

---

# **Extending and Embedding Python**

***Version 2.7.16***

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

**octobre 07, 2019**

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



---

## Table des matières

---

<b>1</b>	<b>Étendre Python en C ou C++</b>	<b>3</b>
1.1	Un exemple simple . . . . .	3
1.2	Intermezzo : Les erreurs et exceptions . . . . .	5
1.3	Retour vers l'exemple . . . . .	7
1.4	The Module's Method Table and Initialization Function . . . . .	7
1.5	Compilation and Linkage . . . . .	9
1.6	Calling Python Functions from C . . . . .	9
1.7	Extracting Parameters in Extension Functions . . . . .	11
1.8	Keyword Parameters for Extension Functions . . . . .	12
1.9	Building Arbitrary Values . . . . .	13
1.10	Reference Counts . . . . .	14
1.11	Writing Extensions in C++ . . . . .	18
1.12	Providing a C API for an Extension Module . . . . .	18
<b>2</b>	<b>Defining New Types</b>	<b>23</b>
2.1	The Basics . . . . .	23
2.2	Type Methods . . . . .	50
<b>3</b>	<b>Construire les extensions C et C++ avec <i>distutils</i></b>	<b>61</b>
3.1	Distribuer vos modules d'extension . . . . .	62
<b>4</b>	<b>Construire des extensions C et C++ sur Windows</b>	<b>65</b>
4.1	Une approche « recette de cuisine » . . . . .	65
4.2	Différences entre Unix et Windows . . . . .	65
4.3	Utiliser les DLL en pratique . . . . .	66
<b>5</b>	<b>Intégrer Python dans une autre application</b>	<b>69</b>
5.1	Intégration de très haut niveau . . . . .	69
5.2	Au delà de l'intégration de haut niveau : survol . . . . .	70
5.3	Intégration pure . . . . .	70
5.4	Étendre un Python intégré . . . . .	73
5.5	Intégrer Python dans du C++ . . . . .	73
5.6	Compiler et Lier en environnement Unix ou similaire . . . . .	73
<b>A</b>	<b>Glossaire</b>	<b>75</b>
<b>B</b>	<b>À propos de ces documents</b>	<b>83</b>

B.1	Contributeurs de la documentation Python . . . . .	83
<b>C</b>	<b>Histoire et licence</b>	<b>85</b>
C.1	Histoire du logiciel . . . . .	85
C.2	Conditions générales pour accéder à, ou utiliser, Python . . . . .	86
C.3	Licences et Remerciements pour les logiciels inclus . . . . .	89
<b>D</b>	<b>Copyright</b>	<b>101</b>
	<b>Index</b>	<b>103</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](#).

---

**Note :** This guide only covers the basic tools for creating extensions provided as part of this version of CPython. Third party tools may offer simpler alternatives. Refer to the [binary extensions section](#) in the Python Packaging User Guide for more information.

---



---

## 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 possible directement en Python : Elles peuvent définir de nouveaux types natifs, et peuvent appeler des fonctions de bibliothèques C ou 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 `ctypes` 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.

---

### 1.1 Un exemple simple

Let's create an extension module called `spam` (the favorite food of Monty Python fans...) and let's say we want to create a Python interface to the C library function `system()`<sup>1</sup>. This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows :

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

---

1. An interface for this function already exists in the standard module `os` — it was chosen as a simple and straightforward example.

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`.)

La première ligne de notre fichier peut être :

```
#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.

---

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 Py_BuildValue("i", 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*.

For module functions, the *self* argument is `NULL` or a pointer selected while initializing the module (see `Py_InitModule4()`). For a method, it would point to the object instance.

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

## 1.2 Intermezzo : Les erreurs et exceptions

An important convention throughout the Python interpreter is the following : when a function fails, it should set an exception condition and return an error value (usually a *NULL* pointer). Exceptions are stored in a static global variable inside the interpreter ; if this variable is *NULL* no exception has occurred. A second global variable stores the « associated value » of the exception (the second argument to *raise*). A third variable contains the stack traceback in case the error originated in Python code. These three variables are the C equivalents of the Python variables `sys.exc_type`, `sys.exc_value` and `sys.exc_traceback` (see the section on module `sys` in the Python Library Reference). It is important to know about them to understand how errors are passed around.

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 de retour.

Lorsqu'une fonction *f* ayant appelé une autre fonction *g* détecte que cette dernière a échoué, *f* devrait donner une valeur d'erreur à son tour (habituellement *NULL* ou *-1*). *f* ne devrait *pas* appeler l'une des fonctions `PyErr_*`, l'une d'elles ayant déjà été appelée par *g*. La fonction appelant *f* est alors censée renvoyer aussi un code d'erreur à celle qui l'a appelée, toujours sans utiliser `PyErr_*`, et ainsi de suite. La cause la plus détaillée de l'erreur a déjà été signalée par la fonction l'ayant détectée en premier. Une fois l'erreur remontée à la boucle principale de l'interpréteur Python, il interrompt le code en cours d'exécution et essaie de trouver un gestionnaire d'exception spécifié par le développeur Python.

(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).

Every failing `malloc()` call must be turned into an exception — the direct caller of `malloc()` (or `realloc()`) must call `PyErr_NoMemory()` and return a failure indicator itself. All the object-creating functions (for example, `PyInt_FromLong()`) already do this, so this note is only relevant to those who call `malloc()` directly.

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;
```

and initialize it in your module's initialization function (`initspam()`) with an exception object (leaving out the error checking for now) :

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

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_INCREF(SpamError);
    PyModule_AddObject(m, "error", SpamError);
}
```

Notez que le nom de exception, côté 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` à devenir un pointeur défaillant. S'il devenait un pointeur défaillant, le C code qui lève l'exception peut engendrer un rejet central ou des effets secondaires inattendus.

Nous traiterons de l'utilisation de `PyMODINIT_FUNC` comme un type de retour 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)
{
    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);
}
```

## 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 qui 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);
```

Our `spam.system()` function must return the value of `sts` as a Python object. This is done using the function `Py_BuildValue()`, which is something like the inverse of `PyArg_ParseTuple()` : it takes a format string and an arbitrary number of C values, and returns a new Python object. More info on `Py_BuildValue()` is given later.

```
return Py_BuildValue("i", 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 le nom 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.

## 1.4 The Module's Method Table and Initialization Function

I promised to show how `spam_system()` is called from Python programs. First, we need to list its name and address in a « method table » :

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

Note the third entry (`METH_VARARGS`). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be `METH_VARARGS` or `METH_VARARGS | METH_KEYWORDS`; a value of 0 means that an obsolete variant of `PyArg_ParseTuple()` is used.

When using only `METH_VARARGS`, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via `PyArg_ParseTuple()` ; more information on this function is provided below.

The `METH_KEYWORDS` bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third `PyObject *` parameter which will be a dictionary of keywords. Use `PyArg_ParseTupleAndKeywords()` to parse the arguments to such a function.

The method table must be passed to the interpreter in the module's initialization function. The initialization function must be named `initname()`, where *name* is the name of the module, and should be the only non-static item defined in the module file :

```
PyMODINIT_FUNC
initspam(void)
{
    (void) Py_InitModule("spam", SpamMethods);
}
```

Note that `PyMODINIT_FUNC` declares the function as `void` return type, declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`.

When the Python program imports module `spam` for the first time, `initspam()` is called. (See below for comments about embedding Python.) It calls `Py_InitModule()`, which creates a « module object » (which is inserted in the dictionary `sys.modules` under the key `"spam"`), and inserts built-in function objects into the newly created module based upon the table (an array of `PyMethodDef` structures) that was passed as its second argument. `Py_InitModule()` returns a pointer to the module object that it creates (which is unused here). It may abort with a fatal error for certain errors, or return `NULL` if the module could not be initialized satisfactorily.

When embedding Python, the `initspam()` function is not called automatically unless there's an entry in the `_PyImport_Inittab` table. The easiest way to handle this is to statically initialize your statically-linked modules by directly calling `initspam()` after the call to `Py_Initialize()` :

```
int
main(int argc, char *argv[])
{
    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(argv[0]);

    /* Initialize the Python interpreter.  Required. */
    Py_Initialize();

    /* Add a static module */
    initspam();

    ...
}
```

An example may be found in the file `Demo/embed/demo.c` in the Python source distribution.

---

**Note :** Removing entries from `sys.modules` or importing compiled modules into multiple interpreters within a process (or following a `fork()` without an intervening `exec()`) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures. Note also that the `reload()` function can be used with extension modules, and will call the module initialization function (`initspam()` in the example), but will not load the module again if it was loaded from a dynamically loadable object file (`.so` on Unix, `.dll` on Windows).

---

A more substantial example module is included in the Python source distribution as `Modules/xxmodule.c`. This file may be used as a template or simply read as an example.

## 1.5 Compilation and Linkage

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 les extensions C et C++ avec distutils*) 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
```

## 1.6 Calling Python Functions from 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 */
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
Py_XDECREF(my_callback); /* Dispose of previous callback */
my_callback = temp;      /* Remember new callback */
/* Boilerplate to return "None" */
Py_INCREF(Py_None);
result = Py_None;
}
return result;
}
```

This function must be registered with the interpreter using the METH\_VARARGS flag; this is described in section *The Module's Method Table and Initialization Function*. The PyArg\_ParseTuple() function and its arguments are documented in section *Extracting Parameters in Extension Functions*.

The macros Py\_XINCRF() and Py\_XDECREF() increment/decrement the reference count of an object and are safe in the presence of NULL pointers (but note that temp will not be NULL in this context). More info on them in section *Reference Counts*.

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.

Before you do this, however, it is important to check that the return value isn't NULL. If it is, the Python function terminated by raising an exception. If the C code that called PyObject\_CallObject() is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling PyErr\_Clear(). For example :

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

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to PyObject\_CallObject(). In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function

object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code :

```
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.

You may also call a function with keyword arguments by using `PyObject_Call()`, which supports arguments and keyword arguments. As in the above example, we use `Py_BuildValue()` to construct the dictionary.

```
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);
```

## 1.7 Extracting Parameters in Extension Functions

The `PyArg_ParseTuple()` function is declared as follows :

```
int PyArg_ParseTuple(PyObject *arg, 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 !

Some example calls :

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

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) */
}
```

## 1.8 Keyword Parameters for Extension Functions

The `PyArg_ParseTupleAndKeywords()` function is declared as follows :

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

The *arg* and *format* parameters are identical to those of the `PyArg_ParseTuple()` function. The *kwdict* parameter is the dictionary of keywords received as the third parameter from the Python runtime. The *kwlist* parameter is a *NULL*-terminated list of strings which identify the parameters; the names are matched with the type information from *format* from left to right. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

**Note :** Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not

present in the *kwlist* will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick ([philbrick@hks.com](mailto:philbrick@hks.com)) :

```
#include "Python.h"

static PyObject *
keywarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    char *state = "a stiff";
    char *action = "vroom";
    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_INCREF(Py_None);

    return Py_None;
}

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

```
void
initkeywarg(void)
{
    /* Create the module and add the functions */
    Py_InitModule("keywarg", keywarg_methods);
}
```

## 1.9 Building Arbitrary Values

This function is the counterpart to `PyArg_ParseTuple()`. It is declared as follows :

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

It recognizes a set of format units similar to the ones recognized by `PyArg_ParseTuple()`, but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from 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.

Examples (to the left the call, to the right the resulting Python value) :

<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("ss", "hello", "world")</code>	<code>('hello', 'world')</code>
<code>Py_BuildValue("s#", "hello", 4)</code>	<code>'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}",                   "abc", 123, "def", 456)</code>	<code>{'abc': 123, 'def': 456}</code>
<code>Py_BuildValue("((ii)(ii)) (ii)",                   1, 2, 3, 4, 5, 6)</code>	<code>(( (1, 2), (3, 4)), (5, 6))</code>

## 1.10 Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions `malloc()` and `free()`. In C++, the operators `new` and `delete` are used with essentially the same meaning and we'll restrict the following discussion to the C case.

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.

Since Python makes heavy use of `malloc()` and `free()`, it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called *reference counting*. The principle is simple : every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of « automatic » to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn't need to call `free()` explicitly. (Another claimed advantage is an improvement in speed

or memory usage — this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions `malloc()` and `free()` are available — which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we'll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them so long as there are no finalizers implemented in Python (`__del__()` methods). When there are such finalizers, the detector exposes the cycles through the `gc` module (specifically, the `garbage` variable in that module). The `gc` module also 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) or by removing the definition of `WITH_CYCLE_GC` in the `pyconfig.h` header on other platforms. If the cycle detector is disabled in this way, the `gc` module will not be available.

### 1.10.1 Reference Counting in Python

There are two macros, `Py_INCREF(x)` and `Py_DECREF(x)`, which handle the incrementing and decrementing of the reference count. `Py_DECREF()` also frees the object when the count reaches zero. For flexibility, it doesn't call `free()` directly — rather, it makes a call through a function pointer in the object's *type object*. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains : when to use `Py_INCREF(x)` and `Py_DECREF(x)` ? Let's first introduce some terms. Nobody « owns » an object; however, you can *own a reference* to an object. An object's reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling `Py_DECREF()` when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned reference : pass it on, store it, or call `Py_DECREF()`. Forgetting to dispose of an owned reference creates a memory leak.

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>.

The advantage of borrowing over owning a reference is that you don't need to take care of disposing of the reference on all possible paths through the code — in other words, with a borrowed reference you don't run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over owning is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

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).

---

2. The metaphor of « borrowing » a reference is not completely correct : the owner still has a copy of the reference.

3. Checking that the reference count is at least 1 **does not work** — the reference count itself could be in freed memory and may thus be reused for another object !

## 1.10.2 Ownership Rules

Whenever an object reference is passed into or out of a function, it is part of the function's interface specification whether ownership is transferred with the reference or not.

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 `PyInt_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, `PyInt_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.

## 1.10.3 Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using `Py_DECREF()` on an unrelated object while borrowing a reference to a list item. For instance:

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

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

This function first borrows a reference to `list[0]`, then replaces `list[1]` with the value 0, and finally prints the borrowed reference. Looks harmless, right? But it's not!

Let's follow the control flow into `PyList_SetItem()`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let's suppose the original item 1 was an instance of a user-defined class, and let's further suppose that the class defined a `__del__()` method. If this class instance has a reference count of 1, disposing of it will call its `__del__()` method.

Since it is written in Python, the `__del__()` method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in `bug()`? You bet! Assuming that the list passed into `bug()` is accessible to

the `__del__()` method, it could execute a statement to the effect of `del list[0]`, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `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, PyInt_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...

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can't get in each other's way, because there is a global lock protecting Python's entire object space. However, it is possible to temporarily release this lock using the macro `Py_BEGIN_ALLOW_THREADS`, and to re-acquire it using `Py_END_ALLOW_THREADS`. This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one :

```
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! */
}
```

### 1.10.4 NULL Pointers

In general, functions that take object references as arguments do not expect you to pass them *NULL* pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return *NULL* only to indicate that an exception occurred. The reason for not testing for *NULL* arguments is that functions often pass the objects they receive on to other function — if each function were to test for *NULL*, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for *NULL* only at the « source : » when a pointer that may be *NULL* is received, for example, from `malloc()` or from a function that may raise an exception.

The macros `Py_INCREF()` and `Py_DECREF()` do not check for *NULL* pointers — however, their variants `Py_XINCREF()` and `Py_XDECREF()` do.

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.

The C function calling mechanism guarantees that the argument list passed to C functions (`args` in the examples) is

never *NULL* — in fact it guarantees that it is always a tuple<sup>4</sup>.

It is a severe error to ever let a *NULL* pointer « escape » to the Python user.

## 1.11 Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initialization functions) have to be declared using `extern "C"`. It is unnecessary to enclose the Python header files in `extern "C" { ... }` — they use this form already if the symbol `__cplusplus` is defined (all recent C++ compilers define this symbol).

## 1.12 Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type « collection » which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy : just write the functions (without declaring them `static`, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details of visibility depend on the operating system ; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet !

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared `static`, except for the module's initialization function, in order to avoid name clashes with other extension modules (as discussed in section *The Module's Method Table and Initialization Function*). And it means that symbols that *should* be accessible from other extension modules must be exported in a different way.

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.

There are many ways in which Capsules can be used to export the C API of an extension module. Each function could get its own Capsule, or all C API pointers could be stored in an array whose address is published in a Capsule. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

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 :

---

4. These guarantees don't hold when you use the « old » style calling convention — this is still found in much existing code.

```
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.

The following example demonstrates an approach that puts most of the burden on the writer of the exporting module, which is appropriate for commonly used library modules. It stores all C API pointers (just one in the example!) in an array of `void` pointers which becomes the value of a Capsule. The header file corresponding to the module provides a macro that takes care of importing the module and retrieving its C API pointers; client modules only have to call this macro before accessing the C API.

The exporting module is a modification of the `spam` module from section *Un exemple simple*. The function `spam.system()` does not call the C library function `system()` directly, but a function `PySpam_System()`, which would of course do something more complicated in reality (such as adding « spam » to every command). This function `PySpam_System()` is also exported to other extension modules.

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

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

The function `spam_system()` is modified in a trivial way :

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

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return Py_BuildValue("i", sts);
}
```

In the beginning of the module, right after the line

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

two more lines must be added :

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

The `#define` is used to tell the header file that it is being included in the exporting module, not a client module. Finally, the module's initialization function must take care of initializing the C API pointer array :

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

(suite sur la page suivante)

(suite de la page précédente)

```

m = Py_InitModule("spam", SpamMethods);
if (m == NULL)
    return;

/* 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 (c_api_object != NULL)
    PyModule_AddObject(m, "_C_API", c_api_object);
}

```

Note that `PySpam_API` is declared `static`; otherwise the pointer array would disappear when `initspam()` terminates!

The bulk of the work is in the header file `spammodule.h`, which looks like this :

```

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

/* Header file for spammodule */

/* 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);
}

```

(suite sur la page suivante)

(suite de la page précédente)

```

    return (PySpam_API != NULL) ? 0 : -1;
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */

```

All that a client module must do in order to have access to the function `PySpam_System()` is to call the function (or rather macro) `import_spam()` in its initialization function :

```

PyMODINIT_FUNC
initclient(void)
{
    PyObject *m;

    m = Py_InitModule("client", ClientMethods);
    if (m == NULL)
        return;
    if (import_spam() < 0)
        return;
    /* additional initialization can happen here */
}

```

The main disadvantage of this approach is that the file `spammodule.h` is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that Capsules offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a Capsule. The details are described in the Python/C API Reference Manual in the section capsules and in the implementation of Capsules (files `Include/pycapsule.h` and `Objects/pycapsule.c` in the Python source code distribution).

## Notes



---

## Defining New Types

---

As mentioned in the last chapter, Python allows the writer of an extension module to define new types that can be manipulated from Python code, much like strings and lists in core Python.

This is not hard ; the code for all extension types follows a pattern, but there are some details that you need to understand before you can get started.

---

**Note :** The way new types are defined changed dramatically (and for the better) in Python 2.2. This document documents how to define new types for Python 2.2 and later. If you need to support older versions of Python, you will need to refer to [older versions of this documentation](#).

---

### 2.1 The Basics

The Python runtime sees all Python objects as variables of type `PyObject*`. A `PyObject` is not a very magnificent object - it just contains the refcount and a pointer to the object's « type object ». This is where the action is ; the type object determines which (C) functions get called when, for instance, an attribute gets looked up on an object or it is multiplied by another object. These C functions are called « type methods ».

So, if you want to define a new object 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 :

```
#include <Python.h>

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

static PyTypeObject noddy_NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy",          /* tp_name */
```

(suite sur la page suivante)

(suite de la page précédente)

```

sizeof(noddy_NoddyObject), /* tp_basicsize */
0, /* tp_itemsize */
0, /* tp_dealloc */
0, /* tp_print */
0, /* tp_getattr */
0, /* tp_setattr */
0, /* tp_compare */
0, /* tp_repr */
0, /* tp_as_number */
0, /* tp_as_sequence */
0, /* tp_as_mapping */
0, /* tp_hash */
0, /* tp_call */
0, /* tp_str */
0, /* tp_getattro */
0, /* tp_setattro */
0, /* tp_as_buffer */
Py_TPFLAGS_DEFAULT, /* tp_flags */
"Noddy objects", /* tp_doc */
};

static PyMethodDef noddy_methods[] = {
    {NULL} /* Sentinel */
};

#ifdef PyMODINIT_FUNC /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy(void)
{
    PyObject* m;

    noddy_NoddyType.tp_new = PyType_GenericNew;
    if (PyType_Ready(&noddy_NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy", noddy_methods,
        "Example module that creates an extension type.");

    Py_INCREF(&noddy_NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&noddy_NoddyType);
}

```

Now that's quite a bit to take in at once, but hopefully bits will seem familiar from the last chapter.

The first bit that will be new is :

```

typedef struct {
    PyObject_HEAD
} noddy_NoddyObject;

```

This is what a Noddy object will contain—in this case, nothing more than every Python object contains, namely a refcount and a pointer to a type object. These are the fields the `PyObject_HEAD` macro brings in. The reason for the macro is to standardize the layout and to enable special debugging fields in debug builds. Note that there is no semicolon after the `PyObject_HEAD` macro; one is included in the macro definition. Be wary of adding one by accident; it's easy to do from habit, and your compiler might not complain, but someone else's probably will! (On Windows, MSVC is known to

call this an error and refuse to compile the code.)

For contrast, let's take a look at the corresponding definition for standard Python integers :

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

Moving on, we come to the crunch — the type object.

```
static PyTypeObject noddy_NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy",           /* tp_name */
    sizeof(noddy_NoddyObject), /* tp_basicsize */
    0,                       /* tp_itemsize */
    0,                       /* tp_dealloc */
    0,                       /* tp_print */
    0,                       /* tp_getattr */
    0,                       /* tp_setattr */
    0,                       /* tp_compare */
    0,                       /* tp_repr */
    0,                       /* tp_as_number */
    0,                       /* tp_as_sequence */
    0,                       /* tp_as_mapping */
    0,                       /* tp_hash */
    0,                       /* tp_call */
    0,                       /* tp_str */
    0,                       /* tp_getattro */
    0,                       /* tp_setattro */
    0,                       /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,     /* tp_flags */
    "Noddy objects",        /* tp_doc */
};
```

Now if you go and look up the definition of `PyTypeObject` in `object.h` you'll see that it 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.

This is so important that we're going to pick the top of it apart still further :

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

This line is a bit of a wart; what we'd like to write is :

```
PyVarObject_HEAD_INIT(&PyType_Type, 0)
```

as the type of a type object is « type », but this isn't strictly conforming C and some compilers complain. Fortunately, this member will be filled in for us by `PyType_Ready()`.

```
"noddy.Noddy",           /* tp_name */
```

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

```
>>> "" + noddy.new_noddy()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot add type "noddy.Noddy" to string
```

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 `noddy` and the type is `Noddy`, so we set the type name to `noddy.Noddy`. One side effect of using an undotted name is that the `pydoc` documentation tool will not list the new type in the module documentation.

```
sizeof(noddy_NoddyObject), /* tp_basicsize */
```

This is so that Python knows how much memory to allocate when you call `PyObject_New()`.

**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.

```
0, /* tp_itemsize */
```

This has to do with variable length objects like lists and strings. Ignore this for now.

Skipping a number of type methods that we don't provide, we set the class flags to `Py_TPFLAGS_DEFAULT`.

```
Py_TPFLAGS_DEFAULT, /* tp_flags */
```

All types should include this constant in their flags. It enables all of the members defined by the current version of Python.

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

```
"Noddy objects", /* tp_doc */
```

Now we get into the type methods, the things that make your objects different from the others. We aren't going to implement any of these in this version of the module. We'll expand this example later to have more interesting behavior.

For now, all we want to be able to do is to create new `Noddy` objects. To enable object creation, we have to provide a `tp_new` implementation. In this case, we can just use the default implementation provided by the API function `PyType_GenericNew()`. We'd like to just assign this to the `tp_new` slot, but we can't, for portability sake. On some platforms or compilers, we can't statically initialize a structure member with a function defined in another C module, so, instead, we'll assign the `tp_new` slot in the module initialization function just before calling `PyType_Ready()` :

```
noddy_NoddyType.tp_new = PyType_GenericNew;
if (PyType_Ready(&noddy_NoddyType) < 0)
    return;
```

All the other type methods are `NULL`, so we'll go over them later — that's for a later section !

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

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

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

```
PyModule_AddObject(m, "Noddy", (PyObject *) &noddy_NoddyType);
```

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

```
>>> import noddy
>>> mynoddy = noddy.Noddy()
```

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

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

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

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

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

That wasn't so hard, was it?

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

## 2.1.1 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, `noddy2` that adds these capabilities :

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

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

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

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

    }

    self->number = 0;
}

return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    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 Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)

```

(suite sur la page suivante)

(suite de la page précédente)

```

        return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

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

static PyTypeObject NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy", /* tp_name */
    sizeof(Noddy), /* tp_basicsize */
    0, /* tp_itemsize */
    (destructor)Noddy_dealloc, /* tp_dealloc */
    0, /* tp_print */
    0, /* tp_getattr */
    0, /* tp_setattr */
    0, /* tp_compare */
    0, /* tp_repr */
    0, /* tp_as_number */
    0, /* tp_as_sequence */
    0, /* tp_as_mapping */
    0, /* tp_hash */
    0, /* tp_call */
    0, /* tp_str */
    0, /* tp_getattro */
    0, /* tp_setattro */
    0, /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
        Py_TPFLAGS_BASETYPE, /* tp_flags */
    "Noddy objects", /* tp_doc */
    0, /* tp_traverse */
    0, /* tp_clear */
    0, /* tp_richcompare */
    0, /* tp_weaklistoffset */

```

(suite sur la page suivante)

(suite de la page précédente)

```

0,                /* tp_iter */
0,                /* tp_iternext */
Noddy_methods,    /* tp_methods */
Noddy_members,    /* tp_members */
0,                /* tp_getset */
0,                /* tp_base */
0,                /* tp_dict */
0,                /* tp_descr_get */
0,                /* tp_descr_set */
0,                /* tp_dictoffset */
(initproc)Noddy_init, /* tp_init */
0,                /* tp_alloc */
Noddy_new,        /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL} /* Sentinel */
};

#ifdef PyMODINIT_FUNC /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy2(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy2", module_methods,
        "Example module that creates an extension type.");

    if (m == NULL)
        return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}

```

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 name of the Noddy object structure has been shortened to Noddy. The type object name has been shortened to NoddyType.

The Noddy type now has three data attributes, *first*, *last*, and *number*. The *first* and *last* variables are Python strings containing first and last names. The *number* attribute is an integer.

The object structure is updated accordingly :

```
typedef struct {
    PyObject_HEAD
```

(suite sur la page suivante)

(suite de la page précédente)

```
PyObject *first;
PyObject *last;
int number;
} Noddy;
```

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
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject*)self);
}
```

which is assigned to the `tp_dealloc` member :

```
(destructor)Noddy_dealloc, /*tp_dealloc*/
```

This method decrements the reference counts of the two Python attributes. We use `Py_XDECREF()` here because the `first` and `last` members could be `NULL`. It then calls the `tp_free` member of the object's type to free the object's memory. Note that the object's type might not be `NoddyType`, because the object may be an instance of a subclass.

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

```
static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
        {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("");
        if (self->last == NULL)
        {
            Py_DECREF(self);
            return NULL;
        }

        self->number = 0;
    }

    return (PyObject *)self;
}
```

and install it in the `tp_new` member :

```
Noddy_new, /* tp_new */
```

The new member is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. See the paper titled « Unifying types and classes in Python » for a detailed discussion of the `__new__()` method. One reason to implement a new method is to assure the initial values of instance variables. In this case, we use the new method to make sure that the initial values of the members `first` and `last` are not `NULL`. If we didn't care whether the initial values were `NULL`, we could have used `PyType_GenericNew()` as our new method, as we did before. `PyType_GenericNew()` initializes all of the instance variable members to `NULL`.

The new method is a static method that is passed the type being instantiated and any arguments passed when the type was called, and that returns the new object created. New methods always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer methods. Note that if the type supports subclassing, the type passed may not be the type being defined. The new method calls the `tp_alloc` slot to allocate memory. We don'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.

---

**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 provide an initialization function :

```
static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    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.

```
(initproc)Noddy_init,          /* tp_init */
```

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created.

Unlike the new method, we can't guarantee that the initializer is called. The initializer isn't called when unpickling objects and it can be overridden. Our initializer accepts arguments to provide initial values for our instance. Initializers always accept positional and keyword arguments.

Initializers can 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 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. 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 not cause any calls back into our type's code
- when decrementing a reference count in a `tp_dealloc` handler when garbage-collections is not supported<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 Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};
```

and put the definitions in the `tp_members` slot :

```
Noddy_members, /* tp_members */
```

Each member definition has a member name, type, offset, access flags and documentation string. See the [Generic Attribute Management](#) 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, `name()`, that outputs the objects name as the concatenation of the first and last names.

```
static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;
```

(suite sur la page suivante)

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. Even if a type supports garbage collection, there are calls that can be made to « untrack » the object from garbage collection, however, these calls are advanced and not covered here.

(suite de la page précédente)

```

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

```

The method is implemented as a C function that takes a Noddy (or Noddy 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 Noddy_methods[] = {
    {"name", (PyCFunction)Noddy_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    },
    {NULL} /* Sentinel */
};

```

and assign them to the `tp_methods` slot :

```

Noddy_methods, /* tp_methods */

```

Note that we used the `METH_NOARGS` flag to indicate that the method is passed no arguments.

Finally, we'll make our type usable as a base class. 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 :

```
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /*tp_flags*/
```

We rename `initnoddy()` to `initnoddy2()` and update the module name passed to `Py_InitModule3()`.

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

```
from distutils.core import setup, Extension
setup(name="noddy", version="1.0",
      ext_modules=[
          Extension("noddy", ["noddy.c"]),
          Extension("noddy2", ["noddy2.c"]),
      ])
```

## 2.1.2 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 Noddy 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.

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

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

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

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
    }
}
```

(suite sur la page suivante)

(suite de la page précédente)

```

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", 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;
}

static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};

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

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }

    if (! PyString_Check(value)) {

```

(suite sur la page suivante)

(suite de la page précédente)

```

        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }

    Py_DECREF(self->first);
    Py_INCREF(value);
    self->first = value;

    return 0;
}

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

static int
Noddy_setlast(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }

    if (! PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }

    Py_DECREF(self->last);
    Py_INCREF(value);
    self->last = value;

    return 0;
}

static PyGetSetDef Noddy_getsetters[] = {
    {"first",
     (getter)Noddy_getfirst, (setter)Noddy_setfirst,
     "first name",
     NULL},
    {"last",
     (getter)Noddy_getlast, (setter)Noddy_setlast,
     "last name",
     NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

```

(suite sur la page suivante)

```

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

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

static PyTypeObject NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy", /* tp_name */
    sizeof(Noddy), /* tp_basicsize */
    0, /* tp_itemsize */
    (destructor)Noddy_dealloc, /* tp_dealloc */
    0, /* tp_print */
    0, /* tp_getattr */
    0, /* tp_setattr */
    0, /* tp_compare */
    0, /* tp_repr */
    0, /* tp_as_number */
    0, /* tp_as_sequence */
    0, /* tp_as_mapping */
    0, /* tp_hash */
    0, /* tp_call */
    0, /* tp_str */
    0, /* tp_getattro */
    0, /* tp_setattro */
    0, /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
        Py_TPFLAGS_BASETYPE, /* tp_flags */
    "Noddy objects", /* tp_doc */
    0, /* tp_traverse */
    0, /* tp_clear */
    0, /* tp_richcompare */
    0, /* tp_weaklistoffset */
    0, /* tp_iter */
    0, /* tp_iternext */
    Noddy_methods, /* tp_methods */
    Noddy_members, /* tp_members */
    Noddy_getseters, /* tp_getset */
    0, /* tp_base */

```

(suite sur la page suivante)

(suite de la page précédente)

```

0,                /* tp_dict */
0,                /* tp_descr_get */
0,                /* tp_descr_set */
0,                /* tp_dictoffset */
(initproc)Noddy_init, /* tp_init */
0,                /* tp_alloc */
Noddy_new,        /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL} /* Sentinel */
};

#ifdef PyMODINIT_FUNC /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnodd3(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("nodd3", module_methods,
        "Example module that creates an extension type.");

    if (m == NULL)
        return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
    
```

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 :

```

Noddy_getfirst(Noddy *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }

    if (! PyString_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
            "The first attribute value must be a string");
        return -1;
    }
}
    
```

(suite sur la page suivante)

(suite de la page précédente)

```
Py_DECREF(self->first);
Py_INCREF(value);
self->first = value;

return 0;
}
```

The getter function is passed a Noddy object and a « closure », which is 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 Noddy 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 the attribute value is not a string.

We create an array of PyGetSetDef structures :

```
static PyGetSetDef Noddy_getseters[] = {
    {"first",
     (getter)Noddy_getfirst, (setter)Noddy_setfirst,
     "first name",
     NULL},
    {"last",
     (getter)Noddy_getlast, (setter)Noddy_setlast,
     "last name",
     NULL},
    {NULL} /* Sentinel */
};
```

and register it in the tp\_getset slot :

```
Noddy_getseters, /* tp_getset */
```

to register our attribute getters and setters.

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

We also remove the member definitions for these attributes :

```
static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};
```

We also need to update the tp\_init handler to only allow strings<sup>3</sup> to be passed :

```
static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwargs)
{
    PyObject *first=NULL, *last=NULL, *tmp;
```

(suite sur la page suivante)

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.

(suite de la page précédente)

```
static char *kwlist[] = {"first", "last", "number", NULL};

if (! PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", 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;
}
```

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 deallocator, where there is the possibility that the initialization of these members failed in the constructor.

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.

## 2.1.3 Supporting cyclic garbage collection

Python has a cyclic-garbage collector 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 Noddy example, we allowed any kind of object to be stored in the `first` or `last` attributes<sup>4</sup>. This means that Noddy objects can participate in cycles :

```
>>> import noddy2
>>> n = noddy2.Noddy()
>>> l = [n]
>>> n.first = l
```

4. Even in the third version, we aren't guaranteed to avoid cycles. Instances of string subclasses are allowed and string subclasses could allow cycles even if normal strings don't.

This is pretty silly, but it gives us an excuse to add support for the cyclic-garbage collector to the Noddy example. To support cyclic garbage collection, types need to fill two slots and set a class flag that enables these slots :

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

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

static int
Noddy_traverse(Noddy *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;
}

static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

    tmp = self->first;
    self->first = NULL;
    Py_XDECREF(tmp);

    tmp = self->last;
    self->last = NULL;
    Py_XDECREF(tmp);

    return 0;
}

static void
Noddy_dealloc(Noddy* self)
{
    PyObject_GC_UnTrack(self);
    Noddy_clear(self);
    Py_TYPE(self)->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->last = PyString_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    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 Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
```

(suite sur la page suivante)

(suite de la page précédente)

```

    "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL} /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

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

static PyTypeObject NoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "noddy.Noddy", /* tp_name */
    sizeof(Noddy), /* tp_basicsize */
    0, /* tp_itemsize */
    (destructor)Noddy_dealloc, /* tp_dealloc */
    0, /* tp_print */
    0, /* tp_getattr */
    0, /* tp_setattr */
    0, /* tp_compare */
    0, /* tp_repr */
    0, /* tp_as_number */

```

(suite sur la page suivante)

(suite de la page précédente)

```

0,                                /* tp_as_sequence */
0,                                /* tp_as_mapping */
0,                                /* tp_hash */
0,                                /* tp_call */
0,                                /* tp_str */
0,                                /* tp_getattro */
0,                                /* tp_setattro */
0,                                /* tp_as_buffer */
Py_TPFLAGS_DEFAULT |
    Py_TPFLAGS_BASETYPE |
    Py_TPFLAGS_HAVE_GC,          /* tp_flags */
"Noddy objects",                 /* tp_doc */
(traverseproc)Noddy_traverse,    /* tp_traverse */
(inquiry)Noddy_clear,            /* tp_clear */
0,                                /* tp_richcompare */
0,                                /* tp_weaklistoffset */
0,                                /* tp_iter */
0,                                /* tp_iternext */
Noddy_methods,                  /* tp_methods */
Noddy_members,                  /* tp_members */
0,                                /* tp_getset */
0,                                /* tp_base */
0,                                /* tp_dict */
0,                                /* tp_descr_get */
0,                                /* tp_descr_set */
0,                                /* tp_dictoffset */
(initproc)Noddy_init,           /* tp_init */
0,                                /* tp_alloc */
Noddy_new,                      /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL} /* Sentinel */
};

#ifdef PyMODINIT_FUNC             /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy4(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy4", module_methods,
        "Example module that creates an extension type.");

    if (m == NULL)
        return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}

```

The traversal method provides access to subobjects that could participate in cycles :

```
static int
Noddy_traverse(Noddy *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 2.4 and higher provide a `Py_VISIT()` macro that automates calling visit functions. With `Py_VISIT()`, `Noddy_traverse()` can be simplified :

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

---

**Note :** Note that the `tp_traverse` implementation must name its arguments exactly *visit* and *arg* in order to use `Py_VISIT()`. This is to encourage uniformity across these boring implementations.

---

We also need to provide a method for clearing any subobjects that can participate in cycles.

```
static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

    tmp = self->first;
    self->first = NULL;
    Py_XDECREF(tmp);

    tmp = self->last;
    self->last = NULL;
    Py_XDECREF(tmp);

    return 0;
}
```

Notice the use of a temporary variable in `Noddy_clear()`. We use the temporary variable so that we can set each

member to *NULL* before decrementing its reference count. We do this because, as was discussed earlier, if the reference count drops to zero, we might cause code to run that calls back into the object. In addition, because we now support garbage collection, we also have to worry about code being run that triggers garbage collection. If garbage collection is run, our `tp_traverse` handler could get called. We can't take a chance of having `Noddy_traverse()` called when a member's reference count has dropped to zero and its value hasn't been set to *NULL*.

Python 2.4 and higher provide a `Py_CLEAR()` that automates the careful decrementing of reference counts. With `Py_CLEAR()`, the `Noddy_clear()` function can be simplified :

```
static int
Noddy_clear(Noddy *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}
```

Note that `Noddy_dealloc()` may call arbitrary functions through `__del__` method or weakref callback. It means circular GC can be triggered inside the function. Since GC assumes reference count is not zero, we need to untrack the object from GC by calling `PyObject_GC_UnTrack()` before clearing members. Here is reimplemented deallocator which uses `PyObject_GC_UnTrack()` and `Noddy_clear()`.

```
static void
Noddy_dealloc(Noddy* self)
{
    PyObject_GC_UnTrack(self);
    Noddy_clear(self);
    Py_TYPE(self)->tp_free((PyObject*)self);
}
```

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

```
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC, /* tp_flags */
```

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

## 2.1.4 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 `Shoddy` 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 shoddy
>>> s = shoddy.Shoddy(range(3))
>>> s.extend(s)
>>> print len(s)
6
>>> print s.increment()
1
>>> print s.increment()
2
```

```
#include <Python.h>

typedef struct {
    PyListObject list;
    int state;
} Shoddy;

static PyObject *
Shoddy_increment(Shoddy *self, PyObject *unused)
{
    self->state++;
    return PyInt_FromLong(self->state);
}

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

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

static PyTypeObject ShoddyType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "shoddy.Shoddy", /* tp_name */
    sizeof(Shoddy), /* tp_basicsize */
    0, /* tp_itemsize */
    0, /* tp_dealloc */
    0, /* tp_print */
    0, /* tp_getattr */
    0, /* tp_setattr */
    0, /* tp_compare */
    0, /* tp_repr */
    0, /* tp_as_number */
    0, /* tp_as_sequence */
    0, /* tp_as_mapping */
    0, /* tp_hash */
    0, /* tp_call */
    0, /* tp_str */
    0, /* tp_getattro */
    0, /* tp_setattro */
    0, /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
    Py_TPFLAGS_BASETYPE, /* tp_flags */
    0, /* tp_doc */
    0, /* tp_traverse */
    0, /* tp_clear */

```

(suite sur la page suivante)

(suite de la page précédente)

```

0,                /* tp_richcompare */
0,                /* tp_weaklistoffset */
0,                /* tp_iter */
0,                /* tp_iternext */
Shoddy_methods,   /* tp_methods */
0,                /* tp_members */
0,                /* tp_getset */
0,                /* tp_base */
0,                /* tp_dict */
0,                /* tp_descr_get */
0,                /* tp_descr_set */
0,                /* tp_dictoffset */
(initproc)Shoddy_init, /* tp_init */
0,                /* tp_alloc */
0,                /* tp_new */
};

PyMODINIT_FUNC
initshoddy(void)
{
    PyObject *m;

    ShoddyType.tp_base = &PyList_Type;
    if (PyType_Ready(&ShoddyType) < 0)
        return;

    m = Py_InitModule3("shoddy", NULL, "Shoddy module");
    if (m == NULL)
        return;

    Py_INCREF(&ShoddyType);
    PyModule_AddObject(m, "Shoddy", (PyObject *) &ShoddyType);
}

```

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

```

typedef struct {
    PyListObject list;
    int state;
} Shoddy;

```

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 Shoddy instance, its `PyObject*` pointer can be safely cast to both `PyListObject*` and `Shoddy*`.

```

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

```

In the `__init__` method for our type, we can see how to call through to the `__init__` method of the base type.

This pattern is important when writing a type with custom `new` and `dealloc` methods. The `new` method should not actually create the memory for the object with `tp_alloc`, that will be handled by the base class when calling its `tp_new`.

When filling out the `PyTypeObject()` for the `Shoddy` type, you see a slot for `tp_base()`. Due to cross platform compiler issues, you can't fill that field directly with the `PyList_Type()`; it can be done later in the module's `init()` function.

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

    ShoddyType.tp_base = &PyList_Type;
    if (PyType_Ready(&ShoddyType) < 0)
        return;

    m = Py_InitModule3("shoddy", NULL, "Shoddy module");
    if (m == NULL)
        return;

    Py_INCREF(&ShoddyType);
    PyModule_AddObject(m, "Shoddy", (PyObject *) &ShoddyType);
}
```

Before calling `PyType_Ready()`, the type structure must have the `tp_base` slot filled in. When we are deriving a new type, it is not necessary to fill out the `tp_alloc` slot with `PyType_GenericNew()` – the allocate 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 `Noddy` examples.

## 2.2 Type Methods

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
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrofunc tp_getattr;
    setattrofunc tp_setattr;
    cmpfunc tp_compare;
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

(suite sur la page suivante)

(suite de la page précédente)

```

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

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 */
long tp_flags;

char *tp_doc; /* Documentation string */

/* Assigned meaning in release 2.0 */
/* call function for all accessible objects */
traverseproc tp_traverse;

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

/* Assigned meaning in release 2.1 */
/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
long tp_weaklistoffset;

/* Added in release 2.2 */
/* 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;
long 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;
} 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; be sure your type initialization keeps the fields in the right order! It's often easiest to find an example that includes all the fields you need (even if they're initialized to 0) and then change the values to suit your new type.

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

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

```
int 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, lists) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```
char *tp_doc;
```

Here you can put a string (or its address) that you want returned when the Python script references `obj.__doc__` to retrieve the doc string.

Now we come to the basic type methods—the ones most extension types will implement.

### 2.2.1 Finalization and 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;
        int have_error = PyErr_Occurred() ? 1 : 0;

        if (have_error)
            PyErr_Fetch(&err_type, &err_value, &err_traceback);

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

        if (have_error)
            PyErr_Restore(err_type, err_value, err_traceback);

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

## 2.2.2 Object Presentation

In Python, there are three ways to generate a textual representation of an object : the `repr()` function (or equivalent back-tick syntax), the `str()` function, and the `print` statement. For most objects, the `print` statement is equivalent to the `str()` function, but it is possible to special-case printing to a `FILE*` if necessary; this should only be done if efficiency is identified as a problem and profiling suggests that creating a temporary string object to be written to a file is too expensive.

These handlers are all optional, and most types at most need to implement the `tp_str` and `tp_repr` handlers.

```

reprfunc tp_repr;
reprfunc tp_str;
printfunc tp_print;
    
```

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 PyString_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.

Here is a simple example :

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

The print function will be called whenever Python needs to « print » an instance of the type. For example, if “node” is an instance of type `TreeNode`, then the print function is called when Python code calls :

```
print node
```

There is a flags argument and one flag, `Py_PRINT_RAW`, and it suggests that you print without string quotes and possibly without interpreting escape sequences.

The print function receives a file object as an argument. You will likely want to write to that file object.

Here is a sample print function :

```
static int
newdatatype_print(newdatatypeobject *obj, FILE *fp, int flags)
{
    if (flags & Py_PRINT_RAW) {
        fprintf(fp, "<{newdatatype object--size: %d}>",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    else {
        fprintf(fp, "\\\"<{newdatatype object--size: %d}>\\\"",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    return 0;
}
```

## 2.2.3 Attribute Management

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;
/* ... */
getattrofunc tp_getattrofunc;    /* PyObject * version */
setattrofunc tp_setattrofunc;
```

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.

## Generic Attribute Management

Nouveau dans la version 2.2.

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. The name of the attributes must be known when `PyType_Ready()` is called.
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.

The tables are declared as three fields of the type object :

```
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*.

XXX Need to refer to some unified discussion of the structure fields, shared with the next section.

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 {
    char *name;
    int type;
    int offset;
    int flags;
    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.

XXX Need to move some of this to a shared section !

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

Constante	Signification
READONLY	Never writable.
RO	Shorthand for READONLY.
READ_RESTRICTED	Not readable in restricted mode.
WRITE_RESTRICTED	Not writable in restricted mode.
RESTRICTED	Not readable or writable in restricted mode.

An interesting advantage of using the `tp_members` table to build descriptors that are used at runtime is that any attribute defined this way can have an associated doc string simply by providing the text in the table. An application can use the introspection API to retrieve the descriptor from the class object, and get the doc string using its `__doc__` attribute.

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

## Type-specific Attribute Management

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. The value in showing this is two-fold : it demonstrates how basic attribute management can be done in a way that is portable to older versions of Python, and 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.

A likely way to handle this is (1) to implement a set of functions (such as `newdatatype_getSize()` and `newdatatype_setSize()` in the example below), (2) provide a method table listing these functions, and (3) provide a `getattr` function that returns the result of a lookup in that table. The method table uses the same structure as the `tp_methods` field of the type object.

Voici un exemple :

```
static PyMethodDef newdatatype_methods[] = {
    {"getSize", (PyCFunction)newdatatype_getSize, METH_VARARGS,
     "Return the current size."},
    {"setSize", (PyCFunction)newdatatype_setSize, METH_VARARGS,
     "Set the size."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    return Py_FindMethod(newdatatype_methods, (PyObject *)obj, name);
}
```

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)
{
    (void)PyErr_Format(PyExc_RuntimeError, "Read-only attribute: %s", name);
    return -1;
}
```

## 2.2.4 Object Comparison

```
cmpfunc tp_compare;
```

The `tp_compare` handler is called when comparisons are needed and the object does not implement the specific rich comparison method which matches the requested comparison. (It is always used if defined and the `PyObject_Compare()` or `PyObject_Cmp()` functions are used, or if `cmp()` is used from Python.) It is analogous to the `__cmp__()` method. This function should return `-1` if *obj1* is less than *obj2*, `0` if they are equal, and `1` if *obj1* is greater than *obj2*. (It was previously allowed to return arbitrary negative or positive integers for less than and greater than, respectively; as of Python 2.2, this is no longer allowed. In the future, other return values may be assigned a different meaning.)

A `tp_compare` handler may raise an exception. In this case it should return a negative value. The caller has to test for the exception using `PyErr_Occurred()`.

Here is a sample implementation :

```
static int
newdatatype_compare(newdatatypeobject * obj1, newdatatypeobject * obj2)
{
    long result;

    if (obj1->obj_UnderlyingDatatypePtr->size <
        obj2->obj_UnderlyingDatatypePtr->size) {
        result = -1;
    }
    else if (obj1->obj_UnderlyingDatatypePtr->size >
             obj2->obj_UnderlyingDatatypePtr->size) {
        result = 1;
    }
    else {
        result = 0;
    }
    return result;
}
```

## 2.2.5 Abstract Protocol Support

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 moderately pointless example :

```
static long
newdatatype_hash(newdatatypeobject *obj)
{
    long result;
    result = obj->obj_UnderlyingDatatypePtr->size;
    result = result * 3;
    return result;
}
```

```
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.

This function takes three arguments :

1. *arg1* is the instance of the data type which is the subject of the call. If the call is `obj1('hello')`, then *arg1* is `obj1`.
2. *arg2* is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.
3. *arg3* 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.

Here is a desultory example of the implementation of the call function.

```
/* Implement the call function.
 *   obj1 is the instance receiving the call.
 *   obj2 is a tuple containing the arguments to the call, in this
 *       case 3 strings.
 */
static PyObject *
newdatatype_call(newdatatypeobject *obj, PyObject *args, PyObject *other)
{
    PyObject *result;
    char *arg1;
    char *arg2;
    char *arg3;

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

XXX some fields need to be added here...

```
/* Added in release 2.2 */
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

These functions provide support for the iterator protocol. Any object which wishes to support iteration over its contents (which may be generated during iteration) must implement the `tp_iter` handler. Objects which are returned by a `tp_iter` handler must implement both the `tp_iter` and `tp_iternext` handlers. 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`.

For an object which represents an iterable collection, the `tp_iter` handler must return an iterator object. The iterator object is responsible for maintaining the state of the iteration. For collections which can support multiple iterators which do not interfere with each other (as lists and tuples do), a new iterator should be created and returned. Objects which can only be iterated over once (usually due to side effects of iteration) should implement this handler by returning a new reference to themselves, and should also implement the `tp_iternext` handler. File objects are an example of such an iterator.

Iterator objects should implement both handlers. The `tp_iter` handler should return a new reference to the iterator (this is the same as the `tp_iter` handler for objects which can only be iterated over destructively). The `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, it may return `NULL` without setting an exception or it may set `StopIteration`; avoiding the exception can yield slightly better performance. If an actual error occurs, it should set an exception and return `NULL`.

## 2.2.6 Weak Reference Support

One of the goals of Python's weak-reference implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on those objects which do not benefit by weak referencing (such as numbers).

For an object to be weakly referencable, the extension must include a `PyObject*` field in the instance structure for the use of the weak reference mechanism; it must be initialized to `NULL` by the object's constructor. It must also set the `tp_weaklistoffset` field of the corresponding type object to the offset of the field. For example, the instance type is defined with the following structure :

```
typedef struct {
    PyObject_HEAD
    PyClassObject *in_class;      /* The class object */
    PyObject      *in_dict;       /* A dictionary */
    PyObject      *in_weakreflist; /* List of weak references */
} PyInstanceObject;
```

The statically-declared type object for instances is defined this way :

```
PyTypeObject PyInstance_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "module.instance",

    /* Lots of stuff omitted for brevity... */

    Py_TPFLAGS_DEFAULT,          /* tp_flags */
    0,                           /* tp_doc */
    0,                           /* tp_traverse */
    0,                           /* tp_clear */
```

(suite sur la page suivante)

(suite de la page précédente)

```
0, /* tp_richcompare */
offsetof(PyInstanceObject, in_weakreflist), /* tp_weaklistoffset */
};
```

The type constructor is responsible for initializing the weak reference list to *NULL* :

```
static PyObject *
instance_new() {
    /* Other initialization stuff omitted for brevity */

    self->in_weakreflist = NULL;

    return (PyObject *) self;
}
```

The only further addition is that the destructor needs to call the weak reference manager to clear any weak references. This is only required if the weak reference list is non-*NULL* :

```
static void
instance_dealloc(PyInstanceObject *inst)
{
    /* Allocate temporaries if needed, but do not begin
       destruction just yet.
    */

    if (inst->in_weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) inst);

    /* Proceed with object destruction normally. */
}
```

## 2.2.7 More Suggestions

Remember that you can omit most of these functions, in which case you provide 0 as a value. There are type definitions for each of the functions you must provide. They are in `object.h` in the Python include directory that comes with the source distribution of Python.

In order to learn how to implement any specific method for your new data type, do the following : Download and unpack the Python source distribution. Go the `Objects` directory, then search the C source files for `tp_` plus the function you want (for example, `tp_print` or `tp_compare`). You will find examples of the function you want to implement.

When you need to verify that an object is an instance of the type you are implementing, use the `PyObject_TypeCheck()` function. A sample of its use might be something like the following :

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

## Notes

---

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

---

Starting in Python 1.4, Python provides, on Unix, a special make file for building make files for building dynamically-linked extensions and custom interpreters. Starting with Python 2.0, this mechanism (known as related to Makefile.pre.in, and Setup files) is no longer supported. Building custom interpreters was rarely used, and extension modules can be built using *distutils*.

Building an extension module using *distutils* requires that *distutils* is installed on the build machine, which is included in Python 2.x and available separately for Python 1.5. Since *distutils* also supports creation of binary packages, users don't necessarily need a compiler and *distutils* to install the 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 une quantité variable d'arguments nommés, l'exemple précédent n'en utilisant 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-construire les arguments de `setup()`, pour mieux structurer le script pilote. Dans l'exemple

précédent, l'argument `ext_modules` donné à `setup()` est une liste de modules d'extension, dont chacun est une instance de `Extension`. Dans l'exemple, l'instance définit une extension nommée `demo` qui est construite en compilant 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, `setup()` est appelée avec des méta-informations supplémentaires, ce qui est recommandé lorsque le paquet est amené à être distribué. Pour l'extension elle-même, elle donne des variables de préprocesseur (*defines*), dossiers de bibliothèques, et bibliothèques. En fonction du compilateur `distutils` peut lui donner ces informations de différentes manières, par exemple, sur Linux, on pourrait obtenir cette ligne :

```
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.

### 3.1 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 joints à la distribution source, c'est possible grâce à un fichier `MANIFEST.in`. Voir la documentation de *distutils* pour plus de détails.

Si la distribution source a été construite avec succès, les mainteneurs peuvent 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
```



---

## 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.

---

### 4.1 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 empaqueter 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.

### 4.2 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'import (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'import 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'import 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 lié, une bibliothèque d'import peut être générée, qui devra être utilisée pour tous 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'import 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.

## 4.3 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 lieur. 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'import 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 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 leur 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.

A simple demo of embedding Python can be found in the directory `Demo/embed/` of the source distribution.

**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.

### 5.1 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.

```
#include <Python.h>

int
main(int argc, char *argv[])
{
    Py_SetProgramName(argv[0]); /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                      "print 'Today is',ctime(time())\n");
    Py_Finalize();
    return 0;
}
```

La fonction `Py_SetProgramName()` devrait être appelée avant `Py_Initialize()` pour informer l'interpréteur des chemins vers les bibliothèque Python. Ensuite, l'interpréteur Python est initialisé avec `Py_Initialize()`, suivi par l'exécution d'un script Python, codé en dur, qui affiche la date et l'heure. Ensuite, l'interpréteur est arrêté par un appel à `Py_Finalize()`, puis le programme se termine. Dans un vrai programme, vous voudrez peut-être obtenir le script Python d'une autre source, peut-être d'une fonction d'un éditeur de texte, un fichier ou une base de données. Obtenir le code Python à partir d'un fichier se fait plus aisément avec `PyRun_SimpleFile()`, ce qui vous évite d'avoir à allouer la mémoire et d'y charger le contenu du fichier.

## 5.2 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 au 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.

## 5.3 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 :

```

#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 = PyString_FromString(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 = PyInt_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) {
                printf("Result of call: %ld\n", PyInt_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]);
        }
    }
}

```

(suite sur la page suivante)

(suite de la page précédente)

```

        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    }
    else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
        return 1;
    }
    Py_Finalize();
    return 0;
}

```

This code loads a Python script using `argv[1]`, and calls the function named in `argv[2]`. Its integer arguments are the other values of the `argv` array. If you compile and link this program (let's call the finished executable **call**), and use it to execute a Python script, such as :

```

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

```

After initializing the interpreter, the script is loaded using `PyImport_Import()`. This routine needs a Python string as its argument, which is constructed using the `PyString_FromString()` data conversion routine.

```

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.

## 5.4 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 Py_BuildValue("i", numargs);
}

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

Insert the above code just above the `main()` function. Also, insert the following two statements directly after `Py_Initialize()` :

```
numargs = argc;
Py_InitModule("emb", EmbMethods);
```

Ces deux lignes initialisent la variable `numarg`, et rend la fonction `emb.numargs()` accessible à l'interpréteur intégré. Avec ces ajouts, le script Python petit 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 a Python.

## 5.5 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++.

## 5.6 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.

To find out the required compiler and linker flags, you can execute the `pythonX.Y-config` script which is generated as part of the installation process (a `python-config` script may also be available). This script has several options, of which the following will be directly useful to you :

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

```
$ /opt/bin/python2.7-config --cflags
-I/opt/include/python2.7 -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -
↳Wstrict-prototypes
```

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

```
$ /opt/bin/python2.7-config --ldflags
-L/opt/lib/python2.7/config -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.

. . . The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

**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 subitement fausse (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`), les nombres (dans le module `numbers`), les flux (dans le module `io`). Vous pouvez créer vos propres ABC avec le module `abc`.

**argument** Une 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, et la question dans la FAQ à propos de la différence entre argument et paramètre.

**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*.

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

**Objet bytes-compatible** Un objet gérant le `bufferobjects`, comme les classes `str`, `bytearray`, ou `memoryview`. Les objets bytes-compatibles peuvent manipuler des données binaires et ainsi servir à leur compression, sauvegarde, ou envoi sur une socket. Certaines actions nécessitent que la donnée binaire soit modifiable, ce qui n'est pas possible avec tous les objets byte-compatibles.

**code intermédiaire (bytecode)** Le code source, en Python, est compilé en un bytecode, la représentation interne à CPython d'un programme Python. Le bytecode est stocké dans un fichier nommé `.pyc` ou `.pyo`. Ces caches permettent de charger les fichiers plus rapidement lors de la deuxième exécution (en évitant ainsi de recommencer la compilation en bytecode). On dit que ce *langage intermédiaire* est exécuté sur une *machine virtuelle* qui exécute des instructions machine pour chaque instruction du bytecode. Notez que le bytecode n'a pas vocation à fonctionner entre différentes machines virtuelle Python, encore moins entre différentes version de Python.

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

**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.

**classic class** Any class which does not inherit from `object`. See [new-style class](#). Classic classes have been removed in Python 3.

**coercition** The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `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](#).

**CPython** L'implémentation canonique du langage de programmation Python, tel que distribué sur [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** Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Pour plus d'informations sur les méthodes des descripteurs, consultez `descriptors`.

**dictionnaire** An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

**vue de dictionnaire** The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See `dict-views`.

**docstring** 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** A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statements* which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not 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.

**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, une socket réseau, ...). Les objets fichiers sont aussi appelés *file-like-objects* ou *streams*.

There are actually three categories of file objects : raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

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

**chercheur** An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See [PEP 302](#) for details.

**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 *function*.

**`__future__`** A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to 2. If the module in which it is executed had enabled *true division* by executing :

```
from __future__ import division
```

the expression `11/4` would evaluate to 2.75. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default :

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

**ramasse-miettes** (*garbage collection*) Le 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.

**générateur** A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a `for`-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

**expression génératrice** Expression qui donne un itérateur. Elle ressemble à une expression normale, suivie d'une expression `for` définissant une variable de boucle, un intervalle et une expression `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
```

**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.

**hachable** An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

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*.

Tous les types immuables fournis par Python sont hachables, et aucun type mutable (comme les listes ou les dictionnaires) ne l'est. Toutes les instances de classes définies par les utilisateurs sont hachables par défaut, elles

sont toutes différentes selon `__eq__`, sauf comparées à elles mêmes, et leur empreinte (*hash*) 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.

**integer division** Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to 2 in contrast to the 2.75 returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also `__future__`.

**importer** 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*.

**itérable** An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

**itérateur** An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container. 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 clef pour maîtriser comment les éléments sont triés ou groupés. Typiquement les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, et `itertools.groupby()`.

La méthode `str.lower()` peut servir en fonction clef pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clef au besoin avec des expressions `lambda`, comme `lambda r: (r[0], r[2])`. Finalement le module `operator` fournit des constructeurs de fonctions clef : `attrgetter()`, `itemgetter()`, et `methodcaller()`. Voir *Comment Trier* pour avoir des exemple de création et d'utilisation de fonctions clés.

**argument nommé** Voir *argument*.

**lambda** An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

**LBYL** Regarde avant de tomber, (*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 « tomber ». 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** A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is  $O(1)$ .

**liste en compréhension (ou liste en intension)** A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

**chargeur** An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

**magic method** An informal synonym for *special method*.

**Tableau de correspondances** Un conteneur permettant d'accéder à des éléments par clef et implémente les méthodes spécifiées dans `Mapping` ou `~collections.MutableMapping` :ref:`classes de base abstraites`. Les classes suivantes sont des exemples de `mapping` : `dict`, `collections.defaultdict`, `collections.OrderedDict`, et `collections.Counter`.

**métaclasses** 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 métaclasses 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'aura 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 sûr 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 noms et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*. Voir aussi *paquet*.

**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** (*named-tuple* en anglais) Classe qui, comme un *n-uplet* (*tuple* en anglais), a ses éléments accessibles par leur indice. Et en plus, les éléments sont accessibles par leur nom. Par exemple, `time.localtime()` donne un objet ressemblant à un *n-uplet*, dont `year` est accessible par son indice : `t[0]` ou par son nom : `t.tm_year`. Un *n-uplet nommé* peut être un type natif tel que `time.struct_time` ou il peut être construit comme une simple classe. Un *n-uplet nommé* complet peut aussi être créé via la fonction `collections.namedtuple()`. Cette dernière approche fournit automatiquement des fonctionnalités supplémentaires, tel qu'une représentation lisible comme `Employee(name='jones', title='programmer')`.

**espace de noms** The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

**portée imbriquée** The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

**nouvelle classe** Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattr__()`.

More information can be found in `newstyle`.

**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__`.

**paramètre** A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters :

- *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* : l'argument ne peut être donné que par sa position. Python n'a pas de syntaxe pour déclarer de tels paramètres, cependant des fonctions natives, comme `abs()`, en utilisent.
- *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.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

**PEP** Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.

**argument positionnel** Voir *argument*.

**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
```

**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.

**\_\_slots\_\_** A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

**séquence** An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

**tranche** An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

**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`.

**struct sequence** A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

**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)`.

**retours à la ligne universels** A manner of interpreting text streams in which all of the following are recognized as ending a line : the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See [PEP 278](#) and [PEP 3116](#), as well as `str.splitlines()` for an additional use.

**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.

**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 parti 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 <http://www.zope.com/>). En 2001, la Python Software Foundation (PSF, voir <http://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 supérieur	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 compatible 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

### C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.16

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),  
→and  
the Individual or Organization ("Licensee") accessing and otherwise using  
→Python  
2.7.16 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 2.7.16 alone or in any derivative  
version, provided, however, that PSF's License Agreement and PSF's notice  
→of  
copyright, i.e., "Copyright © 2001-2019 Python Software Foundation; All  
→Rights  
Reserved" are retained in Python 2.7.16 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 2.7.16 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  
2.7.16.
4. PSF is making Python 2.7.16 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 2.7.16 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.16  
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT  
→OF  
MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.16, 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 2.7.16, 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.
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

(suite sur la page suivante)

(suite de la page précédente)

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.
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

(suite sur la page suivante)

(suite de la page précédente)

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.3 Licences et Remerciements pour les logiciels inclus

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).
```

(suite sur la page suivante)

(suite de la page précédente)

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.

(suite sur la page suivante)

(suite de la page précédente)

```
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 Virgule flottante et contrôle d'exception

Le code source pour le module `fpectl` inclut la note suivante :

```
-----
/                               Copyright (c) 1996.                               \
|                               The Regents of the University of California.         |
|                               All rights reserved.                               |
|                                                                              |
|  Permission to use, copy, modify, and distribute this software for             |
|  any purpose without fee is hereby granted, provided that this en-             |
|  tire 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 work was produced at the University of California, Lawrence                |
|  Livermore National Laboratory under contract no. W-7405-ENG-48                 |
|  between the U.S. Department of Energy and The Regents of the                 |
|  University of California for the operation of UC LLNL.                        |
|                                                                              |
|                               DISCLAIMER                                          |
|                                                                              |
|  This software was prepared as an account of work sponsored by an              |
|  agency of the United States Government. Neither the United States              |
|  Government nor the University of California nor any of their em-              |
|  ployees, makes any warranty, express or implied, or assumes any               |
|  liability or responsibility for the accuracy, completeness, or                 |
|  usefulness of any information, apparatus, product, or process                 |
|  disclosed, or represents that its use would not infringe                     |
|  privately-owned rights. Reference herein to any specific commer-              |
|  cial products, process, or service by trade name, trademark,                  |
|  manufacturer, or otherwise, does not necessarily constitute or                |
|  imply its endorsement, recommendation, or favoring by the United              |
|  States Government or the University of California. The views and               |
|  opinions of authors expressed herein do not necessarily state or               |
|  reflect those of the United States Government or the University                |
|  of California, and shall not be used for advertising or product               |
|  endorsement purposes.                                                         |
\                                                                              /
-----
```

### C.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice :

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

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.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
    http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
    references to Ghostscript; clarified derivation from RFC 1321;
    now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
    added conditionalization for C++ compilation from Martin
    Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

### C.3.5 Interfaces de connexion asynchrones

Les modules asynchat et asyncore contiennent la note suivante :

```
Copyright 1996 by Sam Rushing

    All Rights Reserved
```

(suite sur la page suivante)

(suite de la page précédente)

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.3.6 Gestion de témoin (*cookie*)

The Cookie module contains the following notice :

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.3.7 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>

(suite sur la page suivante)

(suite de la page précédente)

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.3.8 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.3.9 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

The xmlrpclib module contains the following notice :

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 ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.10 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.3.11 Select queue

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
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

### C.3.12 *strtod* et *dtoa*

Le fichier `Python/dtoa.c`, qui fournit les fonctions `dtoa` et `strtod` pour la conversions de *doubles* C vers et depuis les chaînes, et 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.
 *
 * *****/
```

### C.3.13 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](mailto: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/)"
 *
 * 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
* 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
* 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
* 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.3.14 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.3.15 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:
```

(suite sur la page suivante)

(suite de la page précédente)

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.3.16 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) 1995-2010 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  
jloup@gzip.org

Mark Adler  
madler@alumni.caltech.edu

## ANNEXE D

---

### Copyright

---

Python et cette documentation sont :

Copyright © 2001-2019 Python Software Foundation. All rights reserved.

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-alphabetical

..., [75](#)

2to3, [75](#)

>>>, [75](#)

\_\_future\_\_, [78](#)

\_\_slots\_\_, [82](#)

## A

argument, [75](#)

argument nommé, [80](#)

argument positionnel, [81](#)

attribut, [76](#)

## B

BDFL, [76](#)

## C

chaîne entre triple guillemets, [82](#)

chargeur, [80](#)

chercheur, [77](#)

classe, [76](#)

classe de base abstraite, [75](#)

classic class, [76](#)

code intermédiaire (*bytecode*), [76](#)

coercition, [76](#)

CPython, [76](#)

## D

deallocation, object, [52](#)

décorateur, [76](#)

descripteur, [77](#)

dictionnaire, [77](#)

division entière, [78](#)

docstring, [77](#)

duck-typing, [77](#)

## E

EAFP, [77](#)

environnement virtuel, [82](#)

espace de noms, [81](#)

expression, [77](#)

expression génératrice, [78](#)

## F

finalization, of objects, [52](#)

fonction, [78](#)

fonction clé, [79](#)

fonction de base

repr, [53](#)

str, [53](#)

## G

générateur, [78](#)

generator, [78](#)

generator expression, [78](#)

gestionnaire de contexte, [76](#)

GIL, [78](#)

## H

hachable, [78](#)

## I

IDLE, [79](#)

immuable, [79](#)

importateur, [79](#)

importer, [79](#)

instruction, [82](#)

integer division, [79](#)

interactif, [79](#)

interprété, [79](#)

itérable, [79](#)

itérateur, [79](#)

## L

lambda, [80](#)

LYL, [80](#)

Le zen de Python, [82](#)

list, [80](#)

liste en compréhension (*ou liste en intension*), [80](#)

## M

machine virtuelle, [82](#)

magic

method, [80](#)

magic method, [80](#)

métaclasse, [80](#)

method

magic, [80](#)

special, [82](#)

méthode, [80](#)

méthode spéciale, [82](#)

module, [80](#)

module d'extension, [77](#)

MRO, [80](#)

muable, [80](#)

## N

nombre complexe, [76](#)

nombre de références, [82](#)

nouvelle classe, [81](#)

n-uplet nommé, [80](#)

## O

object

deallocation, [52](#)

finalization, [52](#)

objet, [81](#)

Objet bytes-compatible, [76](#)

objet fichier, [77](#)

objet fichier-compatible, [77](#)

ordre de résolution des méthodes, [80](#)

## P

paquet, [81](#)

paramètre, [81](#)

PEP, [81](#)

Philbrick, Geoff, [13](#)

portée imbriquée, [81](#)

PyArg\_ParseTuple(), [11](#)

PyArg\_ParseTupleAndKeywords(), [12](#)

PyErr\_Fetch(), [52](#)

PyErr\_Restore(), [52](#)

PyObject\_CallObject(), [10](#)

Python 3000, [81](#)

Python Enhancement Proposals

PEP 1, [81](#)

PEP 278, [82](#)

PEP 302, [77](#), [80](#)

PEP 328, [78](#)

PEP 343, [76](#)

PEP 3116, [82](#)

Pythonique, [81](#)

## R

ramasse-miettes, [78](#)

READ\_RESTRICTED, [56](#)

READONLY, [56](#)

repr

fonction de base, [53](#)

RESTRICTED, [56](#)

retours à la ligne universels, [82](#)

RO, [56](#)

## S

séquence, [82](#)

special

method, [82](#)

str

fonction de base, [53](#)

struct sequence, [82](#)

## T

Tableau de correspondances, [80](#)

tranche, [82](#)

type, [82](#)

## V

verrou global de l'interpréteur, [78](#)

vue de dictionnaire, [77](#)

## W

WRITE\_RESTRICTED, [56](#)