Un exemple simple*. La fonction `spam.system()` n'appelle pas directement la fonction de la bibliothèque C `system()`, mais une fonction `PySpam_System()`, qui ferait bien sûr quelque chose de plus compliqué en réalité (comme ajouter du *spam* à chaque commande). Cette fonction `PySpam_System()` est également exportée vers d'autres modules d'extension.

The function `PySpam_System()` is a plain C function, declared `static` like everything else :

```
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

La fonction `spam_system()` est modifiée de manière simple :

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
```

(suite sur la page suivante)

(suite de la page précédente)

```

        return NULL;
    sts = PySpam_System(command);
    return PyLong_FromLong(sts);
}

```

Au début du module, immédiatement après la ligne :

```
#include <Python.h>
```

on doit ajouter deux lignes supplémentaires :

```
#define SPAM_MODULE
#include "spammodule.h"
```

L'indicateur `#define` est utilisé pour indiquer au fichier d'en-tête qu'il est inclus dans le module d'exportation, et non dans un module client. Enfin, la fonction d'initialisation du module doit prendre en charge l'initialisation du tableau de pointeurs de l'API C :

```

PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a Capsule containing the API pointer array's address */
    c_api_object = PyCapsule_New((void *)PySpam_API, "spam._C_API", NULL);

    if (PyModule_AddObject(m, "_C_API", c_api_object) < 0) {
        Py_XDECREF(c_api_object);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

Notez que `PySpam_API` est déclaré `static`; sinon le tableau de pointeurs disparaîtrait lorsque `PyInit_spam`()` se finit !

L'essentiel du travail se trouve dans le fichier d'en-tête `spammodule.h`, qui ressemble à ceci :

```

#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

```

(suite sur la page suivante)

(suite de la page précédente)

```

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1

#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
    (*(PySpam_System_RETURN (*)(PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

/* Return -1 on error, 0 on success.
 * PyCapsule_Import will set an exception if there's an error.
 */
static int
import_spam(void)
{
    PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
    return (PySpam_API != NULL) ? 0 : -1;
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */

```

Tout ce qu'un module client doit faire pour avoir accès à la fonction `PySpam_System()` est d'appeler la fonction (ou plutôt la macro) `import_spam()` dans sa fonction d'initialisation :

```

PyMODINIT_FUNC
PyInit_client(void)
{
    PyObject *m;

    m = PyModule_Create(&clientmodule);
    if (m == NULL)
        return NULL;
    if (import_spam() < 0)
        return NULL;
    /* additional initialization can happen here */
    return m;
}

```

Le principal inconvénient de cette approche est que le fichier `spammodule.h` est assez compliqué. Cependant, la structure de base est la même pour chaque fonction exportée, ce qui fait qu'elle ne doit être apprise qu'une seule fois.

Enfin, il convient de mentionner que Capsules offrent des fonctionnalités supplémentaires, qui sont particulièrement utiles pour l'allocation de la mémoire et la dés-allocation du pointeur stocké dans un objet Capsule. Les détails sont décrits dans le manuel de référence de l'API Python/C dans la section capsules et dans l'implémentation des Capsules (fichiers `Include/pycapsule.h` et `Objects/pycapsule.c` dans la distribution du code source Python).

## Notes

## 2.2 Defining Extension Types : Tutorial

Python allows the writer of a C extension module to define new types that can be manipulated from Python code, much like the built-in `str` and `list` types. The code for all extension types follows a pattern, but there are some details that you need to understand before you can get started. This document is a gentle introduction to the topic.

### 2.2.1 The Basics

The *CPython* runtime sees all Python objects as variables of type `PyObject*`, which serves as a "base type" for all Python objects. The `PyObject` structure itself only contains the object's *reference count* and a pointer to the object's "type object". This is where the action is; the type object determines which (C) functions get called by the interpreter when, for instance, an attribute gets looked up on an object, a method called, or it is multiplied by another object. These C functions are called "type methods".

So, if you want to define a new extension type, you need to create a new type object.

This sort of thing can only be explained by example, so here's a minimal, but complete, module that defines a new type named `Custom` inside a C extension module `custom`:

---

**Note :** What we're showing here is the traditional way of defining *static* extension types. It should be adequate for most uses. The C API also allows defining heap-allocated extension types using the `PyType_FromSpec()` function, which isn't covered in this tutorial.

---

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} CustomObject;

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};

static PyModuleDef custommodule = {
```

(suite sur la page suivante)

```
PyModuleDef_HEAD_INIT,
.m_name = "custom",
.m_doc = "Example module that creates an extension type.",
.m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

Now that's quite a bit to take in at once, but hopefully bits will seem familiar from the previous chapter. This file defines three things :

1. What a Custom **object** contains : this is the CustomObject struct, which is allocated once for each Custom instance.
2. How the Custom **type** behaves : this is the CustomType struct, which defines a set of flags and function pointers that the interpreter inspects when specific operations are requested.
3. How to initialize the custom module : this is the PyInit\_custom function and the associated custommodule struct.

The first bit is :

```
typedef struct {
    PyObject_HEAD
} CustomObject;
```

This is what a Custom object will contain. PyObject\_HEAD is mandatory at the start of each object struct and defines a field called ob\_base of type PyObject, containing a pointer to a type object and a reference count (these can be accessed using the macros Py\_REFCNT and Py\_TYPE respectively). The reason for the macro is to abstract away the layout and to enable additional fields in debug builds.

**Note :** There is no semicolon above after the PyObject\_HEAD macro. Be wary of adding one by accident : some compilers will complain.

Of course, objects generally store additional data besides the standard PyObject\_HEAD boilerplate; for example, here is the definition for standard Python floats :

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

The second bit is the definition of the type object.

```
static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};
```

**Note :** We recommend using C99-style designated initializers as above, to avoid listing all the `PyTypeObject` fields that you don't care about and also to avoid caring about the fields' declaration order.

The actual definition of `PyTypeObject` in `object.h` has many more fields than the definition above. The remaining fields will be filled with zeros by the C compiler, and it's common practice to not specify them explicitly unless you need them.

We're going to pick it apart, one field at a time :

```
PyVarObject_HEAD_INIT(NULL, 0)
```

This line is mandatory boilerplate to initialize the `ob_base` field mentioned above.

```
.tp_name = "custom.Custom",
```

The name of our type. This will appear in the default textual representation of our objects and in some error messages, for example :

```
>>> "" + custom.Custom()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "custom.Custom") to str
```

Note that the name is a dotted name that includes both the module name and the name of the type within the module. The module in this case is `custom` and the type is `Custom`, so we set the type name to `custom.Custom`. Using the real dotted import path is important to make your type compatible with the `pydoc` and `pickle` modules.

```
.tp_basicsize = sizeof(CustomObject),
.tp_itemsize = 0,
```

This is so that Python knows how much memory to allocate when creating new `Custom` instances. `tp_itemsize` is only used for variable-sized objects and should otherwise be zero.

**Note :** If you want your type to be subclassable from Python, and your type has the same `tp_basicsize` as its base type, you may have problems with multiple inheritance. A Python subclass of your type will have to list your type first in its `__bases__`, or else it will not be able to call your type's `__new__()` method without getting an error. You can avoid this problem by ensuring that your type has a larger value for `tp_basicsize` than its base type does. Most of the

time, this will be true anyway, because either your base type will be `object`, or else you will be adding data members to your base type, and therefore increasing its size.

We set the class flags to `Py_TPFLAGS_DEFAULT`.

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

All types should include this constant in their flags. It enables all of the members defined until at least Python 3.3. If you need further members, you will need to OR the corresponding flags.

We provide a doc string for the type in `tp_doc`.

```
.tp_doc = "Custom objects",
```

To enable object creation, we have to provide a `tp_new` handler. This is the equivalent of the Python method `__new__()`, but has to be specified explicitly. In this case, we can just use the default implementation provided by the API function `PyType_GenericNew()`.

```
.tp_new = PyType_GenericNew,
```

Everything else in the file should be familiar, except for some code in `PyInit_custom()` :

```
if (PyType_Ready(&CustomType) < 0)
    return;
```

This initializes the `Custom` type, filling in a number of members to the appropriate default values, including `ob_type` that we initially set to `NULL`.

```
Py_INCREF(&CustomType);
if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
    Py_DECREF(&CustomType);
    Py_DECREF(m);
    return NULL;
}
```

This adds the type to the module dictionary. This allows us to create `Custom` instances by calling the `Custom` class :

```
>>> import custom
>>> mycustom = custom.Custom()
```

That's it ! All that remains is to build it; put the above code in a file called `custom.c` and :

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[Extension("custom", ["custom.c"])])
```

in a file called `setup.py`; then typing

```
$ python setup.py build
```

at a shell should produce a file `custom.so` in a subdirectory; move to that directory and fire up Python --- you should be able to `import custom` and play around with `Custom` objects.

That wasn't so hard, was it?

Of course, the current `Custom` type is pretty uninteresting. It has no data and doesn't do anything. It can't even be subclassed.



**Note :** While this documentation showcases the standard `distutils` module for building C extensions, it is recommended in real-world use cases to use the newer and better-maintained `setuptools` library. Documentation on how to do this is out of scope for this document and can be found in the [Python Packaging User's Guide](#).

## 2.2.2 Adding data and methods to the Basic example

Let's extend the basic example to add some data and methods. Let's also make the type usable as a base class. We'll create a new module, `custom2` that adds these capabilities :

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;
```

(suite sur la page suivante)

```

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))

        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom2.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),

```

(suite sur la page suivante)

(suite de la page précédente)

```

        .tp_itemsize = 0,
        .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
        .tp_new = Custom_new,
        .tp_init = (initproc) Custom_init,
        .tp_dealloc = (destructor) Custom_dealloc,
        .tp_members = Custom_members,
        .tp_methods = Custom_methods,
    };

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom2",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom2(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

This version of the module has a number of changes.

We've added an extra include :

```
#include <structmember.h>
```

This include provides declarations that we use to handle attributes, as described a bit later.

The `Custom` type now has three data attributes in its C struct, *first*, *last*, and *number*. The *first* and *last* variables are Python strings containing first and last names. The *number* attribute is a C integer.

The object structure is updated accordingly :

```

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

```

Because we now have data to manage, we have to be more careful about object allocation and deallocation. At a minimum,

we need a deallocation method :

```
static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

which is assigned to the `tp_dealloc` member :

```
.tp_dealloc = (destructor) Custom_dealloc,
```

This method first clears the reference counts of the two Python attributes. `Py_XDECREF()` correctly handles the case where its argument is `NULL` (which might happen here if `tp_new` failed midway). It then calls the `tp_free` member of the object's type (computed by `Py_TYPE(self)`) to free the object's memory. Note that the object's type might not be `CustomType`, because the object may be an instance of a subclass.

---

**Note :** The explicit cast to `destructor` above is needed because we defined `Custom_dealloc` to take a `CustomObject *` argument, but the `tp_dealloc` function pointer expects to receive a `PyObject *` argument. Otherwise, the compiler will emit a warning. This is object-oriented polymorphism, in C!

---

We want to make sure that the first and last names are initialized to empty strings, so we provide a `tp_new` implementation :

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}
```

and install it in the `tp_new` member :

```
.tp_new = Custom_new,
```

The `tp_new` handler is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. It is not required to define a `tp_new` member, and indeed many extension types will simply reuse `PyType_GenericNew()` as done in the first version of the `Custom` type above. In this case, we use the `tp_new` handler to initialize the `first` and `last` attributes to non-`NULL` default values.

`tp_new` is passed the type being instantiated (not necessarily `CustomType`, if a subclass is instantiated) and any arguments passed when the type was called, and is expected to return the instance created. `tp_new` handlers always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer (a.k.a. `tp_init` in C or `__init__` in Python) methods.

**Note :** `tp_new` shouldn't call `tp_init` explicitly, as the interpreter will do it itself.

The `tp_new` implementation calls the `tp_alloc` slot to allocate memory :

```
self = (CustomObject *) type->tp_alloc(type, 0);
```

Since memory allocation may fail, we must check the `tp_alloc` result against `NULL` before proceeding.

**Note :** We didn't fill the `tp_alloc` slot ourselves. Rather `PyType_Ready()` fills it for us by inheriting it from our base class, which is `object` by default. Most types use the default allocation strategy.

**Note :** If you are creating a co-operative `tp_new` (one that calls a base type's `tp_new` or `__new__()`), you must *not* try to determine what method to call using method resolution order at runtime. Always statically determine what type you are going to call, and call its `tp_new` directly, or via `type->tp_base->tp_new`. If you do not do this, Python subclasses of your type that also inherit from other Python-defined classes may not work correctly. (Specifically, you may not be able to create instances of such subclasses without getting a `TypeError`.)

We also define an initialization function which accepts arguments to provide initial values for our instance :

```
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}
```

by filling the `tp_init` slot.

```
.tp_init = (initproc) Custom_init,
```

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created. Initializers always accept positional and keyword arguments, and they should return either 0 on success or -1 on error.

Unlike the `tp_new` handler, there is no guarantee that `tp_init` is called at all (for example, the `pickle` module by default doesn't call `__init__()` on unpickled instances). It can also be called multiple times. Anyone can call the `__init__()` method on our objects. For this reason, we have to be extra careful when assigning the new attribute values. We might be tempted, for example to assign the `first` member like this :

```
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}
```

But this would be risky. Our type doesn't restrict the type of the `first` member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the `first` member; or that destructor could release the *Global interpreter Lock* and let arbitrary code run in other threads that accesses and modifies our object.

To be paranoid and protect ourselves against this possibility, we almost always reassign members before decrementing their reference counts. When don't we have to do this ?

- when we absolutely know that the reference count is greater than 1 ;
- when we know that deallocation of the object<sup>1</sup> will neither release the *GIL* nor cause any calls back into our type's code ;
- when decrementing a reference count in a `tp_dealloc` handler on a type which doesn't support cyclic garbage collection<sup>2</sup>.

We want to expose our instance variables as attributes. There are a number of ways to do that. The simplest way is to define member definitions :

```
static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};
```

and put the definitions in the `tp_members` slot :

```
.tp_members = Custom_members,
```

Each member definition has a member name, type, offset, access flags and documentation string. See the *Gestion des attributs génériques* section below for details.

A disadvantage of this approach is that it doesn't provide a way to restrict the types of objects that can be assigned to the Python attributes. We expect the first and last names to be strings, but any Python objects can be assigned. Further, the attributes can be deleted, setting the C pointers to NULL. Even though we can make sure the members are initialized to non-NULL values, the members can be set to NULL if the attributes are deleted.

We define a single method, `Custom.name()`, that outputs the objects name as the concatenation of the first and last names.

1. This is true when we know that the object is a basic type, like a string or a float.

2. We relied on this in the `tp_dealloc` handler in this example, because our type doesn't support garbage collection.

```
static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

The method is implemented as a C function that takes a Custom (or Custom subclass) instance as the first argument. Methods always take an instance as the first argument. Methods often take positional and keyword arguments as well, but in this case we don't take any and don't need to accept a positional argument tuple or keyword argument dictionary. This method is equivalent to the Python method :

```
def name(self):
    return "%s %s" % (self.first, self.last)
```

Note that we have to check for the possibility that our first and last members are NULL. This is because they can be deleted, in which case they are set to NULL. It would be better to prevent deletion of these attributes and to restrict the attribute values to be strings. We'll see how to do that in the next section.

Now that we've defined the method, we need to create an array of method definitions :

```
static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};
```

(note that we used the METH\_NOARGS flag to indicate that the method is expecting no arguments other than self)

and assign it to the tp\_methods slot :

```
.tp_methods = Custom_methods,
```

Finally, we'll make our type usable as a base class for subclassing. We've written our methods carefully so far so that they don't make any assumptions about the type of the object being created or used, so all we need to do is to add the Py\_TPFLAGS\_BASETYPE to our class flag definition :

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
```

We rename PyInit\_custom() to PyInit\_custom2(), update the module name in the PyModuleDef struct, and update the full class name in the PyTypeObject struct.

Finally, we update our setup.py file to build the new module :

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[
          Extension("custom", ["custom.c"]),
          Extension("custom2", ["custom2.c"]),
      ])
```

## 2.2.3 Providing finer control over data attributes

In this section, we'll provide finer control over how the `first` and `last` attributes are set in the `Custom` example. In the previous version of our module, the instance variables `first` and `last` could be set to non-string values or even deleted. We want to make sure that these attributes always contain strings.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwargs)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
```

(suite sur la page suivante)



(suite de la page précédente)

```

        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{

```

(suite sur la page suivante)

(suite de la page précédente)

```
PyObject *tmp;
if (value == NULL) {
    PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
    return -1;
}
if (!PyUnicode_Check(value)) {
    PyErr_SetString(PyExc_TypeError,
                    "The last attribute value must be a string");
    return -1;
}
tmp = self->last;
Py_INCREF(value);
self->last = value;
Py_DECREF(tmp);
return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom3.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom3",
    .m_doc = "Example module that creates an extension type.",

```

(suite sur la page suivante)

(suite de la page précédente)

```

        .m_size = -1,
    };

PyMODINIT_FUNC
PyInit_custom3(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

To provide greater control, over the first and last attributes, we'll use custom getter and setter functions. Here are the functions for getting and setting the first attribute :

```

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}

```

The getter function is passed a Custom object and a "closure", which is a void pointer. In this case, the closure is ignored. (The closure supports an advanced usage in which definition data is passed to the getter and setter. This could, for example, be used to allow a single set of getter and setter functions that decide the attribute to get or set based on data in the closure.)

The setter function is passed the `Custom` object, the new value, and the closure. The new value may be `NULL`, in which case the attribute is being deleted. In our setter, we raise an error if the attribute is deleted or if its new value is not a string.

We create an array of `PyGetSetDef` structures :

```
static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};
```

and register it in the `tp_getset` slot :

```
.tp_getset = Custom_getsetters,
```

The last item in a `PyGetSetDef` structure is the "closure" mentioned above. In this case, we aren't using a closure, so we just pass `NULL`.

We also remove the member definitions for these attributes :

```
static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};
```

We also need to update the `tp_init` handler to only allow strings<sup>3</sup> to be passed :

```
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwargs)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}
```

3. We now know that the first and last members are strings, so perhaps we could be less careful about decrementing their reference counts, however, we accept instances of string subclasses. Even though deallocating normal strings won't call back into our objects, we can't guarantee that deallocating an instance of a string subclass won't call back into our objects.

With these changes, we can assure that the `first` and `last` members are never `NULL` so we can remove checks for `NULL` values in almost all cases. This means that most of the `Py_XDECREF()` calls can be converted to `Py_DECREF()` calls. The only place we can't change these calls is in the `tp_dealloc` implementation, where there is the possibility that the initialization of these members failed in `tp_new`.

We also rename the module initialization function and module name in the initialization function, as we did before, and we add an extra definition to the `setup.py` file.

## 2.2.4 Supporting cyclic garbage collection

Python has a *cyclic garbage collector (GC)* that can identify unneeded objects even when their reference counts are not zero. This can happen when objects are involved in cycles. For example, consider :

```
>>> l = []
>>> l.append(l)
>>> del l
```

In this example, we create a list that contains itself. When we delete it, it still has a reference from itself. Its reference count doesn't drop to zero. Fortunately, Python's cyclic garbage collector will eventually figure out that the list is garbage and free it.

In the second version of the `Custom` example, we allowed any kind of object to be stored in the `first` or `last` attributes<sup>4</sup>. Besides, in the second and third versions, we allowed subclassing `Custom`, and subclasses may add arbitrary attributes. For any of those two reasons, `Custom` objects can participate in cycles :

```
>>> import custom3
>>> class Derived(custom3.Custom): pass
...
>>> n = Derived()
>>> n.some_attribute = n
```

To allow a `Custom` instance participating in a reference cycle to be properly detected and collected by the cyclic GC, our `Custom` type needs to fill two additional slots and to enable a flag that enables these slots :

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include "structmember.h"

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last; /* last name */
    int number;
} CustomObject;

static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}

static int
```

(suite sur la page suivante)

4. Also, even with our attributes restricted to strings instances, the user could pass arbitrary `str` subclasses and therefore still create reference cycles.

(suite de la page précédente)

```
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

        Py_DECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->first);
    self->first = value;
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->last);

```

(suite sur la page suivante)

(suite de la page précédente)

```

        self->last = value;
        return 0;
    }

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_traverse = (traverseproc) Custom_traverse,
    .tp_clear = (inquiry) Custom_clear,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom4(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);

```

(suite sur la page suivante)



(suite de la page précédente)

```

    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

First, the traversal method lets the cyclic GC know about subobjects that could participate in cycles :

```

static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    int vret;
    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }
    return 0;
}

```

For each subobject that can participate in cycles, we need to call the `visit()` function, which is passed to the traversal method. The `visit()` function takes as arguments the subobject and the extra argument `arg` passed to the traversal method. It returns an integer value that must be returned if it is non-zero.

Python provides a `Py_VISIT()` macro that automates calling visit functions. With `Py_VISIT()`, we can minimize the amount of boilerplate in `Custom_traverse` :

```

static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}

```

**Note :** The `tp_traverse` implementation must name its arguments exactly `visit` and `arg` in order to use `Py_VISIT()`.

Second, we need to provide a method for clearing any subobjects that can participate in cycles :

```

static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
}

```

(suite sur la page suivante)

(suite de la page précédente)

```
Py_CLEAR(self->last);
return 0;
}
```

Notice the use of the `Py_CLEAR()` macro. It is the recommended and safe way to clear data attributes of arbitrary types while decrementing their reference counts. If you were to call `Py_XDECREF()` instead on the attribute before setting it to `NULL`, there is a possibility that the attribute's destructor would call back into code that reads the attribute again (*especially* if there is a reference cycle).

**Note :** You could emulate `Py_CLEAR()` by writing :

```
PyObject *tmp;
tmp = self->first;
self->first = NULL;
Py_XDECREF(tmp);
```

Nevertheless, it is much easier and less error-prone to always use `Py_CLEAR()` when deleting an attribute. Don't try to micro-optimize at the expense of robustness !

The deallocator `Custom_dealloc` may call arbitrary code when clearing attributes. It means the circular GC can be triggered inside the function. Since the GC assumes reference count is not zero, we need to untrack the object from the GC by calling `PyObject_GC_UnTrack()` before clearing members. Here is our reimplemented deallocator using `PyObject_GC_UnTrack()` and `Custom_clear` :

```
static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

Finally, we add the `Py_TPFLAGS_HAVE_GC` flag to the class flags :

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
```

That's pretty much it. If we had written custom `tp_alloc` or `tp_free` handlers, we'd need to modify them for cyclic garbage collection. Most extensions will use the versions automatically provided.

## 2.2.5 Subclassing other types

It is possible to create new extension types that are derived from existing types. It is easiest to inherit from the built in types, since an extension can easily use the `PyTypeObject` it needs. It can be difficult to share these `PyTypeObject` structures between extension modules.

In this example we will create a `SubList` type that inherits from the built-in `list` type. The new type will be completely compatible with regular lists, but will have an additional `increment()` method that increases an internal counter :

```
>>> import sublist
>>> s = sublist.SubList(range(3))
>>> s.extend(s)
>>> print(len(s))
6
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> print(s.increment())
1
>>> print(s.increment())
2
```

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

typedef struct {
    PyObject list;
    int state;
} SubListObject;

static PyObject *
SubList_increment(SubListObject *self, PyObject *unused)
{
    self->state++;
    return PyLong_FromLong(self->state);
}

static PyMethodDef SubList_methods[] = {
    {"increment", (PyCFunction) SubList_increment, METH_NOARGS,
     PyDoc_STR("increment state counter")},
    {NULL},
};

static int
SubList_init(SubListObject *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *) self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}

static PyTypeObject SubListType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "sublist.SubList",
    .tp_doc = "SubList objects",
    .tp_basicsize = sizeof(SubListObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_init = (initproc) SubList_init,
    .tp_methods = SubList_methods,
};

static PyModuleDef sublistmodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "sublist",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_sublist(void)
{

```

(suite sur la page suivante)

(suite de la page précédente)

```
PyObject *m;
SubListType.tp_base = &PyList_Type;
if (PyType_Ready(&SubListType) < 0)
    return NULL;

m = PyModule_Create(&sublistmodule);
if (m == NULL)
    return NULL;

Py_INCREF(&SubListType);
if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
    Py_DECREF(&SubListType);
    Py_DECREF(m);
    return NULL;
}

return m;
}
```

As you can see, the source code closely resembles the Custom examples in previous sections. We will break down the main differences between them.

```
typedef struct {
    PyListObject list;
    int state;
} SubListObject;
```

The primary difference for derived type objects is that the base type's object structure must be the first value. The base type will already include the `PyObject_HEAD()` at the beginning of its structure.

When a Python object is a `SubList` instance, its `PyObject *` pointer can be safely cast to both `PyListObject *` and `SubListObject *`:

```
static int
SubList_init(SubListObject *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *) self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}
```

We see above how to call through to the `__init__` method of the base type.

This pattern is important when writing a type with custom `tp_new` and `tp_dealloc` members. The `tp_new` handler should not actually create the memory for the object with its `tp_alloc`, but let the base class handle it by calling its own `tp_new`.

The `PyTypeObject` struct supports a `tp_base` specifying the type's concrete base class. Due to cross-platform compiler issues, you can't fill that field directly with a reference to `PyList_Type`; it should be done later in the module initialization function :

```
PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject* m;
    SubListType.tp_base = &PyList_Type;
```

(suite sur la page suivante)

(suite de la page précédente)

```

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

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

Before calling `PyType_Ready()`, the type structure must have the `tp_base` slot filled in. When we are deriving an existing type, it is not necessary to fill out the `tp_alloc` slot with `PyType_GenericNew()` -- the allocation function from the base type will be inherited.

After that, calling `PyType_Ready()` and adding the type object to the module is the same as with the basic Custom examples.

## Notes

## 2.3 Définir les types d'extension : divers sujets

This section aims to give a quick fly-by on the various type methods you can implement and what they do.

Here is the definition of `PyTypeObject`, with some fields only used in debug builds omitted :

```

typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrofunc tp_getattr;
    setattrofunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                   or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

```

(suite sur la page suivante)

```

hashfunc tp_hash;
ternaryfunc tp_call;
reprfunc tp_str;
getattrofunc tp_getattro;
setattrofunc tp_setattro;

/* Functions to access object as input/output buffer */
PyBufferProcs *tp_as_buffer;

/* Flags to define presence of optional/expanded features */
unsigned long tp_flags;

const char *tp_doc; /* Documentation string */

/* call function for all accessible objects */
traverseproc tp_traverse;

/* delete references to contained objects */
inquiry tp_clear;

/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
Py_ssize_t tp_weaklistoffset;

/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
descrsetfunc tp_descr_set;
Py_ssize_t tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new;
freefunc tp_free; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
PyObject *tp_cache;
PyObject *tp_subclasses;
PyObject *tp_weaklist;
destructor tp_del;

/* Type attribute cache version tag. Added in version 2.6 */
unsigned int tp_version_tag;

destructor tp_finalize;
} PyTypeObject;

```

Now that's a *lot* of methods. Don't worry too much though -- if you have a type you want to define, the chances are very good that you will only implement a handful of these.

As you probably expect by now, we're going to go over this and give more information about the various handlers. We won't go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields. It's often easiest to find an example that includes the fields you need and then change the values to suit your new type.

```
const char *tp_name; /* For printing */
```

The name of the type -- as mentioned in the previous chapter, this will appear in various places, almost entirely for diagnostic purposes. Try to choose something that will be helpful in such a situation !

```
Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */
```

These fields tell the runtime how much memory to allocate when new objects of this type are created. Python has some built-in support for variable length structures (think : strings, tuples) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```
const char *tp_doc;
```

Ici vous pouvez mettre une chaîne (ou son adresse) que vous voulez renvoyer lorsque le script Python référence `obj.__doc__` pour récupérer le *docstring*.

Nous en arrivons maintenant aux méthodes de type basiques -- celles que la plupart des types d'extension mettront en œuvre.

### 2.3.1 Finalisation et de-allocation

```
destructor tp_dealloc;
```

This function is called when the reference count of the instance of your type is reduced to zero and the Python interpreter wants to reclaim it. If your type has memory to free or other clean-up to perform, you can put it here. The object itself needs to be freed here as well. Here is an example of this function :

```
static void
newdatatype_dealloc(newdatatypeobject *obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    Py_TYPE(obj)->tp_free(obj);
}
```

One important requirement of the deallocator function is that it leaves any pending exceptions alone. This is important since deallocators are frequently called as the interpreter unwinds the Python stack; when the stack is unwound due to an exception (rather than normal returns), nothing is done to protect the deallocators from seeing that an exception has already been set. Any actions which a deallocator performs which may cause additional Python code to be executed may detect that an exception has been set. This can lead to misleading errors from the interpreter. The proper way to protect against this is to save a pending exception before performing the unsafe action, and restoring it when done. This can be done using the `PyErr_Fetch()` and `PyErr_Restore()` functions :

```
static void
my_dealloc(PyObject *obj)
{
    PyObject *self = (PyObject *) obj;
    PyObject *cbresult;
```

(suite sur la page suivante)

(suite de la page précédente)

```

if (self->my_callback != NULL) {
    PyObject *err_type, *err_value, *err_traceback;

    /* This saves the current exception state */
    PyErr_Fetch(&err_type, &err_value, &err_traceback);

    cbresult = PyObject_CallNoArgs(self->my_callback);
    if (cbresult == NULL)
        PyErr_WriteUnraisable(self->my_callback);
    else
        Py_DECREF(cbresult);

    /* This restores the saved exception state */
    PyErr_Restore(err_type, err_value, err_traceback);

    Py_DECREF(self->my_callback);
}
Py_TYPE(obj)->tp_free((PyObject*)self);
}

```

**Note :** There are limitations to what you can safely do in a deallocator function. First, if your type supports garbage collection (using `tp_traverse` and/or `tp_clear`), some of the object's members can have been cleared or finalized by the time `tp_dealloc` is called. Second, in `tp_dealloc`, your object is in an unstable state : its reference count is equal to zero. Any call to a non-trivial object or API (as in the example above) might end up calling `tp_dealloc` again, causing a double free and a crash.

Starting with Python 3.4, it is recommended not to put any complex finalization code in `tp_dealloc`, and instead use the new `tp_finalize` type method.

**Voir aussi :**

**PEP 442** explique le nouveau schéma de finalisation.

## 2.3.2 Présentation de l'objet

In Python, there are two ways to generate a textual representation of an object : the `repr()` function, and the `str()` function. (The `print()` function just calls `str()`.) These handlers are both optional.

```

reprfunc tp_repr;
reprfunc tp_str;

```

The `tp_repr` handler should return a string object containing a representation of the instance for which it is called. Here is a simple example :

```

static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Repr-ified_newdatatype{{size:%d}}",
                                obj->obj_UnderlyingDatatypePtr->size);
}

```

If no `tp_repr` handler is specified, the interpreter will supply a representation that uses the type's `tp_name` and a uniquely-identifying value for the object.



The `tp_str` handler is to `str()` what the `tp_repr` handler described above is to `repr()`; that is, it is called when Python code calls `str()` on an instance of your object. Its implementation is very similar to the `tp_repr` function, but the resulting string is intended for human consumption. If `tp_str` is not specified, the `tp_repr` handler is used instead.

Voici un exemple simple :

```
static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Stringified_newdatatype{{size:%d}}",
                                obj->obj_UnderlyingDatatypePtr->size);
}
```

### 2.3.3 Gestion des attributs

For every object which can support attributes, the corresponding type must provide the functions that control how the attributes are resolved. There needs to be a function which can retrieve attributes (if any are defined), and another to set attributes (if setting attributes is allowed). Removing an attribute is a special case, for which the new value passed to the handler is `NULL`.

Python supports two pairs of attribute handlers; a type that supports attributes only needs to implement the functions for one pair. The difference is that one pair takes the name of the attribute as a `char*`, while the other accepts a `PyObject*`. Each type can use whichever pair makes more sense for the implementation's convenience.

```
getattrfunc tp_getattr;      /* char * version */
setattrfunc tp_setattr;
/* ... */
getattrrofunc tp_getattro;   /* PyObject * version */
setattrrofunc tp_setattro;
```

If accessing attributes of an object is always a simple operation (this will be explained shortly), there are generic implementations which can be used to provide the `PyObject*` version of the attribute management functions. The actual need for type-specific attribute handlers almost completely disappeared starting with Python 2.2, though there are many examples which have not been updated to use some of the new generic mechanism that is available.

### Gestion des attributs génériques

Most extension types only use *simple* attributes. So, what makes the attributes simple? There are only a couple of conditions that must be met :

1. Le nom des attributs doivent être déjà connus lorsqu'on lance `PyType_Ready()`.
2. No special processing is needed to record that an attribute was looked up or set, nor do actions need to be taken based on the value.

Note that this list does not place any restrictions on the values of the attributes, when the values are computed, or how relevant data is stored.

When `PyType_Ready()` is called, it uses three tables referenced by the type object to create *descriptors* which are placed in the dictionary of the type object. Each descriptor controls access to one attribute of the instance object. Each of the tables is optional; if all three are `NULL`, instances of the type will only have attributes that are inherited from their base type, and should leave the `tp_getattro` and `tp_setattro` fields `NULL` as well, allowing the base type to handle attributes.

Les tables sont déclarées sous la forme de trois champs de type objet :

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

If `tp_methods` is not `NULL`, it must refer to an array of `PyMethodDef` structures. Each entry in the table is an instance of this structure :

```
typedef struct PyMethodDef {
    const char *ml_name;           /* method name */
    PyCFunction ml_meth;          /* implementation function */
    int ml_flags;                  /* flags */
    const char *ml_doc;            /* docstring */
} PyMethodDef;
```

One entry should be defined for each method provided by the type; no entries are needed for methods inherited from a base type. One additional entry is needed at the end; it is a sentinel that marks the end of the array. The `ml_name` field of the sentinel must be `NULL`.

The second table is used to define attributes which map directly to data stored in the instance. A variety of primitive C types are supported, and access may be read-only or read-write. The structures in the table are defined as :

```
typedef struct PyMemberDef {
    const char *name;
    int type;
    int offset;
    int flags;
    const char *doc;
} PyMemberDef;
```

For each entry in the table, a *descriptor* will be constructed and added to the type which will be able to extract a value from the instance structure. The `type` field should contain one of the type codes defined in the `structmember.h` header; the value will be used to determine how to convert Python values to and from C values. The `flags` field is used to store flags which control how the attribute can be accessed.

The following flag constants are defined in `structmember.h`; they may be combined using bitwise-OR.

Constante	Signification
<code>READONLY</code>	Jamais disponible en écriture.
<code>READ_RESTRICTED</code>	Non disponible en lecture, dans le mode restreint.
<code>WRITE_RESTRICTED</code>	Non disponible en écriture dans le mode restreint.
<code>RESTRICTED</code>	Non disponible en lecture ou écriture, en mode restreint.

Un avantage intéressant de l'utilisation de la table `tp_members` pour construire les descripteurs qui sont utilisés à l'exécution, est que à tout attribut défini de cette façon on peut associer un *docstring*, en écrivant simplement le texte dans la table. Une application peut utiliser l'API d'inspection pour récupérer le descripteur de l'objet de classe, et utiliser son attribut `__doc__` pour renvoyer le *docstring*.

As with the `tp_methods` table, a sentinel entry with a name value of `NULL` is required.

## Gestion des attributs de type spécifiques

For simplicity, only the `char*` version will be demonstrated here; the type of the name parameter is the only difference between the `char*` and `PyObject*` flavors of the interface. This example effectively does the same thing as the generic example above, but does not use the generic support added in Python 2.2. It explains how the handler functions are called, so that if you do need to extend their functionality, you'll understand what needs to be done.

The `tp_getattr` handler is called when the object requires an attribute look-up. It is called in the same situations where the `__getattr__()` method of a class would be called.

Voici un exemple :

```
static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    if (strcmp(name, "data") == 0)
    {
        return PyLong_FromLong(obj->data);
    }

    PyErr_Format(PyExc_AttributeError,
                 "'%.50s' object has no attribute '%.400s'",
                 tp->tp_name, name);
    return NULL;
}
```

The `tp_setattr` handler is called when the `__setattr__()` or `__delattr__()` method of a class instance would be called. When an attribute should be deleted, the third parameter will be `NULL`. Here is an example that simply raises an exception; if this were really all you wanted, the `tp_setattr` handler should be set to `NULL`.

```
static int
newdatatype_setattr(newdatatypeobject *obj, char *name, PyObject *v)
{
    PyErr_Format(PyExc_RuntimeError, "Read-only attribute: %s", name);
    return -1;
}
```

## 2.3.4 Comparaison des objets

```
richcmpfunc tp_richcompare;
```

The `tp_richcompare` handler is called when comparisons are needed. It is analogous to the rich comparison methods, like `__lt__()`, and also called by `PyObject_RichCompare()` and `PyObject_RichCompareBool()`.

This function is called with two Python objects and the operator as arguments, where the operator is one of `Py_EQ`, `Py_NE`, `Py_LE`, `Py_GT`, `Py_LT` or `Py_GE`. It should compare the two objects with respect to the specified operator and return `Py_True` or `Py_False` if the comparison is successful, `Py_NotImplemented` to indicate that comparison is not implemented and the other object's comparison method should be tried, or `NULL` if an exception was set.

Here is a sample implementation, for a datatype that is considered equal if the size of an internal pointer is equal :

```
static PyObject *
newdatatype_richcmp(PyObject *obj1, PyObject *obj2, int op)
{
    PyObject *result;
    int c, size1, size2;
```

(suite sur la page suivante)

(suite de la page précédente)

```

/* code to make sure that both arguments are of type
   newdatatype omitted */

size1 = obj1->obj_UnderlyingDatatypePtr->size;
size2 = obj2->obj_UnderlyingDatatypePtr->size;

switch (op) {
case Py_LT: c = size1 < size2; break;
case Py_LE: c = size1 <= size2; break;
case Py_EQ: c = size1 == size2; break;
case Py_NE: c = size1 != size2; break;
case Py_GT: c = size1 > size2; break;
case Py_GE: c = size1 >= size2; break;
}
result = c ? Py_True : Py_False;
Py_INCREF(result);
return result;
}

```

### 2.3.5 Support pour le protocole abstrait

Python supports a variety of *abstract* 'protocols;' the specific interfaces provided to use these interfaces are documented in abstract.

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object. For newer protocols there are additional slots in the main type object, with a flag bit being set to indicate that the slots are present and should be checked by the interpreter. (The flag bit does not indicate that the slot values are non-NULL. The flag may be set to indicate the presence of a slot, but a slot may still be unfilled.)

```

PyNumberMethods    *tp_as_number;
PySequenceMethods  *tp_as_sequence;
PyMappingMethods    *tp_as_mapping;

```

If you wish your object to be able to act like a number, a sequence, or a mapping object, then you place the address of a structure that implements the C type `PyNumberMethods`, `PySequenceMethods`, or `PyMappingMethods`, respectively. It is up to you to fill in this structure with appropriate values. You can find examples of the use of each of these in the `Objects` directory of the Python source distribution.

```
hashfunc tp_hash;
```

This function, if you choose to provide it, should return a hash number for an instance of your data type. Here is a simple example :

```

static Py_hash_t
newdatatype_hash(newdatatypeobject *obj)
{
    Py_hash_t result;
    result = obj->some_size + 32767 * obj->some_number;
    if (result == -1)
        result = -2;
}

```

(suite sur la page suivante)

(suite de la page précédente)

```

    return result;
}

```

`Py_hash_t` is a signed integer type with a platform-varying width. Returning `-1` from `tp_hash` indicates an error, which is why you should be careful to avoid returning it when hash computation is successful, as seen above.

```
ternaryfunc tp_call;
```

This function is called when an instance of your data type is "called", for example, if `obj1` is an instance of your data type and the Python script contains `obj1('hello')`, the `tp_call` handler is invoked.

Cette fonction prend trois arguments :

1. *self* is the instance of the data type which is the subject of the call. If the call is `obj1('hello')`, then *self* is `obj1`.
2. *args* is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.
3. *kwargs* is a dictionary of keyword arguments that were passed. If this is non-NULL and you support keyword arguments, use `PyArg_ParseTupleAndKeywords()` to extract the arguments. If you do not want to support keyword arguments and this is non-NULL, raise a `TypeError` with a message saying that keyword arguments are not supported.

Ceci est une implémentation `tp_call` très simple :

```

static PyObject *
newdatatypeobject_call(PyObject *self, PyObject *args, PyObject *kwargs)
{
    PyObject *result;
    const char *arg1;
    const char *arg2;
    const char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3)) {
        return NULL;
    }
    result = PyUnicode_FromFormat(
        "Returning -- value: [%d] arg1: [%s] arg2: [%s] arg3: [%s]\n",
        obj->obj_UnderlyingDatatypePtr->size,
        arg1, arg2, arg3);
    return result;
}

```

```

/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

```

These functions provide support for the iterator protocol. Both handlers take exactly one parameter, the instance for which they are being called, and return a new reference. In the case of an error, they should set an exception and return NULL. `tp_iter` corresponds to the Python `__iter__()` method, while `tp_iternext` corresponds to the Python `__next__()` method.

Tout objet *iterable* doit implémenter le gestionnaire `tp_iter`, qui doit renvoyer un objet de type *iterator*. Ici, les mêmes directives s'appliquent de la même façon que pour les classes *Python* :

- Pour les collections (telles que les listes et les n-uplets) qui peuvent implémenter plusieurs itérateurs indépendants, un nouvel itérateur doit être créé et renvoyé par chaque appel de type `tp_iter`.

- Objects which can only be iterated over once (usually due to side effects of iteration, such as file objects) can implement `tp_iter` by returning a new reference to themselves -- and should also therefore implement the `tp_iternext` handler.

Any *iterator* object should implement both `tp_iter` and `tp_iternext`. An iterator's `tp_iter` handler should return a new reference to the iterator. Its `tp_iternext` handler should return a new reference to the next object in the iteration, if there is one. If the iteration has reached the end, `tp_iternext` may return `NULL` without setting an exception, or it may set `StopIteration` *in addition* to returning `NULL`; avoiding the exception can yield slightly better performance. If an actual error occurs, `tp_iternext` should always set an exception and return `NULL`.

## 2.3.6 Prise en charge de la référence faible

L'un des objectifs de l'implémentation de la référence faible de *Python* est de permettre à tout type d'objet de participer au mécanisme de référence faible sans avoir à supporter le surcoût de la performance critique des certains objets, tels que les nombres.

### Voir aussi :

Documentation pour le module `weakref`.

Pour qu'un objet soit faiblement référençable, le type d'extension doit faire deux choses :

1. Inclure un champ `PyObject*` dans la structure d'objet C dédiée au mécanisme de référence faible. Le constructeur de l'objet doit le laisser à la valeur `NULL` (ce qui est automatique lorsque l'on utilise le champ par défaut `tp_alloc`).
2. Définissez le membre de type `tp_weaklistoffset` à la valeur de décalage (*offset*) du champ susmentionné dans la structure de l'objet C, afin que l'interpréteur sache comment accéder à ce champ et le modifier.

Concrètement, voici comment une structure d'objet simple serait complétée par le champ requis :

```
typedef struct {
    PyObject_HEAD
    PyObject *weakreflist; /* List of weak references */
} TrivialObject;
```

Et le membre correspondant dans l'objet de type déclaré statiquement :

```
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    /* ... other members omitted for brevity ... */
    .tp_weaklistoffset = offsetof(TrivialObject, weakreflist),
};
```

Le seul ajout supplémentaire est que `tp_dealloc` doit effacer toute référence faible (en appelant `PyObject_ClearWeakRefs()`) si le champ est non `NULL` :

```
static void
Trivial_dealloc(TrivialObject *self)
{
    /* Clear weakrefs first before calling any destructors */
    if (self->weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) self);
    /* ... remainder of destruction code omitted for brevity ... */
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

## 2.3.7 Plus de suggestions

Pour savoir comment mettre en œuvre une méthode spécifique pour votre nouveau type de données, téléchargez le code source *CPython*. Allez dans le répertoire `Objects`, puis cherchez dans les fichiers sources *C* la fonction `tp_` plus la fonction que vous voulez (par exemple, `tp_richcompare`). Vous trouverez des exemples de la fonction que vous voulez implémenter.

Lorsque vous avez besoin de vérifier qu'un objet est une instance concrète du type que vous implémentez, utilisez la fonction `PyObject_TypeCheck()`. Voici un exemple de son utilisation :

```
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

Voir aussi :

Télécharger les versions sources de *CPython*. <https://www.python.org/downloads/source/>

Le projet *CPython* sur *GitHub*, où se trouve le code source *CPython*. <https://github.com/python/cpython>

## 2.4 Construire des extensions C et C++

Une extension C pour *CPython* est une bibliothèque partagée (Un `.so` sur Linux, un `.pyd` sur Windows), qui expose une *fonction d'initialisation*.

Pour pouvoir être importée, la bibliothèque partagée doit pouvoir être trouvée dans `PYTHONPATH`, et doit porter le nom du module, avec l'extension appropriée. En utilisant *distutils*, le nom est généré automatiquement.

La fonction d'initialisation doit avoir le prototype :

`PyObject* PyInit_modulename (void)`

Elle doit donner soit un module entièrement initialisé, soit une instance de `PyModuleDef`. Voir *initializing-modules* pour plus de détails.

Pour les modules dont les noms sont entièrement en ASCII, la fonction doit être nommée `PyInit_<modulename>`, dont `<modulename>` est remplacé par le nom du module. En utilisant *multi-phase-initialization*, il est possible d'utiliser des noms de modules comptant des caractères non ASCII. Dans ce cas, le nom de la fonction d'initialisation est `PyInitU_<modulename>`, où `modulename` est encodé avec l'encodage *punyencode* de Python, dont les tirets sont remplacés par des tirets-bas. En Python ça donne :

```
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-' , b'_')
    return b'PyInit' + suffix
```

Il est possible d'exporter plusieurs modules depuis une seule bibliothèque partagée en définissant plusieurs fonctions d'initialisation. Cependant pour les importer, un lien symbolique doit être créé pour chacun, ou un *importer* personnalisé, puisque par défaut seule la fonction correspondant au nom du fichier est cherchée. Voir le chapitre *"Multiple modules in one library"* dans la **PEP 489** pour plus d'informations.

## 2.4.1 Construire les extensions C et C++ avec *distutils*

Des modules d'extension peuvent être construits avec *distutils*, qui est inclus dans Python. Puisque *distutils* gère aussi la création de paquets binaires, les utilisateurs n'auront pas nécessairement besoin ni d'un compilateur ni de *distutils* pour installer l'extension.

Un paquet *distutils* contient un script `setup.py`. C'est un simple fichier Python, ressemblant dans la plupart des cas à :

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       ext_modules = [module1])
```

Avec ce `setup.py` et un fichier `demo.c`, lancer

```
python setup.py build
```

compilera `demo.c`, et produira un module d'extension nommé `demo` dans le dossier `build`. En fonction du système, le fichier du module peut se retrouver dans `build/lib.system`, et son nom peut être `demo.py` ou `demo.pyd`.

Dans le fichier `setup.py`, tout est exécuté en appelant la fonction `setup`. Elle prend un nombre variable d'arguments nommés, dont l'exemple précédent n'utilise qu'une partie. L'exemple précise des méta-informations pour construire les paquets, et définir le contenu du paquet. Normalement un paquet contient des modules additionnels, comme des modules sources, documentation, sous paquets, etc. Reférez-vous à la documentation de *distutils* dans `distutils-index` pour en apprendre plus sur les fonctionnalités de *distutils*. Cette section n'explique que la construction de modules d'extension.

Il est classique de pré-calculer les arguments à la fonction `setup()`, pour plus de lisibilité. Dans l'exemple ci-dessus, l'argument `ext_modules` à `setup()` est une liste de modules d'extension, chacun est une instance de la classe `Extension`. Dans l'exemple, l'instance définit une extension nommée `demo` construite par la compilation d'un seul fichier source `demo.c`.

Dans la plupart des cas, construire une extension est plus complexe à cause des bibliothèques et définitions de préprocesseurs dont la compilation pourrait dépendre. C'est ce qu'on remarque dans l'exemple plus bas.

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    define_macros = [('MAJOR_VERSION', '1'),
                                    ('MINOR_VERSION', '0')],
                    include_dirs = ['/usr/local/include'],
                    libraries = ['tcl83'],
                    library_dirs = ['/usr/local/lib'],
                    sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       author = 'Martin v. Loewis',
       author_email = 'martin@v.loewis.de',
       url = 'https://docs.python.org/extending/building',
       long_description = '''
This is really just a demo package.
''',
       ext_modules = [module1])
```



Dans cet exemple, la fonction `setup()` est appelée avec quelques autres méta-informations, ce qui est recommandé pour distribuer des paquets. En ce qui concerne l'extension, sont définis quelques macros préprocesseur, dossiers pour les en-têtes et bibliothèques. En fonction du compilateur, *distutils* peut donner ces informations de manière différente. Par exemple, sur Unix, ça peut ressembler aux commandes

```
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION=1 -DMINOR_
↪VERSION=0 -I/usr/local/include -I/usr/local/include/python2.2 -c demo.c -o build/
↪temp.linux-i686-2.2/demo.o

gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib -ltcl83 -o build/lib.
↪linux-i686-2.2/demo.so
```

Ces lignes ne sont qu'à titre d'exemple, les utilisateurs de *distutils* doivent avoir confiance en *distutils* qui fera les appels correctement.

## 2.4.2 Distribuer vos modules d'extension

Lorsqu'une extension a été construite avec succès, il existe trois moyens de l'utiliser.

Typiquement, les utilisateurs vont vouloir installer le module, ils le font en exécutant

```
python setup.py install
```

Les mainteneurs de modules voudront produire des paquets source, pour ce faire ils exécuteront

```
python setup.py sdist
```

Dans certains cas, des fichiers supplémentaires doivent être inclus dans une distribution source : c'est possible via un fichier `MANIFEST.in`, c.f. `manifest`.

Si la distribution source a été construite avec succès, les mainteneurs peuvent aussi créer une distribution binaire. En fonction de la plateforme, une des commandes suivantes peut être utilisée.

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```

## 2.5 Construire des extensions C et C++ sur Windows

Cette page explique rapidement comment créer un module d'extension Windows pour Python en utilisant Microsoft Visual C++, et donne plus d'informations contextuelles sur son fonctionnement. Le texte explicatif est utile tant pour le développeur Windows qui apprend à construire des extensions Python que pour le développeur Unix souhaitant produire des logiciels pouvant être construits sur Unix et Windows.

Les auteurs de modules sont invités à utiliser l'approche *distutils* pour construire des modules d'extension, au lieu de celle décrite dans cette section. Vous aurez toujours besoin du compilateur C utilisé pour construire Python ; typiquement Microsoft Visual C++.

---

**Note :** Cette page mentionne plusieurs noms de fichiers comprenant un numéro de version Python encodé. Ces noms de fichiers sont construits sous le format de version `XY` ; en pratique, 'X' représente le numéro de version majeure et 'Y' représente le numéro de version mineure de la version Python avec laquelle vous travaillez. Par exemple, si vous utilisez Python 2.2.1, `XY` correspond à 22.

---

### 2.5.1 Une approche "recette de cuisine"

Il y a deux approches lorsque l'on construit des modules d'extension sur Windows, tout comme sur Unix : utiliser le paquet `distutils` pour contrôler le processus de construction, ou faire les choses manuellement. L'approche `distutils` fonctionne bien pour la plupart des extensions ; la documentation pour utiliser `distutils` pour construire et emballer les modules d'extension est disponible dans `distutils-index`. Si vous considérez que vous avez réellement besoin de faire les choses manuellement, il pourrait être enrichissant d'étudier le fichier de projet `winsound` pour le module de la bibliothèque standard.

### 2.5.2 Différences entre Unix et Windows

Unix et Windows utilisent des paradigmes complètement différents pour le chargement du code pendant l'exécution. Avant d'essayer de construire un module qui puisse être chargé dynamiquement, soyez conscient du mode de fonctionnement du système.

Sur Unix, un fichier objet partagé (`.so`) contient du code servant au programme, ainsi que les noms des fonctions et les données que l'on s'attend à trouver dans le programme. Quand le fichier est attaché au programme, toutes les références à ces fonctions et données dans le code du fichier sont modifiées pour pointer vers les localisations actuelles dans le programme où sont désormais placées les fonctions et données dans la mémoire. C'est tout simplement une opération de liaison.

Sur Windows, un fichier bibliothèque de liens dynamiques (`.dll`) n'a pas de références paresseuses. A la place, un accès aux fonctions ou données passe par une table de conversion. Cela est fait pour que le code DLL ne doive pas être réarrangé à l'exécution pour renvoyer à la mémoire du programme ; à la place, le code utilise déjà la table de conversion DLL, et cette table est modifiée à l'exécution pour pointer vers les fonctions et données.

Sur Unix, il n'y a qu'un type de bibliothèque de fichier (`.a`) qui contient du code venant de plusieurs fichiers objets (`.o`). Durant l'étape de liaison pour créer un fichier objet partagé (`.so`), le lieur peut informer qu'il ne sait pas où un identificateur est défini. Le lieur le cherchera dans les fichiers objet dans les bibliothèques ; s'il le trouve, il inclura tout le code provenant de ce fichier objet.

Sur Windows, il y a deux types de bibliothèques, une bibliothèque statique et une bibliothèque d'importation (toutes deux appelées `.lib`). Une bibliothèque statique est comme un fichier Unix `.a` ; elle contient du code pouvant être inclus si nécessaire. Une bibliothèque d'importation est uniquement utilisée pour rassurer le lieur qu'un certain identificateur est légal, et sera présent dans le programme quand la DLL est chargée. Comme ça le lieur utilise les informations provenant de la bibliothèque d'importation pour construire la table de conversion pour utiliser les identificateurs qui ne sont pas inclus dans la DLL. Quand une application ou une DLL est liée, une bibliothèque d'importation peut être générée, qui devra être utilisée pour toutes les futures DLL dépendantes aux symboles provenant de l'application ou de la DLL.

Supposons que vous construisez deux modules de chargement dynamiques, B et C, qui ne devraient pas partager un autre bloc de code avec A. Sur Unix, vous ne transmettez pas A.`a` au lieur pour B.`so` et C.`so` ; cela le ferait être inclus deux fois, pour que B et C aient chacun leur propre copie. Sur Windows, construire A.`dll` construira aussi A.`lib`. Vous transmettez A.`lib` au lieur pour B et C. A.`lib` ne contient pas de code ; il contient uniquement des informations qui seront utilisées lors de l'exécution pour accéder au code de A.

Sur Windows, utiliser une bibliothèque d'importation est comme utiliser `import spam` ; cela vous donne accès aux noms des spams, mais ne crée pas de copie séparée. Sur Unix, se lier à une bibliothèque est plus comme `from spam import *` ; cela crée une copie séparée.

### 2.5.3 Utiliser les DLL en pratique

Le Python de Windows est construit en Microsoft Visual C++; utiliser d'autres compilateurs pourrait fonctionner, ou pas (cependant Borland a l'air de fonctionner). Le reste de cette section est spécifique à MSVC++.

Lorsque vous créez des DLL sur Windows, vous devez transmettre `pythonXY.lib` au lieu. Pour construire deux DLL, `spam` et `ni` (qui utilisent des fonctions C trouvées dans `spam`), vous pouvez utiliser ces commandes :

```
cl /LD /I/python/include spam.c ../libs/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ../libs/pythonXY.lib
```

La première commande a créé trois fichiers : `spam.obj`, `spam.dll` et `spam.lib`. `Spam.dll` ne contient pas de fonctions Python (telles que `PyArg_ParseTuple()`), mais il sait comment trouver le code Python grâce à `pythonXY.lib`.

La seconde commande a créé `ni.dll` (et `.obj` et `.lib`), qui sait comment trouver les fonctions nécessaires dans `spam`, ainsi qu'à partir de l'exécutable Python.

Chaque identificateur n'est pas exporté vers la table de conversion. Si vous voulez que tout autre module (y compris Python) soit capable de voir vos identificateurs, vous devez préciser `_declspec(dllexport)`, comme dans `void _declspec(dllexport) initspam(void)` ou `PyObject _declspec(dllexport) *NiGetSpamData(void)`.

Developer Studio apportera beaucoup de bibliothèques d'importation dont vous n'avez pas vraiment besoin, augmentant d'environ *100ko* votre exécutable. Pour s'en débarrasser, allez dans les Paramètres du Projet, onglet Lien, pour préciser *ignorer les bibliothèques par défaut*. Et la `msvcrtxx.lib` correcte à la liste des bibliothèques.



---

## Intégrer l'interpréteur CPython dans une plus grande application

---

Parfois, plutôt que de créer une extension qui s'exécute dans l'interpréteur Python comme application principale, il est préférable d'intégrer l'interpréteur Python dans une application plus large. Cette section donne quelques informations nécessaires au succès de cette opération.

### 3.1 Intégrer Python dans une autre application

Les chapitres précédents couvraient l'extension de Python, c'est-à-dire, comment enrichir une fonctionnalité de Python en y attachant une bibliothèque de fonctions C. C'est aussi possible dans l'autre sens : enrichir vos applications C/C++ en y intégrant Python. Intégrer Python vous permet d'implémenter certaines fonctionnalités de vos applications en Python plutôt qu'en C ou C++. C'est utile dans de nombreux cas, un exemple serait de permettre aux utilisateurs d'adapter une application à leurs besoins en y écrivant des scripts Python. Vous pouvez aussi l'utiliser vous même si certaines fonctionnalités peuvent être rédigées plus facilement en Python.

Intégrer et étendre Python sont des tâches presque identiques. La différence est qu'en étendant Python, le programme principal reste l'interpréteur Python, alors qu'en intégrant Python le programme principal peut ne rien à voir avec Python. C'est simplement quelques parties du programme qui appellent l'interpréteur Python pour exécuter un peu de code Python.

En intégrant Python, vous fournissez le programme principal. L'une de ses tâches sera d'initialiser l'interpréteur. Au minimum vous devrez appeler `Py_Initialize()`. Il est possible, avec quelques appels supplémentaires, de passer des options à Python. Ensuite vous pourrez appeler l'interpréteur depuis n'importe quelle partie de votre programme.

Il existe différents moyens d'appeler l'interpréteur : vous pouvez donner une chaîne contenant des instructions Python à `PyRun_SimpleString()`, ou vous pouvez donner un pointeur de fichier *stdio* et un nom de fichier (juste pour nommer les messages d'erreur) à `PyRunSimpleFile()`. Vous pouvez aussi appeler les API de bas niveau décrites dans les chapitres précédents pour construire et utiliser des objets Python.

**Voir aussi :**

**c-api-index** Les détails sur l'interface entre Python et le C sont donnés dans ce manuel. Pléthore d'informations s'y trouvent.

### 3.1.1 Intégration de très haut niveau

La manière la plus simple d'intégrer Python est d'utiliser une interface de très haut niveau. Cette interface a pour but d'exécuter un script Python sans avoir à interagir avec directement. C'est utile, par exemple, pour effectuer une opération sur un fichier.

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    Py_SetProgramName(program); /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                      "print('Today is', ctime(time()))\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    PyMem_RawFree(program);
    return 0;
}
```

C'est la fonction `Py_SetProgramName()` qui devrait être appelée en premier, avant `Py_Initialize()`, afin d'informer l'interpréteur des chemins vers ses bibliothèques. Ensuite l'interpréteur est initialisé par `Py_Initialize()`, suivi de l'exécution de Python codé en dur affichant la date et l'heure, puis, l'appel à `Py_FinalizeEx()` éteint l'interpréteur, engendrant ainsi la fin du programme. Dans un vrai programme, vous pourriez vouloir lire le script Python depuis une autre source, peut être depuis un éditeur de texte, un fichier, ou une base de donnée. Récupérer du code Python depuis un fichier se fait via `PyRun_SimpleFile()`, qui vous économise le travail d'allouer de la mémoire et de charger le contenu du fichier.

### 3.1.2 Au-delà de l'intégration de haut niveau : survol

L'interface de haut niveau vous permet d'exécuter n'importe quel morceau de code Python depuis votre application, mais échanger des données est quelque peu alambiqué. Si c'est ce dont vous avez besoin, vous devez utiliser des appels de niveau plus bas. Il vous en coûtera plus de lignes de C à écrire, mais vous pourrez presque tout faire.

Il est à souligner qu'étendre ou intégrer Python revient à la louche au même, en dépit de la différence d'intention. La plupart des sujets parcourus dans les chapitres précédents sont toujours valides. Pour le prouver, regardez ce qu'un code d'extension de Python vers C fait réellement :

1. Convertir des valeurs de Python vers le C,
2. Appeler une fonction C en utilisant les valeurs converties, et
3. Convertir les résultats de l'appel à la fonction C pour Python.

Lors de l'intégration de Python, le code de l'interface fait :

1. Convertir les valeurs depuis le C vers Python,
2. Effectuer un appel de fonction de l'interface Python en utilisant les valeurs converties, et
3. Convertir les valeurs de l'appel Python pour le C.

Tel que vous le voyez, les conversions sont simplement inversées pour s'adapter aux différentes directions de transfert inter-langage. La seule différence est la fonction que vous appelez entre les deux conversions de données. Lors de l'extension, vous appelez une fonction C, lors de l'intégration vous appelez une fonction Python.

Ce chapitre ne couvrira pas la conversion des données de Python vers le C ni l'inverse. Aussi, un usage correct des références, ainsi que savoir gérer les erreurs sont considérés acquis. Ces aspects étant identiques à l'extension de l'interpréteur, vous pouvez vous référer aux chapitres précédents.

### 3.1.3 Intégration pure

L'objectif du premier programme est d'exécuter une fonction dans un script Python. Comme dans la section à propos des interfaces de haut niveau, l'interpréteur n'interagit pas directement avec l'application (mais le fera dans la section suivante).

Le code pour appeler une fonction définie dans un script Python est :

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize();
    pName = PyUnicode_DecodeFSDefault(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
```

(suite sur la page suivante)

(suite de la page précédente)

```

        printf("Result of call: %ld\n", PyLong_AsLong(pValue));
        Py_DECREF(pValue);
    }
    else {
        Py_DECREF(pFunc);
        Py_DECREF(pModule);
        PyErr_Print();
        fprintf(stderr, "Call failed\n");
        return 1;
    }
}
else {
    if (PyErr_Occurred())
        PyErr_Print();
    fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
}
Py_XDECREF(pFunc);
Py_DECREF(pModule);
}
else {
    PyErr_Print();
    fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
    return 1;
}
if (Py_FinalizeEx() < 0) {
    return 120;
}
return 0;
}

```

Ce code charge un script Python en utilisant `argv[1]`, et appelle une fonction dont le nom est dans `argv[2]`. Ses arguments entiers sont les autres valeurs de `argv`. Si vous *compilez et liez* ce programme (appelons l'exécutable **call**), et l'appeliez pour exécuter un script Python, tel que :

```

def multiply(a,b):
    print("Will compute", a, "times", b)
    c = 0
    for i in range(0, a):
        c = c + b
    return c

```

alors, le résultat sera :

```

$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6

```

Bien que le programme soit plutôt gros pour ses fonctionnalités, la plupart du code n'est que conversion de données entre Python et C, aussi que pour rapporter les erreurs. La partie intéressante, qui concerne l'intégration de Python débute par

```

Py_Initialize();
pName = PyUnicode_DecodeFSDefault(argv[1]);
/* Error checking of pName left out */
pModule = PyImport_Import(pName);

```

Après avoir initialisé l'interpréteur, le script est chargé en utilisant `PyImport_Import()`. Cette fonction prend une chaîne Python pour argument, elle-même construite en utilisant la fonction de conversion



```
PyUnicode_FromString().
```

```
pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc is a new reference */

if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);
```

Une fois le script chargé, le nom recherché est obtenu en utilisant `PyObject_GetAttrString()`. Si le nom existe, et que l'objet récupéré peut être appelé, vous pouvez présumer sans risque que c'est une fonction. Le programme continue, classiquement, par la construction de  $n$ -uplet d'arguments. L'appel à la fonction Python est alors effectué avec :

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

Après l'exécution de la fonction, `pValue` est soit `NULL`, soit une référence sur la valeur donnée par la fonction. Assurez-vous de libérer la référence après avoir utilisé la valeur.

### 3.1.4 Étendre un Python intégré

Jusqu'à présent, l'interpréteur Python intégré n'avait pas accès aux fonctionnalités de l'application elle-même. L'API Python le permet en étendant l'interpréteur intégré. Autrement dit, l'interpréteur intégré est étendu avec des fonctions fournies par l'application. Bien que cela puisse sembler complexe, ce n'est pas si dur. Il suffit d'oublier que l'application démarre l'interpréteur Python, au lieu de cela, voyez l'application comme un ensemble de fonctions, et rédigez un peu de code pour exposer ces fonctions à Python, tout comme vous écririez une extension Python normale. Par exemple :

```
static int numargs=0;

/* Return the number of arguments of the application command line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return PyLong_FromLong(numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
     "Return the number of arguments received by the process."},
    {NULL, NULL, 0, NULL}
};

static PyModuleDef EmbModule = {
    PyModuleDef_HEAD_INIT, "emb", NULL, -1, EmbMethods,
    NULL, NULL, NULL, NULL
};

static PyObject*
PyInit_emb(void)
{
    return PyModule_Create(&EmbModule);
}
```

Insérez le code ci-dessus juste avant la fonction `main()`. Ajoutez aussi les deux instructions suivantes avant l'appel à `Py_Initialize()` :

```
numargs = argc;
PyImport_AppendInittab("emb", &PyInit_emb);
```

Ces deux lignes initialisent la variable `numarg`, et rend la fonction `emb.numargs()` accessible à l'interpréteur intégré. Avec ces ajouts, le script Python peut maintenant faire des choses comme

```
import emb
print("Number of arguments", emb.numargs())
```

Dans un cas réel, les méthodes exposeraient une API de l'application à Python.

### 3.1.5 Intégrer Python dans du C++

Il est aussi possible d'intégrer Python dans un programme en C++, la manière exacte dont cela se fait dépend de détails du système C++ utilisé. En général vous écrirez le programme principal en C++, utiliserez un compilateur C++ pour compiler et lier votre programme. Il n'y a pas besoin de recompiler Python en utilisant C++.

### 3.1.6 Compiler et Lier en environnement Unix ou similaire

Ce n'est pas évident de trouver les bonnes options à passer au compilateur (et *linker*) pour intégrer l'interpréteur Python dans une application, Python ayant besoin de charger des extensions sous forme de bibliothèques dynamiques en C (des `.so`) pour se lier avec.

Pour trouver les bonnes options de compilateur et *linker*, vous pouvez exécuter le script `python(X.Y)-config` généré durant l'installation (un script `python3-config` peut aussi être disponible). Ce script a quelques options, celles-ci vous seront utiles :

- `pythonX.Y-config --cflags` vous donnera les options recommandées pour compiler :

```
$ /opt/bin/python3.4-config --cflags
-I/opt/include/python3.4m -I/opt/include/python3.4m -DNDEBUG -g -fwrapv -O3 -
-Wall -Wstrict-prototypes
```

- `pythonX.Y-config --ldflags` vous donnera les drapeaux recommandés lors de l'édition de lien :

```
$ /opt/bin/python3.4-config --ldflags
-L/opt/lib/python3.4/config-3.4m -lpthread -ldl -lutil -lm -lpthread -Xlinker -
-export-dynamic
```

**Note :** Pour éviter la confusion entre différentes installations de Python, (et plus spécialement entre celle de votre système et votre version compilée), il est recommandé d'utiliser un chemin absolu vers `pythonX.Y-config`, comme dans l'exemple précédent.

Si cette procédure ne fonctionne pas pour vous (il n'est pas garanti qu'elle fonctionne pour toutes les plateformes Unix, mais nous traiteront volontiers les rapports de bugs), vous devrez lire la documentation de votre système sur la liaison dynamique (*dynamic linking*) et / ou examiner le Makefile de Python (utilisez `sysconfig.get_makefile_filename()` pour trouver son emplacement) et les options de compilation. Dans ce cas, le module `sysconfig` est un outil utile pour extraire automatiquement les valeurs de configuration que vous voudrez combiner ensemble. Par exemple :

```
>>> import sysconfig
>>> sysconfig.get_config_var('LIBS')
'-lpthread -ldl -lutil'
>>> sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

>>> L'invite de commande utilisée par défaut dans l'interpréteur interactif. On la voit souvent dans des exemples de code qui peuvent être exécutés interactivement dans l'interpréteur.

... Peut faire référence à :

- L'invite de commande utilisée par défaut dans l'interpréteur interactif lorsqu'on entre un bloc de code indenté, dans des délimiteurs fonctionnant par paires (parenthèses, crochets, accolades, triple guillemets), ou après un avoir spécifié un décorateur.
- La constante `Ellipsis`.

**2to3** Outil qui essaie de convertir du code pour Python 2.x en code pour Python 3.x en gérant la plupart des incompatibilités qui peuvent être détectées en analysant la source et parcourant son arbre syntaxique.

`2to3` est disponible dans la bibliothèque standard sous le nom de `lib2to3`; un point d'entrée indépendant est fourni via `Tools/scripts/2to3`. Cf. `2to3-reference`.

**classe de base abstraite** Les classes de base abstraites (ABC, suivant l'abréviation anglaise *Abstract Base Class*) complètent le *duck-typing* en fournissant un moyen de définir des interfaces pour les cas où d'autres techniques comme `hasattr()` seraient inélégantes ou subtilement fausses (par exemple avec les méthodes magiques). Les ABC introduisent des sous-classes virtuelles qui n'héritent pas d'une classe mais qui sont quand même reconnues par `isinstance()` ou `issubclass()` (voir la documentation du module `abc`). Python contient de nombreuses ABC pour les structures de données (dans le module `collections.abc`), les nombres (dans le module `numbers`), les flux (dans le module `io`) et les chercheurs-chargeurs du système d'importation (dans le module `importlib.abc`). Vous pouvez créer vos propres ABC avec le module `abc`.

**annotation** Étiquette associée à une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour. Elle est utilisée par convention comme *type hint*.

Les annotations de variables locales ne sont pas accessibles au moment de l'exécution, mais les annotations de variables globales, d'attributs de classe et de fonctions sont stockées dans l'attribut spécial `__annotations__` des modules, classes et fonctions, respectivement.

Voir *variable annotation*, *function annotation*, **PEP 484** et **PEP 526**, qui décrivent cette fonctionnalité.

**argument** Valeur, donnée à une *fonction* ou à une *méthode* lors de son appel. Il existe deux types d'arguments :

- *argument nommé* : un argument précédé d'un identifiant (comme `name=`) ou un dictionnaire précédé de `**`, lors d'un appel de fonction. Par exemple, 3 et 5 sont tous les deux des arguments nommés dans l'appel à `complex()` ici :

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argument positionnel* : un argument qui n'est pas nommé. Les arguments positionnels apparaissent au début de la liste des arguments, ou donnés sous forme d'un *itérable* précédé par \*. Par exemple, 3 et 5 sont tous les deux des arguments positionnels dans les appels suivants :

```
complex(3, 5)
complex(*(3, 5))
```

Les arguments se retrouvent dans le corps de la fonction appelée parmi les variables locales. Voir la section `calls` à propos des règles dictant cette affectation. Syntaxiquement, toute expression est acceptée comme argument, et c'est la valeur résultante de l'expression qui sera affectée à la variable locale.

Voir aussi *parameter* dans le glossaire, la question Différence entre argument et paramètre de la FAQ et la **PEP 362**.

**gestionnaire de contexte asynchrone** (*asynchronous context manager* en anglais) Objet contrôlant l'environnement à l'intérieur d'une instruction `with` en définissant les méthodes `__aenter__()` et `__aexit__()`. A été introduit par la **PEP 492**.

**générateur asynchrone** Fonction qui renvoie un *asynchronous generator iterator*. Cela ressemble à une coroutine définie par `async def`, sauf qu'elle contient une ou des expressions `yield` produisant ainsi une série de valeurs utilisables dans une boucle `async for`.

Générateur asynchrone fait généralement référence à une fonction, mais peut faire référence à un *itérateur de générateur asynchrone* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser l'ensemble des termes lève l'ambiguïté.

Un générateur asynchrone peut contenir des expressions `await` ainsi que des instructions `async for`, et `async with`.

**itérateur de générateur asynchrone** Objet créé par une fonction *asynchronous generator*.

C'est un *asynchronous iterator* qui, lorsqu'il est appelé via la méthode `__anext__()` renvoie un objet *awaitable* qui exécute le corps de la fonction du générateur asynchrone jusqu'au prochain `yield`.

Chaque `yield` suspend temporairement l'exécution, en gardant en mémoire l'endroit et l'état de l'exécution (ce qui inclut les variables locales et les `try` en cours). Lorsque l'exécution de l'itérateur de générateur asynchrone reprend avec un nouvel *awaitable* renvoyé par `__anext__()`, elle repart de là où elle s'était arrêtée. Voir la **PEP 492** et la **PEP 525**.

**itérable asynchrone** Objet qui peut être utilisé dans une instruction `async for`. Sa méthode `__aiter__()` doit renvoyer un *asynchronous iterator*. A été introduit par la **PEP 492**.

**itérateur asynchrone** Objet qui implémente les méthodes `__aiter__()` et `__anext__()`. `__anext__()` doit renvoyer un objet *awaitable*. Tant que la méthode `__anext__()` produit des objets *awaitable*, le `async for` appelant les consomme. L'itérateur asynchrone lève une exception `StopAsyncIteration` pour signifier la fin de l'itération. A été introduit par la **PEP 492**.

**attribut** Valeur associée à un objet et désignée par son nom via une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, il sera référencé par *o.a*.

**awaitable** Objet pouvant être utilisé dans une expression `await`. Ce peut être une *coroutine* ou un objet avec une méthode `__await__()`. Voir aussi la **PEP 492**.

**BDFL** Dictateur bienveillant à vie (*Benevolent Dictator For Life* en anglais). Pseudonyme de Guido van Rossum, le créateur de Python.

**fichier binaire** Un *file object* capable de lire et d'écrire des *bytes-like objects*. Des fichiers binaires sont, par exemple, les fichiers ouverts en mode binaire ('rb', 'wb', ou 'rb+'), `sys.stdin.buffer`, `sys.stdout.buffer`, les instances de `io.BytesIO` ou de `gzip.GzipFile`.

Consultez *fichier texte*, un objet fichier capable de lire et d'écrire des objets `str`.

**objet octet-compatible** Un objet gérant les *bufferobjects* et pouvant exporter un tampon (*buffer* en anglais) *C-continuous*. Cela inclut les objets `bytes`, `bytearray` et `array.array`, ainsi que beaucoup d'objets

`memoryview`. Les objets bytes-compatibles peuvent être utilisés pour diverses opérations sur des données binaires, comme la compression, la sauvegarde dans un fichier binaire ou l'envoi sur le réseau.

Certaines opérations nécessitent de travailler sur des données binaires variables. La documentation parle de ceux-ci comme des *read-write bytes-like objects*. Par exemple, `bytearray` ou une `memoryview` d'un `bytearray` en font partie. D'autres opérations nécessitent de travailler sur des données binaires stockées dans des objets immuables (*"read-only bytes-like objects"*), par exemples `bytes` ou `memoryview` d'un objet `byte`.

**code intermédiaire (*bytecode*)** Le code source, en Python, est compilé en un code intermédiaire (*bytecode* en anglais), la représentation interne à CPython d'un programme Python. Le code intermédiaire est mis en cache dans un fichier `.pyc` de manière à ce qu'une seconde exécution soit plus rapide (la compilation en code intermédiaire a déjà été faite). On dit que ce *langage intermédiaire* est exécuté sur une *virtual machine* qui exécute des instructions machine pour chaque instruction du code intermédiaire. Notez que le code intermédiaire n'a pas vocation à fonctionner sur différentes machines virtuelles Python ou à être stable entre différentes versions de Python.

La documentation du module `dis` fournit une liste des instructions du code intermédiaire.

**fonction de rappel** Une sous-fonction passée en argument pour être exécutée plus tard.

**classe** Modèle pour créer des objets définis par l'utilisateur. Une définition de classe (*class*) contient normalement des définitions de méthodes qui agissent sur les instances de la classe.

**variable de classe** Une variable définie dans une classe et destinée à être modifiée uniquement au niveau de la classe (c'est-à-dire, pas dans une instance de la classe).

**coercition** Conversion implicite d'une instance d'un type vers un autre lors d'une opération dont les deux opérandes doivent être de même type. Par exemple `int(3.15)` convertit explicitement le nombre à virgule flottante en nombre entier 3. Mais dans l'opération `3 + 4.5`, les deux opérandes sont d'un type différent (un entier et un nombre à virgule flottante), alors qu'ils doivent avoir le même type pour être additionnés (sinon une exception `TypeError` serait levée). Sans coercition, tous les opérandes, même de types compatibles, devraient être convertis (on parle aussi de *cast*) explicitement par le développeur, par exemple `float(3) + 4.5` au lieu du simple `3 + 4.5`.

**nombre complexe** Extension des nombres réels familiers, dans laquelle tous les nombres sont exprimés sous la forme d'une somme d'une partie réelle et d'une partie imaginaire. Les nombres imaginaires sont les nombres réels multipliés par l'unité imaginaire (la racine carrée de  $-1$ , souvent écrite *i* en mathématiques ou *j* par les ingénieurs). Python comprend nativement les nombres complexes, écrits avec cette dernière notation : la partie imaginaire est écrite avec un suffixe *j*, exemple, `3+1j`. Pour utiliser les équivalents complexes de `math`, utilisez `cmath`. Les nombres complexes sont un concept assez avancé en mathématiques. Si vous ne connaissez pas ce concept, vous pouvez tranquillement les ignorer.

**gestionnaire de contexte** Objet contrôlant l'environnement à l'intérieur d'un bloc `with` en définissant les méthodes `__enter__()` et `__exit__()`. Consultez la [PEP 343](#).

**variable de contexte** Une variable qui peut avoir des valeurs différentes en fonction de son contexte. Cela est similaire au stockage par fil d'exécution (*Thread Local Storage* en anglais) dans lequel chaque fil d'exécution peut avoir une valeur différente pour une variable. Toutefois, avec les variables de contexte, il peut y avoir plusieurs contextes dans un fil d'exécution et l'utilisation principale pour les variables de contexte est de garder une trace des variables dans les tâches asynchrones concourantes. Voir `contextvars`.

**contigu** Un tampon (*buffer* en anglais) est considéré comme contigu s'il est soit *C-contigu* soit *Fortran-contigu*. Les tampons de dimension zéro sont C-contigus et Fortran-contigus. Pour un tableau à une dimension, ses éléments doivent être placés en mémoire l'un à côté de l'autre, dans l'ordre croissant de leur indice, en commençant à zéro. Pour qu'un tableau multidimensionnel soit C-contigu, le dernier indice doit être celui qui varie le plus rapidement lors du parcours de ses éléments dans l'ordre de leur adresse mémoire. À l'inverse, dans les tableaux Fortran-contigu, c'est le premier indice qui doit varier le plus rapidement.

**coroutine** Les coroutines sont une forme généralisée des fonctions. On entre dans une fonction en un point et on en sort en un autre point. On peut entrer, sortir et reprendre l'exécution d'une coroutine en plusieurs points. Elles peuvent être implémentées en utilisant l'instruction `async def`. Voir aussi la [PEP 492](#).

**fonction coroutine** Fonction qui renvoie un objet *coroutine*. Une fonction coroutine peut être définie par l'instruction `async def` et peut contenir les mots clés `await`, `async` `for` ainsi que `async with`. A été introduit par la [PEP 492](#).

**CPython** L'implémentation canonique du langage de programmation Python, tel que distribué sur [python.org](https://python.org). Le terme "CPython" est utilisé dans certains contextes lorsqu'il est nécessaire de distinguer cette implémentation des autres comme *Jython* ou *IronPython*.

**décorateur** Fonction dont la valeur de retour est une autre fonction. Un décorateur est habituellement utilisé pour transformer une fonction via la syntaxe `@wrapper`, dont les exemples typiques sont : `classmethod()` et `staticmethod()`.

La syntaxe des décorateurs est simplement du sucre syntaxique, les définitions des deux fonctions suivantes sont sémantiquement équivalentes :

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Quoique moins fréquemment utilisé, le même concept existe pour les classes. Consultez la documentation définitions de fonctions et définitions de classes pour en savoir plus sur les décorateurs.

**descripteur** N'importe quel objet définissant les méthodes `__get__()`, `__set__()`, ou `__delete__()`. Lorsque l'attribut d'une classe est un descripteur, son comportement spécial est déclenché lors de la recherche des attributs. Normalement, lorsque vous écrivez `a.b` pour obtenir, affecter ou effacer un attribut, Python recherche l'objet nommé `b` dans le dictionnaire de la classe de `a`. Mais si `b` est un descripteur, c'est la méthode de ce descripteur qui est alors appelée. Comprendre les descripteurs est requis pour avoir une compréhension approfondie de Python, ils sont la base de nombre de ses caractéristiques notamment les fonctions, méthodes, propriétés, méthodes de classes, méthodes statiques et les références aux classes parentes.

Pour plus d'informations sur les méthodes des descripteurs, consultez `descriptors` ou le guide pour l'utilisation des descripteurs.

**dictionnaire** Structure de donnée associant des clés à des valeurs. Les clés peuvent être n'importe quel objet possédant les méthodes `__hash__()` et `__eq__()`. En Perl, les dictionnaires sont appelés "*hash*".

**dictionnaire en compréhension (ou dictionnaire en intension)** Écriture concise pour traiter tout ou partie des éléments d'un itérable et renvoyer un dictionnaire contenant les résultats. `results = {n: n ** 2 for n in range(10)}` génère un dictionnaire contenant des clés `n` liée à leur valeurs `n ** 2`. Voir `comprehensions`.

**vue de dictionnaire** Objets retournés par les méthodes `dict.keys()`, `dict.values()` et `dict.items()`. Ils fournissent des vues dynamiques des entrées du dictionnaire, ce qui signifie que lorsque le dictionnaire change, la vue change. Pour transformer une vue en vraie liste, utilisez `list(dictview)`. Voir `dict-views`.

**docstring (chaîne de documentation)** Première chaîne littérale qui apparaît dans l'expression d'une classe, fonction, ou module. Bien qu'ignorée à l'exécution, elle est reconnue par le compilateur et placée dans l'attribut `__doc__` de la classe, de la fonction ou du module. Comme cette chaîne est disponible par introspection, c'est l'endroit idéal pour documenter l'objet.

**duck-typing** Style de programmation qui ne prend pas en compte le type d'un objet pour déterminer s'il respecte une interface, mais qui appelle simplement la méthode ou l'attribut (*Si ça a un bec et que ça cancanne, ça doit être un canard*, *duck* signifie canard en anglais). En se concentrant sur les interfaces plutôt que les types, du code bien construit améliore sa flexibilité en autorisant des substitutions polymorphiques. Le *duck-typing* évite de vérifier les types via `type()` ou `isinstance()`, Notez cependant que le *duck-typing* peut travailler de pair avec les *classes de base abstraites*. À la place, le *duck-typing* utilise plutôt `hasattr()` ou la programmation *EAFP*.

**EAFP** Il est plus simple de demander pardon que demander la permission (*Easier to Ask for Forgiveness than Permission* en anglais). Ce style de développement Python fait l'hypothèse que le code est valide et traite les exceptions si cette hypothèse s'avère fausse. Ce style, propre et efficace, est caractérisé par la présence de beaucoup de mots clés `try` et `except`. Cette technique de programmation contraste avec le style *LBYL* utilisé couramment dans les langages tels que C.



**expression** Suite logique de termes et chiffres conformes à la syntaxe Python dont l'évaluation fournit une valeur. En d'autres termes, une expression est une suite d'éléments tels que des noms, opérateurs, littéraux, accès d'attributs, méthodes ou fonctions qui aboutissent à une valeur. Contrairement à beaucoup d'autres langages, les différentes constructions du langage ne sont pas toutes des expressions. On trouve également des *instructions* qui ne peuvent pas être utilisées comme expressions, tel que `while`. Les affectations sont également des instructions et non des expressions.

**module d'extension** Module écrit en C ou C++, utilisant l'API C de Python pour interagir avec Python et le code de l'utilisateur.

**f-string** Chaîne littérale préfixée de 'f' ou 'F'. Les "f-strings" sont un raccourci pour formatted string literals. Voir la [PEP 498](#).

**objet fichier** Objet exposant une ressource via une API orientée fichier (avec les méthodes `read()` ou `write()`). En fonction de la manière dont il a été créé, un objet fichier peut interfacer l'accès à un fichier sur le disque ou à un autre type de stockage ou de communication (typiquement l'entrée standard, la sortie standard, un tampon en mémoire, un connecteur réseau...). Les objets fichiers sont aussi appelés *file-like-objects* ou *streams*.

Il existe en réalité trois catégories de fichiers objets : les *fichiers binaires* bruts, les *fichiers binaires* avec tampon (*buffer*) et les *fichiers textes*. Leurs interfaces sont définies dans le module `io`. Le moyen le plus simple et direct de créer un objet fichier est d'utiliser la fonction `open()`.

**objet fichier-compatible** Synonyme de *objet fichier*.

**chercheur** Objet qui essaie de trouver un *chargeur* pour le module en cours d'importation.

Depuis Python 3.3, il existe deux types de chercheurs : les *chercheurs dans les méta-chemins* à utiliser avec `sys.meta_path`; les *chercheurs d'entrée dans path* à utiliser avec `sys.path_hooks`.

Voir les [PEP 302](#), [PEP 420](#) et [PEP 451](#) pour plus de détails.

**division entière** Division mathématique arrondissant à l'entier inférieur. L'opérateur de la division entière est `//`. Par exemple l'expression `11 // 4` vaut 2, contrairement à `11 / 4` qui vaut 2.75. Notez que `(-11) // 4` vaut -3 car l'arrondi se fait à l'entier inférieur. Voir la [PEP 328](#).

**fonction** Suite d'instructions qui renvoie une valeur à son appelant. On peut lui passer des *arguments* qui pourront être utilisés dans le corps de la fonction. Voir aussi *paramètre*, *méthode* et *fonction*.

**annotation de fonction** *annotation* d'un paramètre de fonction ou valeur de retour.

Les annotations de fonctions sont généralement utilisées pour des *indications de types* : par exemple, cette fonction devrait prendre deux arguments `int` et devrait également avoir une valeur de retour de type `int` :

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

L'annotation syntaxique de la fonction est expliquée dans la section *fonction*.

Voir *variable annotation* et [PEP 484](#), qui décrivent cette fonctionnalité.

**\_\_future\_\_** Pseudo-module que les développeurs peuvent utiliser pour activer de nouvelles fonctionnalités du langage qui ne sont pas compatibles avec l'interpréteur utilisé.

En important le module `__future__` et en affichant ses variables, vous pouvez voir à quel moment une nouvelle fonctionnalité a été rajoutée dans le langage et quand elle devient le comportement par défaut :

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

**ramasse-miettes** (*garbage collection* en anglais) Mécanisme permettant de libérer de la mémoire lorsqu'elle n'est plus utilisée. Python utilise un ramasse-miettes par comptage de référence et un ramasse-miettes cyclique capable de détecter et casser les références circulaires. Le ramasse-miettes peut être contrôlé en utilisant le module `gc`.

**générateur** Fonction qui renvoie un *itérateur de générateur*. Cela ressemble à une fonction normale, en dehors du fait qu'elle contient une ou des expressions `yield` produisant une série de valeurs utilisable dans une boucle *for* ou récupérées une à une via la fonction `next()`.

Fait généralement référence à une fonction générateur mais peut faire référence à un *itérateur de générateur* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser les termes complets lève l'ambiguïté.

**itérateur de générateur** Objet créé par une fonction *générateur*.

Chaque `yield` suspend temporairement l'exécution, en se rappelant l'endroit et l'état de l'exécution (y compris les variables locales et les `try` en cours). Lorsque l'itérateur de générateur reprend, il repart là où il en était (contrairement à une fonction qui prendrait un nouveau départ à chaque invocation).

**expression génératrice** Expression qui donne un itérateur. Elle ressemble à une expression normale, suivie d'une clause `for` définissant une variable de boucle, un intervalle et une clause `if` optionnelle. Toute cette expression génère des valeurs pour la fonction qui l'entoure :

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

**fonction générique** Fonction composée de plusieurs fonctions implémentant les mêmes opérations pour différents types. L'implémentation à utiliser est déterminée lors de l'appel par l'algorithme de répartition.

Voir aussi *single dispatch*, le décorateur `functools singledispatch()` et la **PEP 443**.

**type générique** Un *type* qui peut être paramétré; typiquement un conteneur comme une `list`. Utilisé pour les *indications de type* et les *annotations*.

Voir la **PEP 483** pour plus de détails, et `typing` ou alias générique de `type` pour ses utilisations.

**GIL** Voir *global interpreter lock*.

**verrou global de l'interpréteur** (*global interpreter lock* en anglais) Mécanisme utilisé par l'interpréteur *CPython* pour s'assurer qu'un seul fil d'exécution (*thread* en anglais) n'exécute le *bytecode* à la fois. Cela simplifie l'implémentation de *CPython* en rendant le modèle objet (incluant des parties critiques comme la classe native `dict`) implicitement protégé contre les accès concourants. Verrouiller l'interpréteur entier rend plus facile l'implémentation de multiples fils d'exécution (*multi-thread* en anglais), au détriment malheureusement de beaucoup du parallélisme possible sur les machines ayant plusieurs processeurs.

Cependant, certains modules d'extension, standards ou non, sont conçus de manière à libérer le GIL lorsqu'ils effectuent des tâches lourdes tel que la compression ou le hachage. De la même manière, le GIL est toujours libéré lors des entrées / sorties.

Les tentatives précédentes d'implémenter un interpréteur Python avec une granularité de verrouillage plus fine ont toutes échouées, à cause de leurs mauvaises performances dans le cas d'un processeur unique. Il est admis que corriger ce problème de performance induit mènerait à une implémentation beaucoup plus compliquée et donc plus coûteuse à maintenir.

**pyc utilisant le hachage** Un fichier de cache de code intermédiaire (*bytecode* en anglais) qui utilise le hachage plutôt que l'heure de dernière modification du fichier source correspondant pour déterminer sa validité. Voir *pyc-invalidation*.

**hachable** Un objet est *hachable* s'il a une empreinte (*hash*) qui ne change jamais (il doit donc implémenter une méthode `__hash__()`) et s'il peut être comparé à d'autres objets (avec la méthode `__eq__()`). Les objets hachables dont la comparaison par `__eq__` est vraie doivent avoir la même empreinte.

La hachabilité permet à un objet d'être utilisé comme clé de dictionnaire ou en tant que membre d'un ensemble (type *set*), car ces structures de données utilisent ce *hash*.

La plupart des types immuables natifs de Python sont hachables, mais les conteneurs muables (comme les listes ou les dictionnaires) ne le sont pas; les conteneurs immuables (comme les *n-uplets* ou les ensembles figés) ne sont hachables que si leurs éléments sont hachables. Les instances de classes définies par les utilisateurs sont hachables par défaut. Elles sont toutes considérées différentes (sauf avec elles-mêmes) et leur valeur de hachage est calculée à partir de leur `id()`.

**IDLE** Environnement de développement intégré pour Python. IDLE est un éditeur basique et un interpréteur livré avec la distribution standard de Python.

**immuable** Objet dont la valeur ne change pas. Les nombres, les chaînes et les *n-uplets* sont immuables. Ils ne peuvent être modifiés. Un nouvel objet doit être créé si une valeur différente doit être stockée. Ils jouent un rôle important quand une valeur de *hash* constante est requise, typiquement en clé de dictionnaire.

**chemin des importations** Liste de *entrées* dans lesquelles le *chercheur basé sur les chemins* cherche les modules à importer. Typiquement, lors d'une importation, cette liste vient de `sys.path`; pour les sous-paquets, elle peut aussi venir de l'attribut `__path__` du paquet parent.



**importing** Processus rendant le code Python d'un module disponible dans un autre.

**importateur** Objet qui trouve et charge un module, en même temps un *chercheur* et un *chargeur*.

**interactif** Python a un interpréteur interactif, ce qui signifie que vous pouvez écrire des expressions et des instructions à l'invite de l'interpréteur. L'interpréteur Python va les exécuter immédiatement et vous en présenter le résultat. Démarrez juste `python` (probablement depuis le menu principal de votre ordinateur). C'est un moyen puissant pour tester de nouvelles idées ou étudier de nouveaux modules (souvenez-vous de `help(x)`).

**interprété** Python est un langage interprété, en opposition aux langages compilés, bien que la frontière soit floue en raison de la présence d'un compilateur en code intermédiaire. Cela signifie que les fichiers sources peuvent être exécutés directement, sans avoir à compiler un fichier exécutable intermédiaire. Les langages interprétés ont généralement un cycle de développement / débogage plus court que les langages compilés. Cependant, ils s'exécutent généralement plus lentement. Voir aussi *interactif*.

**arrêt de l'interpréteur** Lorsqu'on lui demande de s'arrêter, l'interpréteur Python entre dans une phase spéciale où il libère graduellement les ressources allouées, comme les modules ou quelques structures de données internes. Il fait aussi quelques appels au *ramasse-miettes*. Cela peut déclencher l'exécution de code dans des destructeurs ou des fonctions de rappels de *weakrefs*. Le code exécuté lors de l'arrêt peut rencontrer des exceptions puisque les ressources auxquelles il fait appel sont susceptibles de ne plus fonctionner, (typiquement les modules des bibliothèques ou le mécanisme de *warning*).

La principale raison d'arrêt de l'interpréteur est que le module `__main__` ou le script en cours d'exécution a terminé de s'exécuter.

**itérable** Objet capable de renvoyer ses éléments un à un. Par exemple, tous les types séquence (comme `list`, `str`, et `tuple`), quelques autres types comme `dict`, *objets fichiers* ou tout objet d'une classe ayant une méthode `__iter__()` ou `__getitem__()` qui implémente la sémantique d'une *Sequence*.

Les itérables peuvent être utilisés dans des boucles `for` et à beaucoup d'autres endroits où une séquence est requise (`zip()`, `map()` ...). Lorsqu'un itérable est passé comme argument à la fonction native `iter()`, celle-ci fournit en retour un itérateur sur cet itérable. Cet itérateur n'est valable que pour une seule passe sur le jeu de valeurs. Lors de l'utilisation d'itérables, il n'est habituellement pas nécessaire d'appeler `iter()` ou de s'occuper soi-même des objets itérateurs. L'instruction `for` le fait automatiquement pour vous, créant une variable temporaire anonyme pour garder l'itérateur durant la boucle. Voir aussi *itérateur*, *séquence* et *générateur*.

**itérateur** Objet représentant un flux de donnée. Des appels successifs à la méthode `__next__()` de l'itérateur (ou le passer à la fonction native `next()`) donne successivement les objets du flux. Lorsque plus aucune donnée n'est disponible, une exception `StopIteration` est levée. À ce point, l'itérateur est épuisé et tous les appels suivants à sa méthode `__next__()` lèveront encore une exception `StopIteration`. Les itérateurs doivent avoir une méthode `__iter__()` qui renvoie l'objet itérateur lui-même, de façon à ce que chaque itérateur soit aussi itérable et puisse être utilisé dans la plupart des endroits où d'autres itérables sont attendus. Une exception notable est un code qui tente plusieurs itérations complètes. Un objet conteneur, (tel que `list`) produit un nouvel itérateur neuf à chaque fois qu'il est passé à la fonction `iter()` ou s'il est utilisé dans une boucle `for`. Faire ceci sur un itérateur donnerait simplement le même objet itérateur épuisé utilisé dans son itération précédente, le faisant ressembler à un conteneur vide.

Vous trouverez davantage d'informations dans `typeiter`.

**fonction clé** Une fonction clé est un objet callable qui renvoie une valeur à fins de tri ou de classement. Par exemple, la fonction locale `strxfrm()` est utilisée pour générer une clé de classement prenant en compte les conventions de classement spécifiques aux paramètres régionaux courants.

Plusieurs outils dans Python acceptent des fonctions clés pour déterminer comment les éléments sont classés ou groupés. On peut citer les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` et `itertools.groupby()`.

Il existe plusieurs moyens de créer une fonction clé. Par exemple, la méthode `str.lower()` peut servir de fonction clé pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clés avec des expressions `lambda`, comme `lambda r: (r[0], r[2])`. Vous noterez que le module `operator` propose des constructeurs de fonctions clefs : `attrgetter()`, `itemgetter()` et `methodcaller()`. Voir *Comment Trier* pour des exemples de création et d'utilisation de fonctions clefs.

**argument nommé** Voir *argument*.

**lambda** Fonction anonyme sous la forme d'une *expression* et ne contenant qu'une seule expression, exécutée lorsque la fonction est appelée. La syntaxe pour créer des fonctions lambda est : `lambda [parameters]: expression`

**LBYL** Regarde avant de sauter, (*Look before you leap* en anglais). Ce style de programmation consiste à vérifier des conditions avant d'effectuer des appels ou des accès. Ce style contraste avec le style *EAFP* et se caractérise par la présence de beaucoup d'instructions `if`.

Dans un environnement avec plusieurs fils d'exécution (*multi-threaded* en anglais), le style *LBYL* peut engendrer un séquençement critique (*race condition* en anglais) entre le "regarde" et le "sauter". Par exemple, le code `if key in mapping: return mapping[key]` peut échouer si un autre fil d'exécution supprime la clé `key` du *mapping* après le test mais avant l'accès. Ce problème peut être résolu avec des verrous (*locks*) ou avec l'approche *EAFP*.

**list** Un type natif de *sequence* dans Python. En dépit de son nom, une `list` ressemble plus à un tableau (*array* dans la plupart des langages) qu'à une liste chaînée puisque les accès se font en  $O(1)$ .

**liste en compréhension (ou liste en intention)** Écriture concise pour manipuler tout ou partie des éléments d'une séquence et renvoyer une liste contenant les résultats. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` génère la liste composée des nombres pairs de 0 à 255 écrits sous formes de chaînes de caractères et en hexadécimal (0x...). La clause `if` est optionnelle. Si elle est omise, tous les éléments du `range(256)` seront utilisés.

**chargeur** Objet qui charge un module. Il doit définir une méthode nommée `load_module()`. Un chargeur est typiquement donné par un *chercheur*. Voir la [PEP 302](#) pour plus de détails et `importlib.ABC.Loader` pour sa *classe de base abstraite*.

**méthode magique** Un synonyme informel de *special method*.

**tableau de correspondances** (*mapping* en anglais) Conteneur permettant de rechercher des éléments à partir de clés et implémentant les méthodes spécifiées dans les classes de base abstraites `collections.abc.Mapping` ou `collections.abc.MutableMapping`. Les classes suivantes sont des exemples de tableaux de correspondances : `dict`, `collections.defaultdict`, `collections.OrderedDict` et `collections.Counter`.

**chercheur dans les méta-chemins** Un *chercheur* renvoyé par une recherche dans `sys.meta_path`. Les chercheurs dans les méta-chemins ressemblent, mais sont différents des *chercheurs d'entrée dans path*.

Voir `importlib.abc.MetaPathFinder` pour les méthodes que les chercheurs dans les méta-chemins doivent implémenter.

**métaclass** Classe d'une classe. Les définitions de classe créent un nom pour la classe, un dictionnaire de classe et une liste de classes parentes. La métaclass a pour rôle de réunir ces trois paramètres pour construire la classe. La plupart des langages orientés objet fournissent une implémentation par défaut. La particularité de Python est la possibilité de créer des métaclasses personnalisées. La plupart des utilisateurs n'auront jamais besoin de cet outil, mais lorsque le besoin survient, les métaclasses offrent des solutions élégantes et puissantes. Elles sont utilisées pour journaliser les accès à des propriétés, rendre sûrs les environnements *multi-threads*, suivre la création d'objets, implémenter des singletons et bien d'autres tâches.

Plus d'informations sont disponibles dans : *metaclasses*.

**méthode** Fonction définie à l'intérieur d'une classe. Lorsqu'elle est appelée comme un attribut d'une instance de cette classe, la méthode reçoit l'instance en premier *argument* (qui, par convention, est habituellement nommé `self`). Voir *function* et *nested scope*.

**ordre de résolution des méthodes** L'ordre de résolution des méthodes (*MRO* pour *Method Resolution Order* en anglais) est, lors de la recherche d'un attribut dans les classes parentes, la façon dont l'interpréteur Python classe ces classes parentes. Voir [The Python 2.3 Method Resolution Order](#) pour plus de détails sur l'algorithme utilisé par l'interpréteur Python depuis la version 2.3.

**module** Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de nommage et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*.

Voir aussi *paquet*.

**spécificateur de module** Espace de nommage contenant les informations, relatives à l'importation, utilisées pour charger un module. C'est une instance de la classe `importlib.machinery.ModuleSpec`.

**MRO** Voir *ordre de résolution des méthodes*.

**muable** Un objet muable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

**n-uplet nommé** Le terme "n-uplet nommé" s'applique à tous les types ou classes qui héritent de la classe `tuple` et dont les éléments indexables sont aussi accessibles en utilisant des attributs nommés. Les types et classes peuvent avoir aussi d'autres caractéristiques.

Plusieurs types natifs sont appelés n-uplets, y compris les valeurs retournées par `time.localtime()` et `os.stat()`. Un autre exemple est `sys.float_info`:

```
>>> sys.float_info[1]                # indexed access
1024
>>> sys.float_info.max_exp           # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True
```

Certains *n-uplets nommés* sont des types natifs (comme les exemples ci-dessus). Sinon, un *n-uplet nommé* peut être créé à partir d'une définition de classe habituelle qui hérite de `tuple` et qui définit les champs nommés. Une telle classe peut être écrite à la main ou être créée avec la fonction `collections.namedtuple()`. Cette dernière méthode ajoute des méthodes supplémentaires qui ne seront pas trouvées dans celles écrites à la main ni dans les n-uplets nommés natifs.

**espace de nommage** L'endroit où une variable est stockée. Les espaces de nommage sont implémentés avec des dictionnaires. Il existe des espaces de nommage globaux, natifs ou imbriqués dans les objets (dans les méthodes). Les espaces de nommage favorisent la modularité car ils permettent d'éviter les conflits de noms. Par exemple, les fonctions `builtins.open` et `os.open()` sont différenciées par leurs espaces de nom. Les espaces de nommage aident aussi à la lisibilité et la maintenabilité en rendant clair quel module implémente une fonction. Par exemple, écrire `random.seed()` ou `itertools.islice()` affiche clairement que ces fonctions sont implémentées respectivement dans les modules `random` et `itertools`.

**paquet-espace de nommage** Un *paquet* tel que défini dans la [PEP 421](#) qui ne sert qu'à contenir des sous-paquets. Les paquets-espace de nommage peuvent n'avoir aucune représentation physique et, plus spécifiquement, ne sont pas comme un *paquet classique* puisqu'ils n'ont pas de fichier `__init__.py`.

Voir aussi *module*.

**portée imbriquée** Possibilité de faire référence à une variable déclarée dans une définition englobante. Typiquement, une fonction définie à l'intérieur d'une autre fonction a accès aux variables de cette dernière. Souvenez-vous cependant que cela ne fonctionne que pour accéder à des variables, pas pour les assigner. Les variables locales sont lues et assignées dans l'espace de nommage le plus proche. Tout comme les variables globales qui sont stockés dans l'espace de nommage global, le mot clef `nonlocal` permet d'écrire dans l'espace de nommage dans lequel est déclarée la variable.

**nouvelle classe** Ancien nom pour l'implémentation actuelle des classes, pour tous les objets. Dans les anciennes versions de Python, seules les nouvelles classes pouvaient utiliser les nouvelles fonctionnalités telles que `__slots__`, les descripteurs, les propriétés, `__getattr__()`, les méthodes de classe et les méthodes statiques.

**objet** N'importe quelle donnée comportant des états (sous forme d'attributs ou d'une valeur) et un comportement (des méthodes). C'est aussi (`object`) l'ancêtre commun à absolument toutes les *nouvelles classes*.

**paquet module** Python qui peut contenir des sous-modules ou des sous-paquets. Techniquement, un paquet est un module qui possède un attribut `__path__`.

Voir aussi *paquet classique* et *namespace package*.

**paramètre** Entité nommée dans la définition d'une *fonction* (ou méthode), décrivant un *argument* (ou dans certains cas des arguments) que la fonction accepte. Il existe cinq sortes de paramètres :

- *positional-or-keyword* : l'argument peut être passé soit par sa *position*, soit en tant que *argument nommé*. C'est le type de paramètre par défaut. Par exemple, `foo` et `bar` dans l'exemple suivant :

```
def func(foo, bar=None): ...
```

- *positional-only* : définit un argument qui ne peut être fourni que par position. Les paramètres *positional-only* peuvent être définis en insérant un caractère "/" dans la liste de paramètres de la définition de fonction après eux. Par exemple : *posonly1* et *posonly2* dans le code suivant :

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only* : l'argument ne peut être fourni que nommé. Les paramètres *keyword-only* peuvent être définis en utilisant un seul paramètre *var-positional*, ou en ajoutant une étoile (\*) seule dans la liste des paramètres avant eux. Par exemple, *kw\_only1* et *kw\_only2* dans le code suivant :

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional* : une séquence d'arguments positionnels peut être fournie (en plus de tous les arguments positionnels déjà acceptés par d'autres paramètres). Un tel paramètre peut être défini en préfixant son nom par une \*. Par exemple *args* ci-après :

```
def func(*args, **kwargs): ...
```

- *var-keyword* : une quantité arbitraire d'arguments peut être passée, chacun étant nommé (en plus de tous les arguments nommés déjà acceptés par d'autres paramètres). Un tel paramètre est défini en préfixant le nom du paramètre par \*\*. Par exemple, *kwargs* ci-dessus.

Les paramètres peuvent spécifier des arguments obligatoires ou optionnels, ainsi que des valeurs par défaut pour les arguments optionnels.

Voir aussi *argument* dans le glossaire, la question sur la différence entre les arguments et les paramètres dans la FAQ, la classe `inspect.Parameter`, la section *function* et la [PEP 362](#).

**entrée de chemin** Emplacement dans le *chemin des importations* (*import path* en anglais, d'où le *path*) que le *chercheur basé sur les chemins* consulte pour trouver des modules à importer.

**chercheur de chemins** *chercheur* renvoyé par un appelable sur un `sys.path_hooks` (c'est-à-dire un *point d'entrée pour la recherche dans path*) qui sait où trouver des modules lorsqu'on lui donne une *entrée de path*.

Voir `importlib.abc.PathEntryFinder` pour les méthodes qu'un chercheur d'entrée dans *path* doit implémenter.

**point d'entrée pour la recherche dans path** Appelable dans la liste `sys.path_hook` qui donne un *chercheur d'entrée dans path* s'il sait où trouver des modules pour une *entrée dans path* donnée.

**chercheur basé sur les chemins** L'un des *chercheurs dans les méta-chemins* par défaut qui cherche des modules dans un *chemin des importations*.

**objet simili-chemin** Objet représentant un chemin du système de fichiers. Un objet simili-chemin est un objet `str` ou un objet `bytes` représentant un chemin ou un objet implémentant le protocole `os.PathLike`. Un objet qui accepte le protocole `os.PathLike` peut être converti en un chemin `str` ou `bytes` du système de fichiers en appelant la fonction `os.fspath()`. `os.fsdecode()` et `os.fsencode()` peuvent être utilisées, respectivement, pour garantir un résultat de type `str` ou `bytes` à la place. A été Introduit par la [PEP 519](#).

**PEP** *Python Enhancement Proposal* (Proposition d'amélioration Python). Un PEP est un document de conception fournissant des informations à la communauté Python ou décrivant une nouvelle fonctionnalité pour Python, ses processus ou son environnement. Les PEP doivent fournir une spécification technique concise et une justification des fonctionnalités proposées.

Les PEPs sont censés être les principaux mécanismes pour proposer de nouvelles fonctionnalités majeures, pour recueillir les commentaires de la communauté sur une question et pour documenter les décisions de conception qui sont intégrées en Python. L'auteur du PEP est responsable de l'établissement d'un consensus au sein de la communauté et de documenter les opinions contradictoires.

Voir [PEP 1](#).

**portion** Jeu de fichiers dans un seul dossier (pouvant être stocké sous forme de fichier zip) qui contribue à l'espace de nommage d'un paquet, tel que défini dans la [PEP 420](#).

**argument positionnel** Voir *argument*.

**API provisoire** Une API provisoire est une API qui n'offre aucune garantie de rétrocompatibilité (la bibliothèque standard exige la rétrocompatibilité). Bien que des changements majeurs d'une telle interface ne soient pas attendus, tant qu'elle est étiquetée provisoire, des changements cassant la rétrocompatibilité (y compris sa suppression complète) peuvent survenir si les développeurs principaux le jugent nécessaire. Ces modifications ne surviendront que si de sérieux problèmes sont découverts et qu'ils n'avaient pas été identifiés avant l'ajout de l'API.

Même pour les API provisoires, les changements cassant la rétrocompatibilité sont considérés comme des "solutions de dernier recours". Tout ce qui est possible sera fait pour tenter de résoudre les problèmes en conservant la rétrocompatibilité.

Ce processus permet à la bibliothèque standard de continuer à évoluer avec le temps, sans se bloquer longtemps sur des erreurs d'architecture. Voir la [PEP 411](#) pour plus de détails.

**paquet provisoire** Voir *provisional API*.

**Python 3000** Surnom donné à la série des Python 3.x (très vieux surnom donné à l'époque où Python 3 représentait un futur lointain). Aussi abrégé *Py3k*.

**Pythonique** Idée, ou bout de code, qui colle aux idiomes de Python plutôt qu'aux concepts communs rencontrés dans d'autres langages. Par exemple, il est idiomatique en Python de parcourir les éléments d'un itérable en utilisant `for`. Beaucoup d'autres langages n'ont pas cette possibilité, donc les gens qui ne sont pas habitués à Python utilisent parfois un compteur numérique à la place :

```
for i in range(len(food)) :
    print(food[i])
```

Plutôt qu'utiliser la méthode, plus propre et élégante, donc *Pythonique* :

```
for piece in food:
    print(piece)
```

**nom qualifié** Nom, comprenant des points, montrant le "chemin" de l'espace de nommage global d'un module vers une classe, fonction ou méthode définie dans ce module, tel que défini dans la [PEP 3155](#). Pour les fonctions et classes de premier niveau, le nom qualifié est le même que le nom de l'objet :

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Lorsqu'il est utilisé pour nommer des modules, le *nom qualifié complet* (*fully qualified name - FQN* en anglais) signifie le chemin complet (séparé par des points) vers le module, incluant tous les paquets parents. Par exemple : `email.mime.text` :

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

**nombre de références** Nombre de références à un objet. Lorsque le nombre de références à un objet descend à zéro, l'objet est désalloué. Le comptage de référence n'est généralement pas visible dans le code Python, mais c'est un élément clé de l'implémentation *CPython*. Le module `sys` définit une fonction `getrefcount()` que les développeurs peuvent utiliser pour obtenir le nombre de références à un objet donné.

**paquet classique** *paquet* traditionnel, tel qu'un dossier contenant un fichier `__init__.py`.

Voir aussi *paquet-espace de nommage*.

**\_\_slots\_\_** Déclaration dans une classe qui économise de la mémoire en pré-allouant de l'espace pour les attributs des instances et qui élimine le dictionnaire (des attributs) des instances. Bien que populaire, cette technique est difficile à maîtriser et devrait être réservée à de rares cas où un grand nombre d'instances dans une application devient un sujet critique pour la mémoire.

**séquence** *itérable* qui offre un accès efficace à ses éléments par un indice sous forme de nombre entier via la méthode spéciale `__getitem__()` et qui définit une méthode `__len__()` donnant sa taille. Voici quelques séquences natives : `list`, `str`, `tuple`, et `bytes`. Notez que `dict` possède aussi une méthode `__getitem__()` et une méthode `__len__()`, mais il est considéré comme un *mapping* plutôt qu'une séquence, car ses accès se font par une clé arbitraire *immuable* plutôt qu'un nombre entier.

La classe abstraite de base `collections.abc.Sequence` définit une interface plus riche qui va au-delà des simples `__getitem__()` et `__len__()`, en ajoutant `count()`, `index()`, `__contains__()` et `__reversed__()`. Les types qui implémentent cette interface étendue peuvent s'enregistrer explicitement en utilisant `register()`.

**ensemble en compréhension (ou ensemble en intension)** Une façon compacte de traiter tout ou partie des éléments d'un itérable et de renvoyer un *set* avec les résultats. `results = {c for c in 'abracadabra' if c not in 'abc'}` génère l'ensemble contenant les lettres « r » et « d » { 'r', 'd' }. Voir *compréhensions*.

**distribution simple** Forme de distribution, comme les *fonction génériques*, où l'implémentation est choisie en fonction du type d'un seul argument.

**tranche** (*slice* en anglais), un objet contenant habituellement une portion de *séquence*. Une tranche est créée en utilisant la notation `[]` avec des `:` entre les nombres lorsque plusieurs sont fournis, comme dans `variable_name[1:3:5]`. Cette notation utilise des objets *slice* en interne.

**méthode spéciale** (*special method* en anglais) Méthode appelée implicitement par Python pour exécuter une opération sur un type, comme une addition. De telles méthodes ont des noms commençant et terminant par des doubles tirets bas. Les méthodes spéciales sont documentées dans *specialnames*.

**instruction** Une instruction (*statement* en anglais) est un composant d'un "bloc" de code. Une instruction est soit une *expression*, soit une ou plusieurs constructions basées sur un mot-clé, comme `if`, `while` ou `for`.

**encodage de texte** Codec (codeur-décodeur) qui convertit des chaînes de caractères Unicode en octets (classe *bytes*).

**fichier texte** *file object* capable de lire et d'écrire des objets `str`. Souvent, un fichier texte (*text file* en anglais) accède en fait à un flux de donnée en octets et gère l'*text encoding* automatiquement. Des exemples de fichiers textes sont les fichiers ouverts en mode texte ('r' ou 'w'), `sys.stdin`, `sys.stdout` et les instances de `io.StringIO`. Voir aussi *binary file* pour un objet fichier capable de lire et d'écrire *bytes-like objects*.

**chaîne entre triple guillemets** Chaîne qui est délimitée par trois guillemets simples (') ou trois guillemets doubles ("). Bien qu'elle ne fournisse aucune fonctionnalité qui ne soit pas disponible avec une chaîne entre guillemets, elle est utile pour de nombreuses raisons. Elle vous autorise à insérer des guillemets simples et doubles dans une chaîne sans avoir à les protéger et elle peut s'étendre sur plusieurs lignes sans avoir à terminer chaque ligne par un \. Elle est ainsi particulièrement utile pour les chaînes de documentation (*docstrings*).

**type** Le type d'un objet Python détermine quel genre d'objet c'est. Tous les objets ont un type. Le type d'un objet peut être obtenu via son attribut `__class__` ou via `type(obj)`.

**alias de type** Synonyme d'un type, créé en affectant le type à un identifiant.

Les alias de types sont utiles pour simplifier les *indications de types*. Par exemple :

```
def remove_gray_shades(
    colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pourrait être rendu plus lisible comme ceci :

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```



Voir `typing` et [PEP 484](#), qui décrivent cette fonctionnalité.

**indication de type** Le *annotation* qui spécifie le type attendu pour une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour.

Les indications de type sont facultatives et ne sont pas indispensables à l'interpréteur Python, mais elles sont utiles aux outils d'analyse de type statique et aident les IDE à compléter et à réusiner (*code refactoring* en anglais) le code.

Les indicateurs de type de variables globales, d'attributs de classe et de fonctions, mais pas de variables locales, peuvent être consultés en utilisant `typing.get_type_hints()`.

Voir `typing` et [PEP 484](#), qui décrivent cette fonctionnalité.

**retours à la ligne universels** Une manière d'interpréter des flux de texte dans lesquels sont reconnues toutes les fins de ligne suivantes : la convention Unix `'\n'`, la convention Windows `'\r\n'` et l'ancienne convention Macintosh `'\r'`. Voir la [PEP 278](#) et la [PEP 3116](#), ainsi que la fonction `bytes.splitlines()` pour d'autres usages.

**annotation de variable** *annotation* d'une variable ou d'un attribut de classe.

Lorsque vous annotez une variable ou un attribut de classe, l'affectation est facultative :

```
class C:
    field: 'annotation'
```

Les annotations de variables sont généralement utilisées pour des *indications de types* : par exemple, cette variable devrait prendre des valeurs de type `int` :

```
count: int = 0
```

La syntaxe d'annotation de la variable est expliquée dans la section *annassign*.

Reportez-vous à *function annotation*, à la [PEP 484](#) et à la [PEP 526](#) qui décrivent cette fonctionnalité.

**environnement virtuel** Environnement d'exécution isolé (en mode coopératif) qui permet aux utilisateurs de Python et aux applications d'installer et de mettre à jour des paquets sans interférer avec d'autres applications Python fonctionnant sur le même système.

Voir aussi `venv`.

**machine virtuelle** Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *bytecode* produit par le compilateur de *bytecode*.

**Le zen de Python** Liste de principes et de préceptes utiles pour comprendre et utiliser le langage. Cette liste peut être obtenue en tapant `"import this"` dans une invite Python interactive.





---

### À propos de ces documents

---

Ces documents sont générés à partir de sources en [reStructuredText](#) par [Sphinx](#), un analyseur de documents spécialement conçu pour la documentation Python.

Le développement de la documentation et de ses outils est entièrement basé sur le volontariat, tout comme Python. Si vous voulez contribuer, allez voir la page [reporting-bugs](#) qui contient des informations pour vous y aider. Les nouveaux volontaires sont toujours les bienvenus !

Merci beaucoup à :

- Fred L. Drake, Jr., créateur des outils originaux de la documentation Python et rédacteur de la plupart de son contenu ;
- le projet [Docutils](#) pour avoir créé *reStructuredText* et la suite d'outils *Docutils* ;
- Fredrik Lundh pour son projet [Alternative Python Reference](#), dont Sphinx a pris beaucoup de bonnes idées.

## B.1 Contributeurs de la documentation Python

De nombreuses personnes ont contribué au langage Python, à sa bibliothèque standard et à sa documentation. Consultez [Misc/ACKS](#) dans les sources de la distribution Python pour avoir une liste partielle des contributeurs.

Ce n'est que grâce aux suggestions et contributions de la communauté Python que Python a une documentation si merveilleuse — Merci !



---

Histoire et licence

---

## C.1 Histoire du logiciel

Python a été créé au début des années 1990 par Guido van Rossum, au Stichting Mathematisch Centrum (CWI, voir <https://www.cwi.nl/>) au Pays-Bas en tant que successeur d'un langage appelé ABC. Guido est l'auteur principal de Python, bien qu'il inclut de nombreuses contributions de la part d'autres personnes.

En 1995, Guido continua son travail sur Python au Corporation for National Research Initiatives (CNRI, voir <https://www.cnri.reston.va.us/>) de Reston, en Virginie, d'où il diffusa plusieurs versions du logiciel.

En mai 2000, Guido et l'équipe de développement centrale de Python sont partis vers BeOpen.com pour former l'équipe BeOpen PythonLabs. En octobre de la même année, l'équipe de PythonLabs est partie vers Digital Creations (désormais Zope Corporation; voir <https://www.zope.com/>). En 2001, la Python Software Foundation (PSF, voir <https://www.python.org/psf/>) voit le jour. Il s'agit d'une organisation à but non lucratif détenant les droits de propriété intellectuelle de Python. Zope Corporation en est un sponsor.

Toutes les versions de Python sont Open Source (voir <https://www.opensource.org/> pour la définition d'Open Source). Historiquement, la plupart, mais pas toutes, des versions de Python ont également été compatible avec la GPL, le tableau ci-dessous résume les différentes versions.

Version	Dérivé de	Année	Propriétaire	Compatible avec la GPL ?
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	non
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et ultérieure	2.1.1	2001-maintenant	PSF	oui

**Note :** Compatible GPL ne signifie pas que nous distribuons Python sous licence GPL. Toutes les licences Python, excepté la licence GPL, vous permettent la distribution d'une version modifiée sans rendre open source ces changements. La licence « compatible GPL » rend possible la diffusion de Python avec un autre logiciel qui est lui, diffusé sous la licence GPL ; les licences « non-compatibles GPL » ne le peuvent pas.

---

Merci aux nombreux bénévoles qui ont travaillé sous la direction de Guido pour rendre ces versions possibles.

## C.2 Conditions générales pour accéder à, ou utiliser, Python

Le logiciel Python et sa documentation sont distribués sous *la licence d'utilisation PSF*.

Depuis Python 3.8.6, les exemples, recettes et autres codes présents dans la documentation sont sous la double licence d'utilisation PSF et *la licence Zero-Clause BSD*.

Certains logiciels faisant partie de Python sont soumis à d'autres licences. Ces licences sont incluses avec le code lié à celles-ci. Voir *Licences et remerciements pour les logiciels tiers* pour une liste non exhaustive de ces licences.

### C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.9.5

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),  
→and  
the Individual or Organization ("Licensee") accessing and otherwise using  
→Python  
3.9.5 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby  
grants Licensee a nonexclusive, royalty-free, world-wide license to  
→reproduce,  
analyze, test, perform and/or display publicly, prepare derivative works,  
distribute, and otherwise use Python 3.9.5 alone or in any derivative  
version, provided, however, that PSF's License Agreement and PSF's notice  
→of  
copyright, i.e., "Copyright © 2001-2021 Python Software Foundation; All  
→Rights  
Reserved" are retained in Python 3.9.5 alone or in any derivative version  
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or  
incorporates Python 3.9.5 or any part thereof, and wants to make the  
derivative work available to others as provided herein, then Licensee  
→hereby  
agrees to include in any such work a brief summary of the changes made to  
→Python  
3.9.5.
4. PSF is making Python 3.9.5 available to Licensee on an "AS IS" basis.  
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF  
EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION  
→OR  
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT  
→THE

USE OF PYTHON 3.9.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF  
 ↪ OF  
 MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach  
 ↪ of  
 its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any  
 ↪ relationship  
 of agency, partnership, or joint venture between PSF and Licensee. This  
 ↪ License  
 Agreement does not grant permission to use PSF trademarks or trade name in  
 ↪ a  
 trademark sense to endorse or promote products or services of Licensee, or  
 ↪ any  
 third party.
8. By copying, installing or otherwise using Python 3.9.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0

### LICENCE D'UTILISATION LIBRE BEOPEN PYTHON VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

(suite sur la page suivante)

(suite de la page précédente)

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

### C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

(suite sur la page suivante)

(suite de la page précédente)

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

## C.2.5 LICENCE BSD ZERO-CLAUSE POUR LE CODE DANS LA DOCUMENTATION DE PYTHON 3.9.5

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR

(suite sur la page suivante)

(suite de la page précédente)

OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

## C.3 Licences et remerciements pour les logiciels tiers

Cette section est une liste incomplète mais grandissante de licences et remerciements pour les logiciels tiers incorporés dans la distribution de Python.

### C.3.1 Mersenne twister

Le module `_random` inclut du code construit à partir d'un téléchargement depuis <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. Voici mot pour mot les commentaires du code original :

A C-program for MT19937, with initialization improved 2002/1/26.  
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`  
or `init_by_array(init_key, key_length)`.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,  
All rights reserved.

Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:

1. Redistributions of source code must retain the above copyright  
notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright  
notice, this list of conditions and the following disclaimer in the  
documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote  
products derived from this software without specific prior written  
permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS  
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT  
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR  
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR  
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,  
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,  
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR  
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF  
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING  
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS  
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)



### C.3.2 Interfaces de connexion (sockets)

Le module `socket` utilise les fonctions `getaddrinfo()` et `getnameinfo()` codées dans des fichiers source séparés et provenant du projet WIDE : <http://www.wide.ad.jp/>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### C.3.3 Interfaces de connexion asynchrones

Les modules `asynchat` et `asyncore` contiennent la note suivante :

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.4 Gestion de témoin (*cookie*)

Le module `http.cookies` contient la note suivante :

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

    All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

### C.3.5 Traçage d'exécution

Le module `trace` contient la note suivante :

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

### C.3.6 Les fonctions UUencode et UUdecode

Le module uu contient la note suivante :

Copyright 1994 by Lance Ellinghouse  
 Cathedral City, California Republic, United States of America.  
 All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:

- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

### C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

Le module xmlrpc.client contient la note suivante :

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB  
 Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS

(suite sur la page suivante)

(suite de la page précédente)

ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.8 test\_epoll

Le module `test_epoll` contient la note suivante :

Copyright (c) 2001–2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### C.3.9 Select kqueue

Le module `select` contient la note suivante pour l'interface *kqueue* :

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY

(suite sur la page suivante)

(suite de la page précédente)

OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### C.3.10 SipHash24

Le fichier `Python/pyhash.c` contient une implémentation par Marek Majkowski de l'algorithme *SipHash24* de Dan Bernstein. Il contient la note suivante :

```
<MIT License>
Copyright (c) 2013 Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
    https://github.com/majek/csiphash/

Solution inspired by code from:
    Samuel Neves (supercop/crypto_auth/siphash24/little)
    djb (supercop/crypto_auth/siphash24/little2)
    Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

### C.3.11 *strtod* et *dtoa*

Le fichier `Python/dtoa.c`, qui fournit les fonctions `dtoa` et `strtod` pour la conversion de *doubles* C vers et depuis les chaînes, est tiré d'un fichier du même nom par David M. Gay, actuellement disponible sur <http://www.netlib.org/fp/>. Le fichier original, tel que récupéré le 16 mars 2009, contient la licence suivante :

```
/* *****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 */
```

(suite sur la page suivante)

(suite de la page précédente)

```
*
*****/
```

### C.3.12 OpenSSL

Les modules `hashlib`, `posix`, `ssl`, et `crypt` utilisent la bibliothèque OpenSSL pour améliorer les performances, si elle est disponible via le système d'exploitation. Aussi les outils d'installation sur Windows et Mac OS X peuvent inclure une copie des bibliothèques d'OpenSSL, donc on colle une copie de la licence d'OpenSSL ici :

```
LICENSE ISSUES
=====
```

```
The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.
```

```
OpenSSL License
-----
```

```
/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 * endorse or promote products derived from this software without
 * prior written permission. For written permission, please contact
 * openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 * nor may "OpenSSL" appear in their names without prior written
 * permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 * acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *
```

(suite sur la page suivante)

(suite de la page précédente)

```
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
*
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/
```

Original SSLeay License

```
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
* notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in the
* documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
* must display the following acknowledgement:
* "This product includes cryptographic software written by
* Eric Young (eay@cryptsoft.com)"
* The word 'cryptographic' can be left out if the rouines from the library
```

(suite sur la page suivante)

(suite de la page précédente)

```
*   being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*   the apps directory (application code) you must include an acknowledgement:
*   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

### C.3.13 expat

Le module pyexpat est compilé avec une copie des sources d'*expat*, sauf si la compilation est configurée avec `--with-system-expat` :

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```



### C.3.14 libffi

Le module `_ctypes` est compilé en utilisant une copie des sources de la *libffi*, sauf si la compilation est configurée avec `--with-system-libffi` :

```
Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
`Software'), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED `AS IS', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

### C.3.15 zlib

Le module `zlib` est compilé en utilisant une copie du code source de *zlib* si la version de *zlib* trouvée sur le système est trop vieille pour être utilisée :

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly          Mark Adler
jloup@gzip.org            madler@alumni.caltech.edu
```

### C.3.16 cfuhash

L'implémentation des dictionnaires, utilisée par le module `tracemalloc` est basée sur le projet *cfuhash* :

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above
  copyright notice, this list of conditions and the following
  disclaimer in the documentation and/or other materials provided
  with the distribution.

* Neither the name of the author nor the names of its
  contributors may be used to endorse or promote products derived
  from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

### C.3.17 libmpdec

Le module `_decimal` est construit en incluant une copie de la bibliothèque *libmpdec*, sauf si elle est compilée avec `--with-system-libmpdec` :

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
```

(suite sur la page suivante)

(suite de la page précédente)

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### C.3.18 Ensemble de tests C14N du W3C

The C14N 2.0 test suite in the test package (Lib/test/xmltestdata/c14n-20/) was retrieved from the W3C website at <https://www.w3.org/TR/xml-c14n2-testcases/> and is distributed under the 3-clause BSD license :

Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),  
All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



## ANNEXE D

---

### Copyright

---

Python et cette documentation sont :

Copyright © 2001-2021 Python Software Foundation. Tous droits réservés.

Copyright © 2000 *BeOpen.com*. Tous droits réservés.

Copyright © 1995-2000 *Corporation for National Research Initiatives*. Tous droits réservés.

Copyright © 1991-1995 *Stichting Mathematisch Centrum*. Tous droits réservés.

---

Voir [Histoire et licence](#) pour des informations complètes concernant la licence et les permissions.



## Non alphabétique

..., [71](#)  
2to3, [71](#)  
>>>, [71](#)  
\_\_future\_\_, [75](#)  
\_\_slots\_\_, [82](#)

## A

alias de type, [82](#)  
annotation, [71](#)  
annotation de fonction, [75](#)  
annotation de variable, [83](#)  
API provisoire, [81](#)  
argument, [71](#)  
argument nommé, [77](#)  
argument positionnel, [80](#)  
arrêt de l'interpréteur, [77](#)  
attribut, [72](#)  
awaitable, [72](#)

## B

BDFL, [72](#)

## C

C-contiguous, [73](#)  
chaîne entre triple guillemets, [82](#)  
chargeur, [78](#)  
chemin des importations, [76](#)  
chercheur, [75](#)  
chercheur basé sur les chemins, [80](#)  
chercheur dans les méta-chemins, [78](#)  
chercheur de chemins, [80](#)  
classe, [73](#)  
classe de base abstraite, [71](#)  
code intermédiaire (bytecode), [73](#)  
coercition, [73](#)  
contigu, [73](#)  
coroutine, [73](#)  
CPython, [74](#)

## D

deallocation, object, [51](#)  
décorateur, [74](#)  
descripteur, [74](#)  
dictionnaire, [74](#)  
dictionnaire en compréhension (ou *dictionnaire en intension*), [74](#)  
distribution simple, [82](#)  
division entière, [75](#)  
docstring (*chaîne de documentation*), [74](#)  
duck-typing, [74](#)

## E

EAFP, [74](#)  
encodage de texte, [82](#)  
ensemble en compréhension (ou *ensemble en intension*), [82](#)  
entrée de chemin, [80](#)  
environnement virtuel, [83](#)  
espace de nommage, [79](#)  
expression, [75](#)  
expression génératrice, [76](#)

## F

f-string, [75](#)  
fichier binaire, [72](#)  
fichier texte, [82](#)  
finalization, of objects, [51](#)  
fonction, [75](#)  
fonction clé, [77](#)  
fonction coroutine, [73](#)  
fonction de base  
  repr, [52](#)  
fonction de rappel, [73](#)  
fonction générique, [76](#)  
Fortran contiguous, [73](#)

## G

générateur, [75](#)

générateur asynchrone, [72](#)  
 generator, [75](#)  
 generator expression, [76](#)  
 gestionnaire de contexte, [73](#)  
 gestionnaire de contexte asynchrone, [72](#)  
 GIL, [76](#)

## H

hachable, [76](#)

## I

IDLE, [76](#)  
 immuable, [76](#)  
 importateur, [77](#)  
 importing, [77](#)  
 indication de type, [83](#)  
 instruction, [82](#)  
 interactif, [77](#)  
 interprété, [77](#)  
 itérable, [77](#)  
 itérable asynchrone, [72](#)  
 itérateur, [77](#)  
 itérateur asynchrone, [72](#)  
 itérateur de générateur, [76](#)  
 itérateur de générateur asynchrone, [72](#)

## L

lambda, [78](#)  
 LBYL, [78](#)  
 Le zen de Python, [83](#)  
 list, [78](#)  
 liste en compréhension (*ou liste en intention*), [78](#)

## M

machine virtuelle, [83](#)  
 magic  
     method, [78](#)  
 métaclasse, [78](#)  
 method  
     magic, [78](#)  
     special, [82](#)  
 méthode, [78](#)  
 méthode magique, [78](#)  
 méthode spéciale, [82](#)  
 module, [78](#)  
 module d'extension, [75](#)  
 MRO, [79](#)  
 muable, [79](#)

## N

n-uplet nommé, [79](#)  
 nom qualifié, [81](#)  
 nombre complexe, [73](#)  
 nombre de références, [81](#)

nouvelle classe, [79](#)

## O

object  
     deallocation, [51](#)  
     finalization, [51](#)  
 objet, [79](#)  
 objet fichier, [75](#)  
 objet fichier-compatible, [75](#)  
 objet octet-compatible, [72](#)  
 objet simili-chemin, [80](#)  
 ordre de résolution des méthodes, [78](#)

## P

paquet, [79](#)  
 paquet classique, [81](#)  
 paquet provisoire, [81](#)  
 paquet-espace de nommage, [79](#)  
 paramètre, [79](#)  
 PEP, [80](#)  
 Philbrick, Geoff, [16](#)  
 point d'entrée pour la recherche dans  
     path, [80](#)  
 portée imbriquée, [79](#)  
 portion, [80](#)  
 PyArg\_ParseTuple(), [14](#)  
 PyArg\_ParseTupleAndKeywords(), [15](#)  
 pyc utilisant le hachage, [76](#)  
 PyErr\_Fetch(), [51](#)  
 PyErr\_Restore(), [51](#)  
 PyInit\_modulename (*fonction C*), [59](#)  
 PyObject\_CallObject(), [13](#)  
 Python 3000, [81](#)  
 Python Enhancement Proposals  
     PEP 1, [80](#)  
     PEP 278, [83](#)  
     PEP 302, [75](#), [78](#)  
     PEP 328, [75](#)  
     PEP 343, [73](#)  
     PEP 362, [72](#), [80](#)  
     PEP 411, [81](#)  
     PEP 420, [75](#), [80](#)  
     PEP 421, [79](#)  
     PEP 442, [52](#)  
     PEP 443, [76](#)  
     PEP 451, [75](#)  
     PEP 483, [76](#)  
     PEP 484, [71](#), [75](#), [83](#)  
     PEP 489, [11](#), [59](#)  
     PEP 492, [72](#), [73](#)  
     PEP 498, [75](#)  
     PEP 519, [80](#)  
     PEP 525, [72](#)  
     PEP 526, [71](#), [83](#)



- PEP 3116, [83](#)
- PEP 3155, [81](#)
- Pythonique, [81](#)
- PYTHONPATH, [59](#)

## R

- ramasse-miettes, [75](#)
- READ\_RESTRICTED, [54](#)
- READONLY, [54](#)
- repr
  - fonction de base, [52](#)
- RESTRICTED, [54](#)
- retours à la ligne universels, [83](#)

## S

- séquence, [82](#)
- special
  - method, [82](#)
- spécificateur de module, [78](#)
- string
  - object representation, [52](#)

## T

- tableau de correspondances, [78](#)
- tranche, [82](#)
- type, [82](#)
- type générique, [76](#)

## V

- variable de classe, [73](#)
- variable de contexte, [73](#)
- variable d'environnement
  - PYTHONPATH, [59](#)
- verrou global de l'interpréteur, [76](#)
- vue de dictionnaire, [74](#)

## W

- WRITE\_RESTRICTED, [54](#)